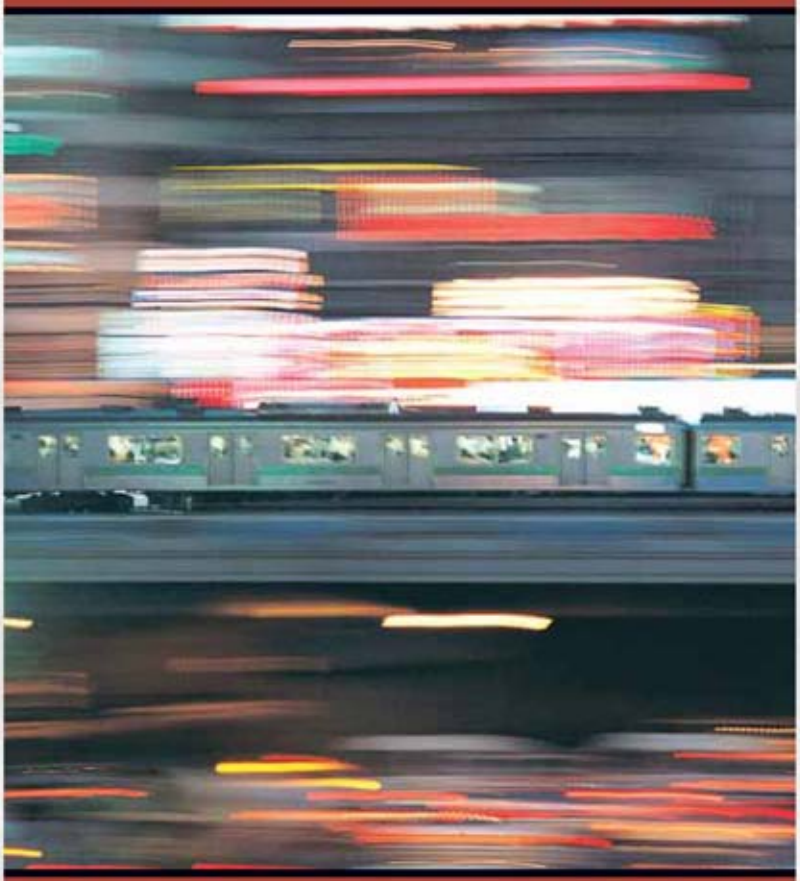
Resolución de Problemas *con* C++



Quinta edición

WALTER SAVITCH

PEARSON Addison Wesley

Resolución de problemas con C++

Quinta edición

Walter Savitch

Traducción:

Alfonso Vidal Romero Elizondo

Ingeniero en computación
Tecnológico de Monterrey, Campus Monterrey

Ana María Montañez Carrillo

Licenciada en Informática

Revisión técnica:

Fabiola Ocampo Botello

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México

Roberto De Luna Caballero

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México

PEARSON Educación ®

México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

Datos de catalogación bibliográfica

SAVITCH, WALTER
Resolución de problemas con C++

PEARSON EDUCACIÓN, México, 2007

ISBN 10: 970-26-0806-6
ISBN 13: 978-970-26-0806-6
Área: Ingeniería

Formato: 20 × 25.5 cm Páginas: 960

ISBN 0321268652

Edición en español:
Editor: Pablo Miguel Guerrero Rosas
e-mail: pablo.guerrero@pearsoned.com
Editor de desarrollo: Bernardino Gutiérrez Hernández
Supervisor de producción: Adriana Rida Montes

Edición en inglés:
Managing Editor Patty Mahtani
Executive Editor Susan Hartman Sullivan
Assistant Editor Elizabeth Paquin
Production Supervisor Marilyn Lloyd
Marketing Manager Michelle Brown
Production Services Argosy Publishing
Composition and Art Argosy Publishing
Copy Editor Nancy Kotary
Proofreader Kim Cofer
Indexer Larry Sweazy
Text Design Alisa Andreola/Dartmouth Publishing, Inc.
Cover Design Joyce Cosentino Wells
Cover and Interior Image ©2004 Photodisc
Prepress and Manufacturing Caroline Fell

QUINTA EDICIÓN, 2007

Atlacomulco 500-5o. piso
Industrial Atoto
53519, Naucalpan de Juárez, Edo. de México
E-mail: editorial.universidades@pearsoned.com

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

ISBN 10: 970-26-0806-6

ISBN 13: 978-970-26-0806-6

Impreso en México. *Printed in Mexico.*

1 2 3 4 5 6 7 8 9 0 - 09 08 07 06

Prefacio

Este libro está diseñado para utilizarse en un primer curso de programación o ciencias de la computación, usando el lenguaje C++. No se requiere de una experiencia previa de programación y bastan conocimientos de matemáticas a nivel de educación media.

Si ha utilizado la edición anterior de este libro, le recomiendo que lea la siguiente sección, en la que se explican los cambios en esta quinta edición; de esta manera podrá saltarse el resto del prefacio. Si es la primera vez que usa este libro, el resto del prefacio le será muy útil dado que presenta un panorama general.

Cambios en la quinta edición

Esta quinta edición presenta la misma filosofía de programación y cubre los mismos temas que la anterior edición; por lo que puede seguir impartiendo su curso sin necesidad de modificaciones. Se conservó mucho del material de la cuarta edición; pero se ha rediseñado la presentación, se han agregado proyectos de programación a cada capítulo, y también material adicional sobre estructuras de datos. Los principales temas que se agregaron fueron sobre las colas en el capítulo 15 y un capítulo completamente nuevo que trata acerca de la STL (Biblioteca de plantillas estándar).

Seleccione su propio orden de temas

La mayoría de los libros de texto introductorios que utilizan C++ presentan una agenda muy detallada que deben seguir los instructores al pie de la letra para usar el libro en sus cursos. En cambio, este libro se adapta a la manera en que usted enseñe, en vez de obligarlo a adaptarse al libro. Puede cambiar con facilidad el orden en el que deben cubrirse los capítulos sin pérdida de continuidad. Al final de este prefacio aparece un *diagrama de dependencias*, el cual le proporciona un panorama general de los órdenes que pueden usarse para cubrir los capítulos y las secciones; además, cada capítulo tiene una sección titulada "Prerrequisitos" en la que se explica qué partes del libro deben cubrirse antes de estudiar esa sección del capítulo. Esto permite a los instructores modificar con facilidad el orden en el que se deben cubrir los capítulos, o incluso las secciones.

Aunque este libro utilice bibliotecas y enseñe a los estudiantes la importancia de ellas, no se requieren otras que no sean las estándar. Este libro utiliza sólo bibliotecas que se proporcionan con la mayoría de las implementaciones de C++.

Introducción anticipada de clases

Este libro permite una cobertura flexible de las clases. Puede cubrirlas al principio del curso, o casi al final.

Existen por lo menos dos formas en que un libro puede introducir clases de manera anticipada: puede enseñar a los estudiantes cómo *diseñar* sus propias clases en sus primeros capítulos, o puede enseñarles tan sólo cómo utilizarlas sin definirlas. Este libro indica a los estudiantes cómo definir sus propias clases casi desde el principio, además de cómo usarlas. Para diseñar clases con efectividad, un estudiante necesita ciertas herramientas básicas, como las estructuras de control simples y las definiciones de las funciones. Por ende, empieza

a cubrir estos fundamentos en los capítulos 2, 3 y 4. Después pasa en forma inmediata a las clases. En el capítulo 5 se utilizan los flujos de E/S de archivos para decirle a los estudiantes cómo utilizar las clases. En el capítulo 6 los estudiantes aprenden a escribir sus propias clases.

Este libro utiliza una metodología ascendente para explicar las clases. Enseña a los estudiantes a escribir ciertas clases bastante simples, después agrega los constructores y la sobrecarga de operadores simples, luego la sobrecarga de los operadores de E/S << y >>, y así sucesivamente. Esta metodología evita que los estudiantes se abrumen con una larga lista de construcciones y conceptos complicados. No obstante, una de las metas de este libro es que los estudiantes escriban definiciones de clases realistas lo más pronto posible, y que no pierdan el tiempo escribiendo clases artificiales y simples. Al final del capítulo 8 los estudiantes estarán escribiendo, en esencia, los mismos tipos de clases que necesitarán cuando terminen el curso.

Los instructores que prefieran introducir las clases en un punto avanzado del curso podrán reorganizar el orden de los capítulos para adaptarlos a su metodología. Esto lo veremos en la sección titulada "Flexibilidad en el orden de los temas", más adelante en este prefacio.

En el capítulo 5 se cubre brevemente la herencia, de manera que los estudiantes puedan estar conscientes del concepto. Sin embargo, este libro no enseña a los estudiantes cómo escribir sus propias clases derivadas sino hasta más adelante, ya que los ejemplos que tienen una fuerte motivación para la herencia y las clases derivadas no se presentan con naturalidad al principio de un primer curso. El capítulo 16 indica a los estudiantes cómo definir y utilizar clases derivadas, incluyendo el uso de funciones virtuales. Algunos instructores podrían optar por dejar ese material para un segundo curso; otros querrán integrar esta cobertura de la herencia en su curso. Si lo desea, puede desplazar el material sobre herencia para verlo antes de los capítulos que le anteceden.

Accesibilidad para los estudiantes

No basta con que un libro presente los temas apropiados en el orden correcto. Ni siquiera basta con que sea claro y correcto cuando lo lea el instructor o algún otro programador experimentado: el material debe presentarse de una forma que sea accesible para los estudiantes principiantes. Por ser éste un texto introductorio, he puesto énfasis en escribir de una manera clara y amigable para los estudiantes. Los reportes de los muchos estudiantes que han utilizado las ediciones anteriores de este libro confirman que este estilo es uno de los más claros, e incluso hasta divertidos, para aprender.

Estándar ANSI/ISO de C++

Esta edición es 100% compatible con compiladores que cumplan con el estándar ANSI/ISO de C++ más reciente.

Temas avanzados

Muchos "temas avanzados" se están integrando a un curso estándar de ciencias computacionales de primer semestre. Aun si no forman parte de un curso, es conveniente que estén disponibles en el texto como material de enriquecimiento. Este libro ofrece una variedad de temas avanzados que algunos pueden integrarse en otro curso o dejarse como temas de enriquecimiento. Proporciona una amplia cobertura de las plantillas de C++, la herencia (incluyendo las funciones virtuales), el manejo de excepciones y la STL (Biblioteca de plantillas estándar).

Cuadros de resumen

Cada uno de los puntos principales se resume en una sección tipo cuadro. Estas secciones están distribuidas a lo largo de todos los capítulos.

Ejercicios de autoevaluación

Cada capítulo contiene muchos ejercicios de autoevaluación en puntos estratégicos, y al final de éste se proporcionan respuestas completas para todos ellos.

Probado en el salón de clases

Cientos de miles de estudiantes han utilizado las primeras cuatro ediciones de este libro. Muchos de ellos; así como sus instructores me han proporcionado sus opiniones y comentarios sobre lo que les funcionó y lo que no les funcionó. La mayoría de los comentarios fueron extremadamente positivos, ya que indican que a los estudiantes y sus maestros les gustó mucho el formato del libro, pero me hicieron sugerencias para ciertas modificaciones. Todas estas sugerencias se consideraron con cuidado. Esa valiosa retroalimentación se utilizó para revisar esta edición, de manera que se adapte a las necesidades de los estudiantes y de los instructores de una mejor manera.

Flexibilidad en el orden de los temas

Este libro se escribió de manera que los instructores tuvieran una gran flexibilidad en cuanto a la organización del material. Para ilustrar esta flexibilidad, sugerimos varias alternativas de cómo presentar los temas. No hay pérdida de continuidad cuando el libro se lee en cualquiera de estas maneras. Si piensa reorganizar el material, tal vez sea conveniente que mueva secciones en lugar de capítulos completos. No obstante, sólo se mueven ciertas secciones grandes en ubicaciones convenientes. Para ayudarle a personalizar un orden específico para las necesidades de cualquier clase, al final de este prefacio aparece un *diagrama de dependencias*, donde cada capítulo tiene una sección titulada “Prerrequisitos”, en la que se explica qué capítulos hay que cubrir antes de ver cada una de las secciones del libro.

Reorganización 1: Clases casi al final

Esta versión cubre en esencia todo un curso tradicional de ANSI C antes de cubrir el tema de las clases. Lo único que es muy parecido a C++ antes de introducir las clases es el uso de los flujos para la E/S. Como la E/S de flujos requiere cierto uso de espacios de nombres y bibliotecas de clases, se integra cierta cobertura mínima acerca de cómo utilizar los espacios de nombres predefinidos y ciertas clases de E/S de la biblioteca estándar en los primeros capítulos.

Fundamentos: capítulos 1, 2, 3, 4, 5 y 7 (se omite el capítulo 6, que trata de la definición de las clases). Este material cubre todas las estructuras de control, las definiciones de funciones y la E/S de archivos básica.

Arreglos: capítulo 10; se omite la sección 10.4, que utiliza clases.

Cadenas tipo C: sección 11.1.

Apuntadores y arreglos dinámicos: capítulo 12; se omite la última sección (12.3), que cubre material sobre las clases.

Recursión: capítulo 13; se omite el ejemplo de programación sobre clases con el que termina el capítulo. (De manera alternativa, se puede mover la recursión hacia un punto posterior en el curso.)

Estructuras y clases: capítulos 6, 8 y sección 11.2.

Compilación por separado y espacios de nombres: capítulo 9.

Apuntadores y listas enlazadas: capítulo 15.

También puede utilizar cualquier subconjunto de los siguientes capítulos:

Vectores: sección 11.3.

Plantillas: capítulo 14.

Herencia: capítulo 16.

Manejo de excepciones: capítulo 17.

Biblioteca de plantillas estándar: capítulo 18.

Reorganización 2: Clases un poco antes

Esta versión cubre todas las estructuras de control y el material básico acerca de los arreglos antes de ver las clases, pero éstas se cubren un poco antes que en la reorganización anterior.

Fundamentos: capítulos 1, 2, 3, 4, 5 y 7 (se omite el capítulo 6, que trata de la definición de las clases). Este material cubre todas las estructuras de control, las definiciones de funciones y la E/S de archivos básica.

Arreglos: capítulo 10; se omite la sección 10.4, que utiliza clases.

Cadenas tipo C: sección 11.1.

Estructuras y clases: capítulos 6, 8 y sección 11.2.

Compilación por separado y espacios de nombres: capítulo 9.

Apuntadores y arreglos dinámicos: capítulo 12.

Recursión: capítulo 13.

Apuntadores y listas enlazadas: capítulo 15.

También puede utilizarse cualquier subconjunto de los siguientes capítulos:

Vectores: sección 11.3.

Plantillas: capítulo 14.

Herencia: capítulo 16.

Manejo de excepciones: capítulo 17.

Biblioteca de plantillas estándar: capítulo 18.

Reorganización 3: las clases un poco antes, pero se cubren todas las estructuras de control antes de las clases

Esta reorganización es casi el orden normal del libro. Sólo necesita mover el capítulo 7 de tal forma que lo cubra antes del capítulo 6.

Material de apoyo

El libro cuenta con material de apoyo disponible para los usuarios de este libro, y material adicional disponible sólo para los instructores calificados.

Material disponible para los usuarios de este libro

Para acceder a estos materiales, vaya a


```
http://www.pearsoneducacion.net/savitch
```

- Se incluyen seis meses de acceso a Codemate, de Addison Wesley, con su nuevo libro de texto.
- Código fuente del libro (en inglés).

Recursos disponibles sólo para instructores calificados (en inglés)

En el sitio web: www.pearsoneducacion.net/savitch, encontrará los siguientes recursos:

- Acceso a CodeMate de Addison-Wesley.
- Guía de recursos para el instructor: incluye sugerencias de enseñanza para cada capítulo, cuestionarios con soluciones y soluciones para muchos proyectos de programación.
- Banco de pruebas y Generador de pruebas.
- Lecturas de PowerPoint: incluye los programas y el arte del libro.
- Manual de laboratorio.

Si requiere mayor información sobre el material de apoyo, contacte a su representante local de Pearson Educación.

Soporte para Visual C++

Además del material antes listado, este libro dispone de los siguientes materiales para Visual C++ (en inglés):

- Solución de problemas con C++: El objeto de la programación, 5ª Edición, edición para Visual C++ 6.0.
- Solución de problemas con C++: El objeto de la programación, 5ª Edición, edición para Visual C++ .NET.

Contáctenos

Me gustaría mucho conocer sus comentarios para seguir mejorando este libro y hacer que se adapte mejor a sus necesidades. Por favor envíe sus comentarios (en inglés) a la siguiente dirección de correo electrónico:

```
wsavitch@ucsd.edu
```

Quiero saber qué le parece el libro y si tiene sugerencias para algunas modificaciones; sin embargo, lamento informarle que no puedo proporcionar a los estudiantes un servicio de consultoría o asesoría vía correo electrónico. El volumen de correo electrónico ha aumentado demasiado como para poder dar este servicio. En especial, no puedo proporcionar

soluciones a los ejercicios de este libro ni a cualquier otro ejercicio que le proporcione su instructor. Es sólo que no tengo el tiempo suficiente como para responder a las numerosas peticiones que recibo en relación con ese tipo de asistencia. Además, no quiero interferir con los planes de ningún instructor acerca de cómo deben resolver sus estudiantes los problemas de programación. Como consuelo parcial a quienes desean tal ayuda, cabe señalar que este libro incluye las respuestas completas a todos los ejercicios de autoevaluación. Además, la guía del instructor (en inglés) proporciona a los instructores algunas de las respuestas a los Proyectos de programación, pero ese material está disponible sólo para los instructores que adopten el libro como texto.

Agradecimientos

Son muchas las personas y los grupos que me han proporcionado sugerencias, discusiones y demás ayuda para mejorar este texto. La mayor parte de la primera edición de este libro la escribí mientras visitaba el Departamento de Ciencias Computacionales de la Universidad de Colorado, en Boulder. El resto de esa primera edición y el trabajo en las ediciones subsiguientes lo realicé en el Departamento de Ingeniería y Ciencias computacionales de la Universidad de California en San Diego (UCSD). Estoy muy agradecido con estas instituciones por haber proporcionado un entorno conductivo para enseñar este material y escribir este libro.

Extiendo un agradecimiento especial a todos los individuos que contribuyeron con sus críticas para ésta o para las ediciones anteriores y los borradores del libro. En orden alfabético son: Claire Bono, Andrew Burt, Karla Chaveau, Joel Cohen, Doug Cosman, Scot Drysdale, Joe Faletti, Alex Feldman, Paulo Franca, Len Garrett, Jerrold Grossman, Eitan M. Gurari, Dennis Heckman, Bob Holloway, Matt Johnson, Bruce Johnston, Thomas Judson, Paul J. Kaiser, Michael Keenan, Paul Kube, Barney MacCabe, Steve Mahaney, Michael Main, Walter A. Manrique, Anne Marchant, John Marsaglia, Nat Martin, Bob Matthews, Jesse Morehouse, Donald Needham, Dung Nguyen, Joseph D. Oldham, Carol Roberts, Ken Rockwood, John Russo, Amber Settle, Naomi Shapiro, David Teague, Jerry Weltman, John J. Westman y Linda F. Wilson.

Deseo agradecer también a Nisar Hundewale por su trabajo en la actualización de los proyectos de programación.

Extiendo un agradecimiento especial a los muchos instructores que utilizaron ediciones anteriores de este texto. Sus comentarios integran parte de la revisión más útil que recibió el libro.

Agradezco a Prentice-Hall por haberme permitido adaptar cierto material de mi libro *Java: An Introduction to Computer Science and Programming* para que pudiera utilizar el material adaptado en el capítulo 17 del presente libro.

Agradezco a todas las personas de Addison-Wesley que contribuyeron tanto para que este libro fuera una realidad. En especial deseo mencionar a Patty Mahtani, Beth Paquin, Marilyn Lloyd, Michelle Neil, Joyce Wells, Michelle Brown y Jennifer Pelland. Un agradecimiento especial para todos ellos. Gracias a mi editora Susan Hartman Sullivan por su aliento y apoyo en esta edición y en las anteriores. El trabajo de estas finas personas fue invaluable para poder llevar a cabo esta quinta edición.

Agradezco a Daniel Rausch, Jen Jackowitz y a todas las personas en Argosy por su excelente trabajo para producir el libro final impreso.

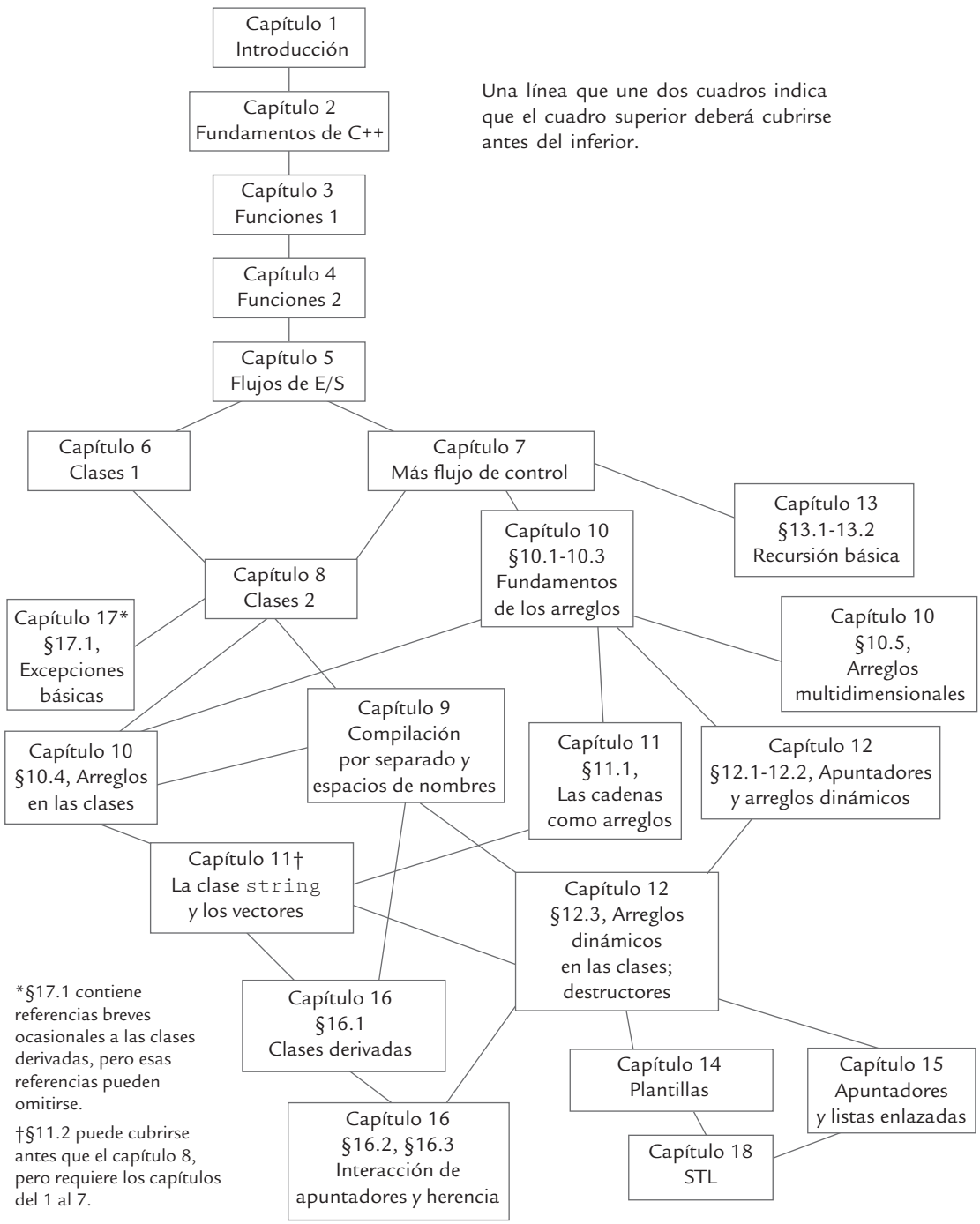
Walter Savitch

`http://www-cse.ucsd.edu/users/savitch/`

Diagrama de dependencias

El diagrama de dependencias que aparece en la siguiente página muestra los posibles órdenes de capítulos y subsecciones. La línea que une dos cuadros indica que debe cubrirse el tema del cuadro superior antes de estudiar el material del inferior. Cualquier secuencia que sea consistente con este ordenamiento parcial podrá seguirse sin pérdida de continuidad. Si un cuadro contiene uno o varios números de sección, significa que hace referencia sólo a esas secciones y no a todo el capítulo.

CUADRO P.1 *Diagrama de dependencia*



Resumen de contenido

Contenido

1 Introducción a las computadoras y a la programación en C++

1 Introducción a las computadoras y a la programación en C++

Todo el desarrollo y el manejo del análisis, ahora es posible a través de una máquina... En cuanto aparezca una máquina analítica, forzosamente guiará el curso de la ciencia.

CHARLES BABBAGE (*1792-1871*)

Introducción

En este capítulo describiremos los componentes básicos de una computadora, así como las técnicas básicas para diseñar y escribir un programa. Después presentaremos un programa de ejemplo en C++, y describiremos su funcionamiento.

1.1 Sistemas de computación

software
hardware

Al conjunto de instrucciones que debe seguir una computadora se le llama **programa**. A la colección de programas que utiliza una computadora se le conoce como **software**. La parte física de las máquinas, la cual constituye la instalación de la computadora, se conoce como **hardware**. Como veremos, el concepto de hardware es muy sencillo. Sin embargo, en la actualidad las computadoras vienen con una amplia variedad de software, que facilita la tarea de programar. Este software incluye editores, traductores y manejadores de varios tipos. El ambiente resultante es un sistema complicado y poderoso. En este libro nos ocuparemos casi exclusivamente del software, pero un leve vistazo a la organización del hardware nos será útil.

Hardware

PC, estaciones de trabajo y mainframes

Existen tres clases principales de computadoras: *computadoras personales*, *estaciones de trabajo* y *mainframes*. Una **PC** (**computadora personal**) es una computadora relativamente pequeña, diseñada para que sólo una persona la use en un momento determinado. La mayoría de las computadoras caseras son PC, aunque éstas también se utilizan mucho en los negocios, la industria y la ciencia. Una **estación de trabajo** es básicamente una PC más grande y más poderosa, a la que se podría considerar como una PC "industrial". Una **mainframe** es una computadora aún más grande que normalmente requiere un equipo de soporte, y por lo general la comparten varios usuarios. Las diferencias entre PC, estaciones de trabajo y mainframes no son precisas, pero estos términos se utilizan con frecuencia y nos proporcionan información general acerca de una computadora.

red

Una **red** consiste en varias computadoras conectadas entre sí, de modo que puedan compartir recursos como impresoras e información. Una red puede contener estaciones de trabajo y una o más mainframes, así como dispositivos compartidos, como impresoras.

En nuestro caso, que pretendemos aprender a programar, no tiene importancia si trabajamos en una PC, una mainframe o una estación de trabajo. Como veremos, la configuración básica de una computadora, desde nuestro punto de vista, es la misma en los tres casos.

El hardware de la mayoría de los sistemas de computación está organizado como se muestra en el cuadro 1.1. Podemos considerar que una computadora tiene cinco componentes principales: los *dispositivos de entrada*, los *dispositivos de salida*, el *procesador* (también conocido como *CPU*, *unidad central de procesamiento*), la *memoria principal* y la *memoria secundaria*. La CPU, la memoria principal, y algunas veces la memoria secundaria, normalmente se alojan en un solo gabinete. La CPU y la memoria principal forman el corazón de la computadora, y pueden considerarse como una unidad integrada. Otros componentes se conectan a la memoria principal y operan bajo la dirección de la CPU. Las flechas del cuadro 1.1 indican la dirección del flujo de información.

dispositivos de entrada

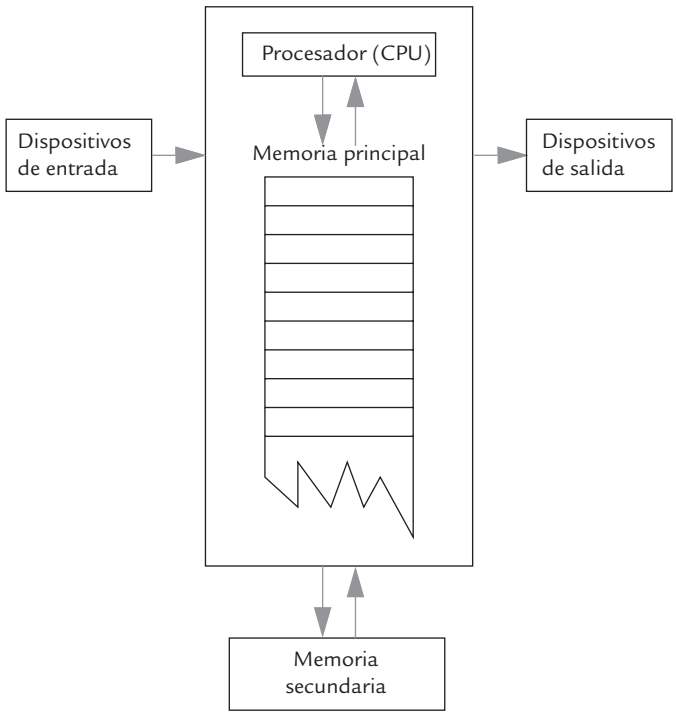
Un **dispositivo de entrada** es cualquier dispositivo que permite a una persona transmitir información a la computadora. Es probable que sus principales dispositivos de entrada sean un teclado y un ratón.

dispositivos de salida

Un **dispositivo de salida** es cualquiera que permita a la computadora transmitirnos información. El dispositivo de salida más común es la pantalla, también conocida como *monitor*. Con frecuencia hay más de un dispositivo de salida. Por ejemplo, además del monitor, es probable que su computadora esté conectada a una impresora para proporcionar una salida en papel. Con frecuencia, al teclado y al monitor se les considera como una sola unidad, llamada *terminal*.

Con el fin de almacenar las entradas y contar con el equivalente a hojas de papel para escribir cuando se realizan cálculos, las computadoras cuentan con *memoria*. El programa

CUADRO 1.1 *Componentes principales de una computadora*



que la computadora ejecuta se almacena bien en esta memoria. Una computadora tiene dos formas de memoria, llamadas *memoria principal* y *memoria secundaria*. El programa en ejecución se mantiene en la memoria principal, y ésta es, como su nombre lo indica, la memoria más importante. La **memoria principal** consiste en una larga lista de ubicaciones numeradas, conocidas como *ubicaciones de memoria*; el número de éstas varía de una computadora a otra, y puede ir desde algunos miles hasta varios millones, y en ocasiones hasta miles de millones. Cada ubicación de memoria contiene una cadena de ceros y unos. El contenido de estas ubicaciones puede cambiar, de modo que podemos considerar a cada ubicación de memoria como un diminuto pizarrón en el que la computadora puede escribir y borrar. En la mayoría de las computadoras, todas las ubicaciones de memoria contienen la misma cantidad de dígitos cero/uno. Un dígito que puede asumir sólo los valores cero o uno se conoce como **dígito binario** o **bit**. En casi todas las computadoras, las ubicaciones de memoria contienen ocho bits (o algún múltiplo de ocho bits). Una porción de memoria de ocho bits se conoce como **byte**, por lo que podemos llamar a estas ubicaciones numeradas de memoria principal como *bytes*. Para replantear la situación, podemos considerar a la memoria principal de la computadora como una larga lista de ubicaciones de memoria numeradas llamadas *bytes*. El número que identifica a un byte se conoce como su **dirección**. Un elemento de datos, por ejemplo un número o una letra, puede almacenarse en uno de estos bytes, y la dirección del byte se utiliza para localizar el dato cuando es necesario.

memoria principal

bit

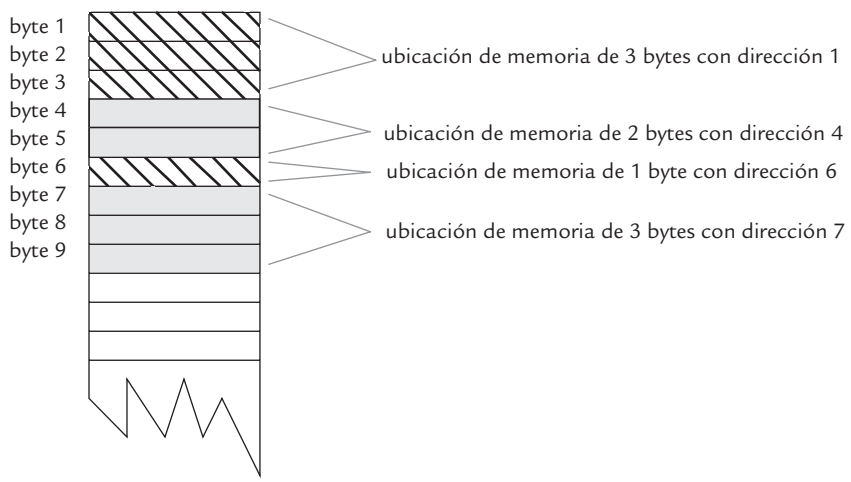
byte

dirección

ubicación de memoria

Si la computadora necesita manejar un dato que es demasiado grande para ocupar un solo byte (por ejemplo, un número grande), utilizará bytes adyacentes para almacenar dicho elemento. En este caso, el trozo completo de memoria que almacena ese dato se conoce como **ubicación de memoria**. La dirección del primer byte de esta gran ubicación de memoria se utiliza como su dirección. Por lo tanto, en la práctica podemos considerar a la memoria principal de una computadora como una larga lista de ubicaciones de memoria de *tamaño variable*. El tamaño de cada una de estas ubicaciones se expresa en bytes, y la dirección del primero se utiliza como la dirección (nombre) de esa ubicación de memoria. El cuadro 1.2 muestra una representación de la memoria principal de una computadora hipotética. Los tamaños de las ubicaciones de memoria no son fijos, y pueden cambiar cuando un nuevo programa se ejecuta en la computadora.

CUADRO 1.2 *Ubicaciones de memoria y bytes*



Bytes y direcciones

La memoria principal se divide en ubicaciones numeradas llamadas **bytes**. El número asociado a un byte es su **dirección**. Para guardar un dato, como un número o una letra, se emplea un número de bytes consecutivos. La dirección del primer byte del grupo se utiliza como la dirección de esta gran ubicación de memoria.

El hecho de que la información que se encuentra en la memoria de la computadora se represente mediante ceros y unos, no necesariamente tiene importancia para usted cuando programa en C++ (o en cualquier otro lenguaje de programación). Sin embargo, hay un aspecto sobre el uso de ceros y unos que sí nos concernirá tan pronto como comencemos a escribir programas. La computadora necesita interpretar estas cadenas de ceros y unos como números, letras, instrucciones u otro tipo de información. La computadora realiza automáticamente estas interpretaciones, de acuerdo con ciertos esquemas de codificación. Para cada tipo de elemento almacenado en la memoria de la computadora, se utiliza un

¿Por qué ocho?

Un **byte** es una ubicación de memoria que puede abarcar ocho bits. ¿Qué tiene de especial que sean ocho? ¿Por qué no diez bits? Hay dos razones por las que son ocho. En primer lugar, ocho es una potencia de 2. (8 es igual a 2^3.) Debido a que las computadoras utilizan bits, los cuales sólo tienen dos valores posibles, las potencias de dos son más convenientes que las potencias de 10. En segundo lugar, por la razón anterior se desprende que se necesitan ocho bits (un byte) para codificar un solo carácter (como una letra o algún otro símbolo del teclado).

código diferente: uno para letras, otro para números enteros, otro para fracciones, otro más para instrucciones y así sucesivamente. Por ejemplo, en un conjunto de códigos comúnmente utilizado, 01000001 es el código para la letra A y también para el número 65. Para saber lo que significa la cadena 01000001 en una ubicación en particular, la computadora debe rastrear el código que actualmente se está utilizando para esa ubicación. Por fortuna, el programador casi nunca necesitará preocuparse por dichos códigos, y puede con certeza razonar si las ubicaciones contienen letras, números o cualquier otra cosa.

La memoria que hemos explicado hasta este punto es la memoria principal. Sin la memoria principal, una computadora no puede hacer nada. Sin embargo, esta memoria en realidad sólo se utiliza mientras la computadora sigue las instrucciones de un programa. La computadora también tiene otro tipo de memoria, conocida como *memoria secundaria* o *almacenamiento secundario*. (En este contexto, las palabras memoria y almacenamiento son sinónimos.) La **memoria secundaria** es aquella que se utiliza para mantener un registro permanente de información después (y antes) de que la computadora se utilice. Algunos términos alternos que se utilizan con frecuencia para hacer referencia a la memoria secundaria son *memoria auxiliar*, *almacenamiento auxiliar*, *memoria externa* y *almacenamiento externo*.

memoria secundaria

La información que se encuentra en almacenamiento secundario se mantiene en unidades llamadas **archivos**, los cuales pueden ser tan grandes o tan pequeños como sea necesario. Por ejemplo, un programa se almacena en un archivo del almacenamiento secundario y se copia en la memoria principal cuando se ejecuta el programa. En un archivo podemos almacenar un programa, una carta, una lista de inventario o cualquier otra unidad de información.

archivos

Es posible agregar diversos tipos diferentes de memoria secundaria a una sola computadora. Las formas más comunes de memoria secundaria son *discos duros*, *diskettes* y *CD*. (Los **diskettes** también se conocen como *discos flexibles*.) Los **CD** (discos compactos) que se utilizan en las computadoras son básicamente los mismos que se utilizan para grabar y reproducir música. Los CD para computadoras pueden ser de sólo lectura, de tal forma que su computadora pueda leer pero no modificar la información del CD; también pueden ser de lectura/escritura, los cuales también pueden ser modificados por la computadora. La información se almacena en discos duros y diskettes básicamente de la misma forma que en CD. Los **discos duros** se fijan en un lugar, y normalmente no se mueven de la unidad de discos. Los diskettes y CD pueden sacarse fácilmente de la unidad de discos y llevarlos a otra computadora. Los diskettes y CD tienen la ventaja de no ser caros y de ser portátiles, aunque los discos duros tienen mayor capacidad para almacenar información y operan más rápidamente. También existen otras formas de memoria secundaria, pero esta lista abarca la mayoría de las formas que puede encontrar.

CD, discos y diskettes

Con frecuencia, a la memoria principal se le llama **memoria RAM** o **memoria de acceso aleatorio**. Se le llama *de acceso aleatorio* porque la computadora puede acceder de inmediato a la información en cualquier ubicación de memoria. La memoria secundaria a menudo requiere un **acceso secuencial**, lo que significa que la computadora debe buscar en todas (o al menos en muchas) ubicaciones de memoria, hasta que encuentre el dato que necesita.

RAM

El **procesador** (también conocido como **unidad central de procesamiento**, o **CPU**) es el "cerebro" de la computadora. Cuando se publicita una computadora, la empresa de

procesador

chip

computadoras le dice qué *chip* contiene. El **chip** es el procesador. El procesador sigue las instrucciones de un programa y realiza los cálculos especificados por él. Sin embargo, el procesador es un cerebro muy simple. Todo lo que puede hacer es cumplir un conjunto de instrucciones simples proporcionadas por el programador. Instrucciones típicas para un procesador dicen, por ejemplo, algo como "interpreta los ceros y unos como números, y luego suma el número que se encuentra en la ubicación de memoria 37 con el número que se encuentra en la ubicación de memoria 59, y coloca la respuesta en la ubicación 43", o "lee una letra de entrada, conviértela en su código como una cadena de ceros y unos, y colócala en la ubicación de memoria 1298". El procesador puede sumar, restar, multiplicar y dividir, y es capaz de transferir cosas de una ubicación de memoria a otra. Puede interpretar cadenas de ceros y unos como letras y enviarlas hacia un dispositivo de salida. El procesador también tiene cierta habilidad primitiva para reacomodar el orden de las instrucciones. Las instrucciones del procesador varían de alguna manera de una computadora a otra. El procesador de una computadora moderna puede tener cientos de instrucciones disponibles. Sin embargo, estas instrucciones normalmente son tan simples como las que acabamos de describir.

Software

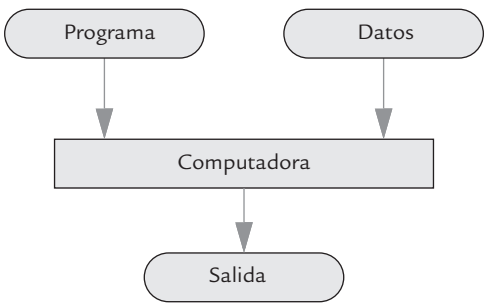
sistema operativo

Por lo general, no hablamos directamente con la computadora, sino que la comunicación se establece a través del *sistema operativo*. El **sistema operativo** distribuye los recursos de la computadora para que ésta cumpla con las tareas que debe realizar. El sistema operativo en realidad es un programa, pero tal vez sea mejor considerarlo como nuestro principal sirviente. Él está a cargo de todos los demás programas, y les entrega todas las solicitudes que usted realiza. Si desea ejecutar un programa, usted le indica al sistema operativo el nombre del archivo que lo contiene, y éste ejecuta el programa. Si queremos editar un archivo, le decimos al sistema operativo el nombre del archivo y él pone en marcha el editor para que trabaje en ese archivo. Para la mayoría de los usuarios, el sistema operativo es la computadora, ya que casi nunca ven a la computadora sin su sistema operativo. Los nombres de algunos sistemas operativos son *UNIX*, *DOS*, *Linux*, *Windows*, *Mac OS* y *VMS*

programa

Un **programa** es un conjunto de instrucciones que debe seguir una computadora. Como muestra el cuadro 1.3, podemos considerar que la entrada a una computadora consta

CUADRO 1.3 ***Vista simple de la ejecución de un programa***



de dos partes, un programa y algunos datos. La computadora sigue las instrucciones del programa, y de esa forma realiza algunos procesos. Los **datos** son lo que nosotros conceptualizamos como la entrada de un programa. Por ejemplo, si el programa suma dos números, entonces los dos números son los datos. En otras palabras los datos son la entrada al programa, y ambos, tanto el programa como los datos, se introducen a la computadora (normalmente a través del sistema operativo). Siempre que le proporcionamos a la computadora un programa a seguir y algunos datos para el programa, se dice que estamos **corriendo el programa** con esos datos, y se dice que la computadora **ejecuta el programa**. La palabra *datos* tiene un significado mucho más general que el que acabamos de dar. En el sentido más amplio significa cualquier información disponible para la computadora. La palabra datos normalmente se utiliza tanto en el sentido estricto como en el sentido más amplio.

datos

correr un programa

ejecutar un programa

Lenguajes de alto nivel

Existen muchos lenguajes para escribir programas. En este texto explicaremos el lenguaje de programación C++, y lo utilizaremos para escribir programas. C++ es un lenguaje de alto nivel, como lo son la mayoría de los lenguajes de programación más conocidos, como C, Java, Pascal, Visual Basic, FORTRAN, COBOL, Lisp, Scheme y Ada. Los **lenguajes de alto nivel** se asemejan en muchos sentidos a los lenguajes empleados por la humanidad. Están diseñados para que a la gente se le facilite escribir programas, y para que sea sencillo leerlos. Un lenguaje de alto nivel como C++ contiene instrucciones mucho más complicadas que las instrucciones sencillas que el procesador de una computadora (CPU) es capaz de seguir.

lenguaje de alto nivel

El tipo de lenguaje que una computadora puede entender se conoce como **lenguaje de bajo nivel**. Los detalles exactos de los lenguajes de bajo nivel varían de una computadora a otra. Una instrucción de bajo nivel típica podría ser la siguiente:

lenguaje de bajo nivel

```
ADD X Y Z
```

Esta instrucción podría significar "suma el número de la ubicación de memoria llamada X con el número de la ubicación de memoria llamada Y, y coloca el resultado en la ubicación de memoria llamada Z". La instrucción del ejemplo anterior está escrita en lo que conocemos como **lenguaje ensamblador**. Aunque el lenguaje ensamblador es casi el mismo que el que entiende la computadora, éste debe pasar por una traducción sencilla antes de que la computadora pueda entenderlo. Para lograr que una computadora siga una instrucción en lenguaje ensamblador, es necesario traducir las palabras en cadenas de ceros y unos. Por ejemplo, la palabra ADD puede traducirse como 0110, la X en 1001, la Y en 1010, y la Z en 1011. Entonces, la versión de la instrucción anterior que la computadora finalmente sigue es:

lenguaje ensamblador

```
0110 1001 1010 1011
```

Las instrucciones en lenguaje ensamblador y su traducción a ceros y unos difieren de una máquina a otra.

Se dice que los programas expresados en ceros y unos están escritos en **lenguaje de máquina**, ya que ésa es la versión del programa que la computadora (la máquina) en realidad lee y sigue. El lenguaje ensamblador y el de máquina son casi lo mismo, y la diferencia entre ellos no resulta importante para nosotros. La diferencia importante entre el lenguaje de máquina y los lenguajes de alto nivel como C++, es que cualquier programa escrito en un lenguaje de alto nivel debe traducirse a lenguaje de máquina, para que la computadora pueda entender y seguir el programa.

lenguaje de máquina

Compiladores

compilador

Un programa que traduce un lenguaje de alto nivel como C++ a un lenguaje de máquina, se llama **compilador**. Entonces un compilador es un tipo especial de programa, cuya entrada o datos es algún programa y su salida es otro programa. Para evitar confusiones, al programa de entrada normalmente se le llama **programa fuente** o **código fuente**, y la versión traducida que produce el compilador es llamada **programa objeto** o **código objeto**. La palabra **código** se utiliza con frecuencia para hacer referencia a un programa o a una parte de él, y este uso es particularmente común cuando se trata de programas objeto. Supongamos que queremos ejecutar un programa que hemos escrito en C++. Para lograr que la computadora siga las instrucciones en C++, hagamos lo siguiente. Primero, ejecutamos el compilador utilizando nuestro programa en C++ como entrada. Observe que en este caso, nuestro programa en C++ no se toma como un conjunto de instrucciones. Para el compilador, dicho programa es sólo una larga cadena de caracteres. La salida será otra larga cadena de caracteres, que es el equivalente al lenguaje de máquina de nuestro programa en C++. Después, ejecutamos este programa en lenguaje de máquina, que es lo que normalmente consideramos como los datos del programa en C++. La salida será lo que normalmente conceptualizamos como la salida del programa en C++. Es más fácil visualizar el proceso básico si contamos con dos computadoras, como muestra el cuadro 1.4. En realidad, todo el proceso se lleva a cabo utilizando dos veces la misma computadora.

programa fuente
programa objeto
código

Compilador

Un **compilador** es un programa que traduce un programa escrito en lenguaje de alto nivel, como C++, a un programa en lenguaje de máquina, que la computadora puede entender y ejecutar de manera directa.

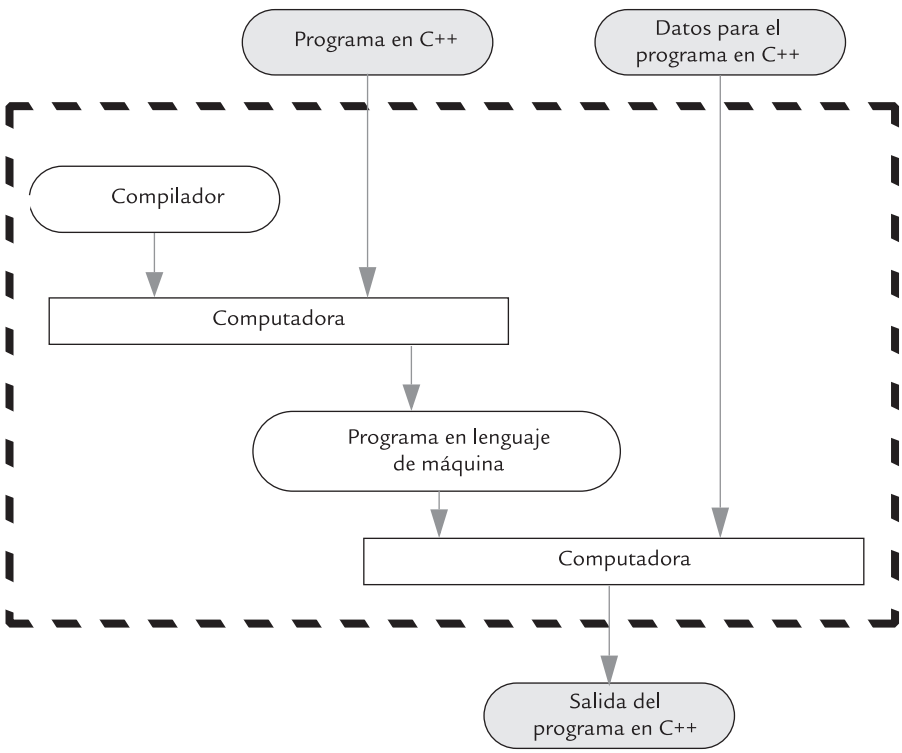
El proceso completo de traducción y ejecución de un programa en C++ es un poco más complejo que lo que muestra el cuadro 1.4. Cualquier programa en C++ que escribamos utilizará algunas operaciones (como rutinas de entrada y salida) que alguien más ha programado antes. Estos elementos previamente programados (como rutinas de entrada y salida) se han compilado con anterioridad, y tienen a sus propios códigos objeto en espera para combinarse con el código objeto de nuestro programa para producir un programa completo en lenguaje de máquina que puede ejecutarse en la computadora. Otro programa, llamado **enlazador**, combina el código objeto de estas partes del programa con el código objeto que el compilador produjo a partir de nuestro programa en C++. En el cuadro 1.5 se muestra la interacción del compilador y el enlazador. En casos muy rutinarios, muchos sistemas realizarán automáticamente este enlazamiento. Por lo tanto, no tendremos que preocuparnos por él.

enlazador

Enlazamiento

El código objeto de nuestro programa en C++ debe combinarse con el código objeto de las rutinas (como las de entrada y salida) que nuestro programa usa. Este proceso de combinación de códigos objeto se conoce como **enlazamiento**, y lo realiza un programa llamado **enlazador**. En programas sencillos el enlazamiento podría ser automático.

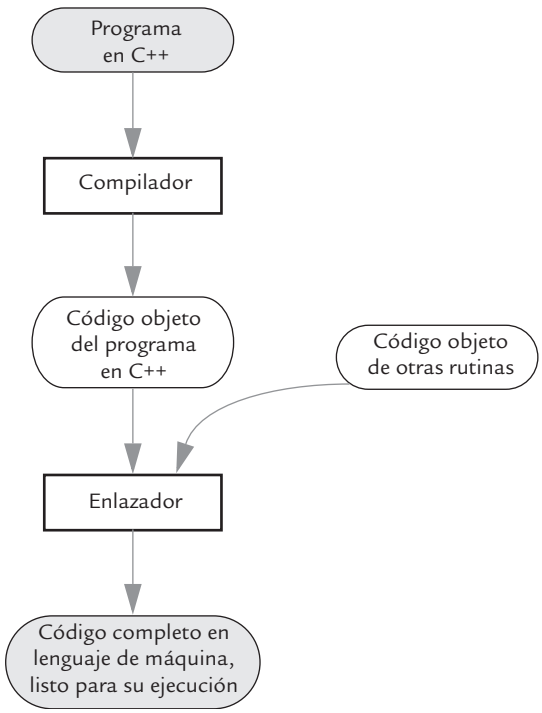
CUADRO 1.4 *Compilación y ejecución de un programa en C++ (diseño básico)*



Ejercicios de AUTOEVALUACIÓN

1. ¿Cuáles son los cinco componentes principales de una computadora?
2. ¿Cuáles serían los datos para un programa que suma dos números?
3. ¿Cuáles serían los datos para un programa que asigna calificaciones a los estudiantes de un grupo?
4. ¿Cuál es la diferencia entre un programa en lenguaje de máquina y uno en lenguaje de alto nivel?
5. ¿Cuál es el papel de un compilador?
6. ¿Qué es un programa fuente?, ¿qué es un programa objeto?

CUADRO 1.5 *Cómo preparar la ejecución de un programa en C++*



7. ¿Qué es un sistema operativo?
8. ¿Cuál es el objetivo de un sistema operativo?
9. Mencione el sistema operativo que ejecutará en su computadora para preparar los programas de este curso.
10. ¿Qué es enlazamiento?
11. Investigue si el compilador que utilizará en este curso enlaza automáticamente.

Nota histórica

Charles Babbage

La primera computadora verdaderamente programable fue diseñada por **Charles Babbage**, un matemático y físico inglés. Babbage comenzó el proyecto poco antes de 1822, y trabajó en él por el resto de su vida. Aunque nunca completó la construcción de su máquina, el diseño fue un hito conceptual en la historia de la computación. Mucho de lo que sabemos sobre Charles Babbage y el diseño de su computadora proviene de los escritos de su colega **Ada Augusta**, Condesa de Lovelace e hija del poeta Byron. Con frecuencia se dice que

Ada Augusta

ella fue la primera programadora de computadoras. Sus comentarios, citados al inicio de la siguiente sección, aún aplican en el proceso de resolución de problemas a través de una computadora. Las computadoras no son mágicas y no tienen, al menos hasta el momento, la habilidad de formular soluciones sofisticadas para todos los problemas que enfrentamos; simplemente hacen lo que el programador les ordena; llevan a cabo las soluciones a los problemas, pero el programador es quien formula el proceso de solución. Nuestra exposición sobre la programación de computadoras inicia con una explicación sobre cómo un programador plantea las soluciones.

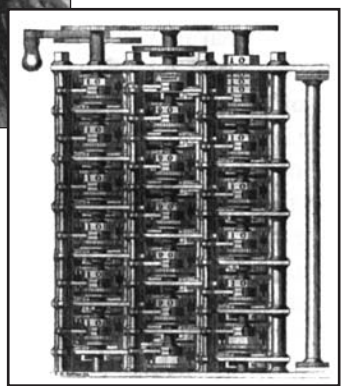
1.2 Solución de problemas y programación

La Máquina Analítica no pretende originar absolutamente nada; sólo puede hacer cualquier cosa que nosotros le ordenemos que realice, si sabemos cómo hacerlo. Puede realizar un análisis sencillo, pero no tiene la capacidad de deducir relaciones analíticas o verdades. Su propósito es ayudarnos a poner disponibles resultados de aquello con lo que ya estamos familiarizados.

Ada Augusta, Condesa de Lovelace (1815-1852)

▲
Ada Augusta,
Condesa de Lovelace
y primera programadora
de computadoras

▲ Charles Babbage



◀ Un modelo de la computadora de Babbage

En esta sección describiremos algunos principios generales que podemos utilizar para diseñar y escribir programas. Estos principios no son exclusivos de C++; son válidos sea cual sea el lenguaje de programación que estemos utilizando.

Algoritmos

Al aprender nuestro primer lenguaje de programación, es fácil hacerse a la idea de que la parte difícil de resolver un problema en una computadora es traducir las ideas en el lenguaje específico que alimentará a la computadora. En definitiva, ése no es el caso. La parte más difícil de la resolución de problemas en computadora es descubrir el método de resolución. Una vez que se tiene el método de resolución, es cuestión de rutina el traducir el método en el lenguaje requerido, sea C++ o algún otro. Por lo tanto, resulta útil ignorar temporalmente el lenguaje de programación y, en su lugar, concentrarse en formular los pasos de la solución y escribirlos en un lenguaje normal (español), como si las instrucciones se le dieran a un ser humano y no a una computadora. A la secuencia de instrucciones expresada de esta forma, se le conoce como *algoritmo*.

algoritmo

A la secuencia de instrucciones precisas que lleva a una solución, se le llama **algoritmo**. Algunas palabras equivalentes son *receta*, *método*, *instrucciones*, *procedimiento* y *rutina*. Las instrucciones pueden expresarse en lenguaje de programación o en lenguaje coloquial. Nuestros algoritmos estarán expresados en español y en lenguaje de programación C++. Un programa de computadora es simplemente un algoritmo expresado en un lenguaje que una computadora puede entender. Entonces, el término *algoritmo* es más general que el término *programa*. Sin embargo, cuando decimos que una secuencia de instrucciones es un algoritmo, normalmente queremos decir que las instrucciones están expresadas en español, ya que si las expresáramos en un lenguaje de programación, utilizaríamos el término específico *programa*. Un ejemplo podría ayudarnos a clarificar el concepto.

algoritmo de ejemplo

El cuadro 1.6 contiene un algoritmo expresado en español. El algoritmo determina el número de veces que un nombre específico se repite en una lista de nombres. Si la lista contiene el nombre de los ganadores de los juegos de fútbol de la última temporada, y el nombre es el de nuestro equipo favorito, entonces el algoritmo determina cuántos partidos ganó su equipo. El algoritmo es corto y sencillo, pero al mismo tiempo es representativo de los algoritmos que manejamos aquí.

Las instrucciones numeradas de la 1 a la 5 de nuestro algoritmo de ejemplo deben llevarse a cabo en el orden en que aparecen. A menos que especifiquemos lo contrario, siempre asumiremos que las instrucciones de un algoritmo se llevan a cabo en el orden en que aparecen (hacia abajo). Sin embargo, los algoritmos más interesantes especifican ciertos cambios en el orden, por lo general la repetición de ciertas instrucciones una y otra vez, como la instrucción 4 de nuestro ejemplo.

origen de la palabra algoritmo

La palabra *algoritmo* tiene una larga historia. Ésta proviene del siglo IX, por el nombre del matemático y astrónomo persa al-Khowarizmi, quien escribió el famoso libro sobre manipulación de números y ecuaciones titulado *Kitab al-jabr w'almugabala*, que traducido al español significa *Reglas de reunión y reducción*. La palabra *álgebra* se derivó de la palabra árabe *al-jabr*, que aparece en el título del libro y que con frecuencia se traduce como *reunión* o *reintegración*. Los significados de las palabras *álgebra* y *algoritmo* solían estar más relacionados de lo que están en la actualidad. De hecho, hasta tiempos modernos, la palabra *algoritmo* sólo se refería a reglas algebraicas para la solución de ecuaciones numéricas. Hoy día, la palabra *algoritmo* puede aplicarse a una amplia variedad de instrucciones para manipular tanto datos simbólicos como numéricos. Las propiedades que califican a un conjunto de instrucciones como un algoritmo, ahora se determinan por la naturaleza de las instrucciones, más que por las cosas manipuladas por las instrucciones.

CUADRO 1.6 *Un algoritmo*

Algoritmo que determina cuántas veces se repite un nombre en una lista de nombres:

1. Obtén la lista de nombres.
2. Obtén el nombre que se está revisando.
3. Establece el contador en cero.
4. Realiza lo siguiente para cada nombre de la lista: Compara el nombre de la lista con el nombre que se está revisando, y si los nombres coinciden, entonces suma uno al contador.
5. Avisa que la respuesta es el número que indica el contador.

Para calificar algo como un algoritmo, un conjunto de instrucciones debe especificar por completo y sin ambigüedades los pasos a seguir, así como el orden en el que deben realizarse. La persona o máquina que siga el algoritmo hace exactamente lo que éste indica, ni más ni menos.

Algoritmo

Un **algoritmo** es una secuencia de instrucciones precisas que llevan a una solución.

Diseño de programas

Diseñar un programa con frecuencia resulta una tarea difícil. No existe un conjunto completo de reglas o un algoritmo que le diga cómo escribir programas. El diseño de programas es un proceso creativo; sin embargo, contamos con un bosquejo del plan a seguir. En el cuadro 1.7 se presenta en forma esquemática dicho bosquejo. Como se indica ahí, todo el proceso de diseño de programas puede dividirse en dos fases: la *fase de resolución de problemas* y la *fase de implementación*. El resultado de la **fase de resolución de problemas** es un algoritmo, expresado en español, para solucionar el problema. Para producir un programa en un lenguaje de programación como C++, el algoritmo se traduce al lenguaje de programación. La producción del programa final, a partir del algoritmo, es la **fase de implementación**.

fase de resolución de problemas

fase de implementación

El primer paso es asegurarse de que la tarea, es decir, lo que queremos que el programa realice, esté especificada de forma completa y precisa. No tome este paso a la ligera. Si no sabe exactamente lo que quiere que arroje su programa, podría sorprenderse con lo que el programa produzca. Debemos asegurarnos de que sabemos qué entradas tendrá el programa exactamente, y qué información se supone que debe estar en la salida, así como la forma que debe tener esa información. Por ejemplo, si el programa es de contabilidad bancaria, debemos conocer no sólo la tasa de interés, sino también si los intereses se com-

ponen anual, mensual, diariamente o de alguna otra forma. Si el programa es para escribir poesías, debemos determinar si los poemas están en versos libres o si deben estar escritos en pentámetro yámbico o en alguna otra métrica.

Muchos programadores principiantes no comprenden la necesidad de escribir un algoritmo antes de escribir un programa en un lenguaje de programación como C++, por lo que intentan reducir el proceso omitiendo por completo la fase de resolución del problema, o reduciéndolo sólo a la parte de definición del problema. Esto parece razonable; ¿por qué no "ir directo al grano" y ahorrar tiempo? La respuesta es que ¡*esto no ahorra tiempo*! La experiencia ha demostrado que el proceso de dos fases producirá un programa que funcione rápida y correctamente. El proceso de dos fases simplifica la fase de diseño del algoritmo, aislándolo de las detalladas reglas de un lenguaje de programación como C++. El resultado es que el proceso de diseño de algoritmos se vuelve mucho menos intrincado y mucho menos propenso a errores. Incluso para un programa no muy grande, puede representar una diferencia de medio día de cuidadoso trabajo y de muchos y frustrantes días en la búsqueda de errores en un programa poco entendible.

La fase de implementación no es un paso trivial; habrá detalles que podrán ser de cuidado, y en ocasiones algunos no serán de importancia, pero esta fase es mucho más sencilla de lo que puede parecer en un principio. Una vez que se familiarice con C++ o con cualquier otro lenguaje de programación, la traducción de un algoritmo en español hacia un lenguaje de programación será una tarea de rutina.

Como muestra el cuadro 1.7, las pruebas se llevan a cabo en ambas fases. Antes de escribir el programa se prueba el algoritmo, y si se descubre que éste es deficiente, entonces se rediseña. Esa prueba de escritorio se realiza repasando mentalmente el algoritmo y ejecutando los pasos nosotros mismos. Con algoritmos extensos será necesario un lápiz y un papel. El programa en C++ se prueba compilándolo y ejecutándolo con algunos datos de muestra. El compilador genera mensajes de error cuando detecta ciertos tipos de errores. Para detectar otros tipos de errores, es preciso verificar que las salidas sean correctas.

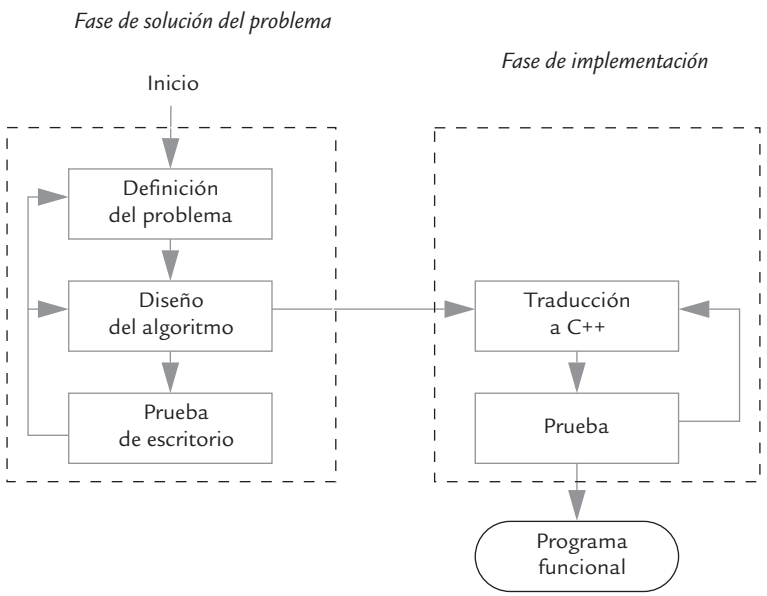
El proceso que se muestra en el cuadro 1.7 es una representación idealizada del proceso de diseño de programas. Es la imagen básica lo que debemos tener en mente, pero en ocasiones la realidad es más compleja. En realidad, los errores y deficiencias se descubren en momentos inesperados, y tendrá que regresar y rehacer algún paso anterior. Por ejemplo, las pruebas del algoritmo podrían revelar que la definición del problema estaba incompleta. En tal caso, hay que retroceder y reformular la definición. En ocasiones, las deficiencias en la definición o en el algoritmo pueden no notarse hasta que se prueba el programa. En ese caso, debe regresar y modificar la definición del problema o el algoritmo y todo lo demás del proceso de diseño.

Programación orientada a objetos

El proceso de diseño de programas que describimos en la sección anterior representa a un programa como un algoritmo (un conjunto de instrucciones) para manipular algunos datos. Ése es un punto de vista correcto, pero no siempre el más productivo. Los programas modernos generalmente se diseñan utilizando un método conocido como *programación orientada a objetos* o **POO**. En la POO, un programa se considera como una colección de objetos en interacción. La metodología es mucho más sencilla de entender cuando el programa es uno de simulación. Por ejemplo, para que un programa simule los rebases en una carretera, los objetos deben representar a los automóviles y a los carriles de la carretera. Cada objeto tiene algoritmos que describen cómo deben comportarse en diferentes situaciones. Programar con estilo de POO consiste en diseñar los objetos y los algoritmos que éstos utilizan. Cuando se programa en un marco

POO

CUADRO 1.7 *Proceso de diseño de programas*



de trabajo de POO, el término *diseño de algoritmos* del cuadro 1.7 se reemplazaría con la frase *diseño de objetos y sus algoritmos*.

Las principales características de la POO son el *encapsulamiento*, la *herencia* y el *polimorfismo*. El encapsulamiento normalmente se describe como una forma de ocultamiento de información o abstracción. Esa descripción es correcta, pero tal vez una caracterización más sencilla sea decir que el encapsulamiento es una forma de simplificar la descripción de los objetos. La herencia tiene que ver con escribir un código de programación reutilizable. El polimorfismo se refiere a la forma en que un solo nombre puede tener diversos significados en el contexto de la herencia. Habiendo establecido lo anterior, debemos admitir que tienen poco significado para los lectores que no han escuchado antes sobre la POO. Sin embargo, más adelante describiremos con detalle todos estos términos. C++ contempla la POO a través de **clases**, un tipo de datos que combina tanto los datos como los algoritmos.

clase

El ciclo de vida del software

ciclo de vida del software

Los diseñadores de grandes sistemas de software, como compiladores y sistemas operativos, dividen el proceso de desarrollo de software en seis fases, que colectivamente se conocen como **ciclo de vida del software**. Las seis fases de este ciclo de vida son:

1. Análisis y especificación de la tarea (definición del problema).
2. Diseño del software (diseño de objetos y algoritmos).
3. Implementación (codificar).
4. Prueba.
5. Mantenimiento y evolución del sistema.
6. Obsolescencia.

Cuando explicamos el diseño de un programa no mencionamos las dos últimas fases anteriores, ya que éstas ocurren una vez que el programa está terminado y en servicio. Sin embargo, siempre debemos mantenerlas en mente. Usted no podrá hacer mejoras o correcciones a su programa a menos que lo diseñe de tal forma que sea fácil leerlo y modificarlo. Diseñar programas para que sean fácilmente modificables es un punto importante, que explicaremos con detalle cuando contemos con más elementos y técnicas de programación. El significado de obsolescencia es obvio, pero no siempre es fácil aceptarlo. Cuando un programa no funciona como debería y no es posible arreglarlo con un esfuerzo razonable, deberá ser descartado y reemplazado por un programa totalmente nuevo.

Ejercicios de AUTOEVALUACIÓN

12. Un algoritmo es casi lo mismo que una receta, pero cierto tipo de pasos que se permitirían en una receta no se permiten en un algoritmo. ¿Cuáles de los pasos en la siguiente receta serían permitidos en un algoritmo?

    Coloque dos cucharaditas de azúcar en un recipiente.

    Agregue 1 huevo.

    Agregue 1 taza de leche.

    Agregue una onza de ron, si no tiene que conducir.

    Agregue extracto de vainilla al gusto.

    Bata hasta que la mezcla sea homogénea.

    Vierta en un hermoso vaso.

    Esparza nuez moscada.

13. ¿Cuál es el primer paso para crear un programa?
14. El proceso de diseño de programas puede dividirse en dos fases principales, ¿cuáles son?
15. Explique por qué la fase de resolución del problema no debe menospreciarse.

1.3 Introducción a C++

El lenguaje es el único instrumento de la ciencia...
Samuel Johnson (1709-1784)

En esta sección le presentaremos el lenguaje de programación C++, que es el lenguaje de programación que utilizaremos en este libro.

Orígenes del lenguaje C++

Lo primero que la gente notó del lenguaje C++ fue su inusual nombre. Podríamos preguntarnos si existe un lenguaje de programación C, o un lenguaje C- o C--, o si existen lenguajes de programación llamados A y B. La respuesta a la mayoría de estas preguntas es no, pero la tendencia general de la pregunta es acertada. Existe un lenguaje de programación llamado B, que no se derivó de un lenguaje llamado A, pero sí de un lenguaje llamado BCPL. El lenguaje C se derivó del lenguaje B, y C++ del lenguaje C. ¿Por qué hay dos signos + en el nombre de C++? Como verá en el siguiente capítulo, ++ es una operación de los lenguajes C y C++, por lo que utilizar ++ produce un buen juego de palabras. Los lenguajes BCPK y B no nos interesan, ya que son versiones anteriores del lenguaje de programación C. Iniciaremos nuestra descripción de C++ con una descripción del lenguaje C.

El lenguaje de programación C fue desarrollado por Dennis Ritchie de AT&T Bell Laboratories en la década de los 70. El primer uso que se le dio fue para escribir y dar mantenimiento al sistema operativo UNIX. (Hasta ese momento, los programas en sistemas UNIX se escribían en lenguaje ensamblador o en B, un lenguaje desarrollado por Ken Thompson, quien es el creador de UNIX.) C es un lenguaje de propósito general que puede utilizarse para escribir cualquier clase de programa, pero su éxito y popularidad están muy relacionados con el sistema operativo UNIX. Si necesitara mantener su sistema UNIX, necesitaría utilizar C. C y UNIX se complementaron tanto que pronto no sólo los programas de sistemas, sino casi todos los programas comerciales que corrían en UNIX se escribieron en lenguaje C. C se volvió tan popular que las versiones del lenguaje se escribieron para otros sistemas operativos populares; su uso no se limita a computadoras con UNIX. Sin embargo, a pesar de su popularidad, C no está libre de defectos.

El lenguaje C es peculiar debido a que es un lenguaje de alto nivel, con muchas características de uno de bajo nivel. C se encuentra entre los extremos de un lenguaje de muy alto nivel y uno de bajo nivel, por lo que tiene tanto sus fortalezas como sus debilidades. Como el lenguaje ensamblador (bajo nivel), los programas en C pueden manipular directamente la memoria de la computadora. Por otra parte, C tiene muchas características de un lenguaje de alto nivel, lo que facilita la lectura y escritura más que en el lenguaje ensamblador. Esto hace que C sea una excelente opción para escribir programas de sistemas, pero para otros programas (y en cierto sentido incluso para programas de sistemas) C no es sencillo de entender, como otros lenguajes; además, no cuenta con muchas verificaciones automáticas, como otros lenguajes de alto nivel.

Para superar estas y otras deficiencias de C, Bjarne Stroustrup de AT&T Bell Laboratories desarrolló C++, a principios de la década de los 80. Stroustrup diseñó C++ para que fuera un mejor C. Casi todo C es un subconjunto de C++, por lo que la mayoría de los programas en C también son programas de C++. (Lo contrario no aplica; muchos programas en C++ no son, en definitiva, programas en C.) A diferencia de C, C++ cuenta con recursos para la *programación orientada a objetos*, que es una técnica de progra-

mación recientemente desarrollada y muy poderosa, como explicamos antes en este capítulo.

Un programa de ejemplo en C++

El cuadro 1.8 contiene un programa sencillo en C++ y la pantalla que podría generarse cuando un *usuario* ejecute e interactúe con este programa. La persona que ejecuta un programa se conoce como el **usuario**. El texto escrito por el usuario aparece en negritas para diferenciarlo del texto escrito por el programa. En la pantalla real, ambos textos se verían iguales. La persona que escribe el programa es llamada el **programador**. No confunda el papel del usuario con el del programador. El usuario y el programador podrían o no ser la misma persona. Por ejemplo, si usted escribe y después ejecuta un programa, usted es ambos, el programador y el usuario. En el caso de programas creados profesionalmente, el programador (o programadores) y el usuario son, en general, diferentes personas.

Usuario

Programador

En el siguiente capítulo explicaremos con detalle todas las características de C++ que necesitará para escribir programas como el del cuadro 1.8, pero para darle una idea de cómo funciona un programa en C++, le daremos una descripción breve de cómo funciona este programa en particular. Si alguno de los detalles no son muy claros, no se preocupe. En esta sección sólo deseamos darle una idea de para qué sirve un programa en C++.

el inicio de un programa

El inicio y final de nuestro programa de ejemplo contienen algunos detalles de los que no tenemos que ocuparnos todavía. El programa inicia con las siguientes líneas:

```
#include <iostream>
using namespace std;
int main()
{
```

Por el momento consideraremos estas líneas como una forma complicada de decir "el programa inicia aquí".

return 0;

El programa finaliza con las siguientes dos líneas:

```
   return 0;
}
```

En un programa sencillo, estas dos líneas simplemente significan "el programa termina aquí".

Las líneas que se encuentran entre estas líneas de inicio y fin son el corazón del programa. Describiremos brevemente estas líneas, y comenzaremos con la siguiente:

```
int numero_de_vainas, chicharos_por_vaina, chicharos_totales;
```

declaración de variables

Esta línea se conoce como **declaración de variables** e indica a la computadora qué `numero_de_vainas`, `chicharos_por_vaina` y `chicharos_totales` se utilizarán como los nombres de tres *variables*. En el siguiente capítulo explicaremos con más detalle las variables, pero es fácil comprender cómo se utilizan en este programa. Aquí, las **variables** se utilizan para nombrar números. La palabra que inicia esta línea, `int`, es una abreviatura de la palabra *integer* (*entero*), y le indica a la computadora que los números nombrados por estas variables serán enteros. Un **entero** es un número sin decimales, como 1, 2, −1, −7, 0, 205, −103, etcétera.

variables

Las líneas restantes son instrucciones que le indican a la computadora que haga algo. Estas instrucciones se conocen simplemente como **instrucciones** o como **instrucciones ejecutables**. En este programa, cada instrucción ocupa exactamente una línea. Esto no nece-

instrucciones

CUADRO 1.8 *Un programa de ejemplo en C++*

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

int main()
{
    int numero_de_vainas, chicharos_por_vaina, chicharos_totales;

    cout << "Oprima entrar despues de introducir un numero. \n" ;
    cout << "Introduzca el numero de vainas:\n";
    cin >> numero_de_vainas;
    cout << "Introduzca el numero de chicharos en una vaina:\n";
    cin >> chicharos_por_vaina;

    chicharos_totales = numero_de_vainas * chicharos_por_vaina;

    cout << "Si tiene ";
    cout << numero_de_vainas;
    cout << " vainas de chicharos \n";
    cout << "y ";
    cout << chicharos_por_vaina;
    cout << " chicharos en cada vaina, entonces\n";
    cout << "tiene ";
    cout << chicharos_totales;
    cout << " chicharos en todas las vainas.\n";

    return 0;
}
```

Diálogo de ejemplo

```
Oprima entrar despues de introducir un numero
Introduzca el numero de vainas
10
Introduzca el numero de chicharos en una vaina
9
Si tiene 10 vainas de chicharos
y 9 chicharos en cada vaina, entonces
tiene 90 chicharos en todas las vainas.
```

sariamente es cierto, pero en programas muy sencillos cada instrucción normalmente ocupa una línea.

cin y cout

La mayoría de las instrucciones comienzan ya sea con la palabra cin o con cout. Éstas son instrucciones de entrada y de salida. La palabra cin se utiliza en la entrada. Las instrucciones que comienzan con cin le indican a la computadora qué hacer cuando la información se introduce desde el teclado. La palabra cout se utiliza en la salida; es decir, para enviar información desde el programa hasta la pantalla final. La letra c está ahí porque el lenguaje es C++. Las flechas, escritas << o >>, le indican la dirección en la que se mueven los datos, y se llaman 'insertar' Y 'extraer', o 'colocar en' y 'obtener desde', respectivamente. Por ejemplo, considere la línea:

```
cout << "Oprima retorno despues de introducir un numero. \n" ;
```

Esta línea puede leerse como, 'coloca "Oprima...numero.\n" en cout' o simplemente 'arroja "Oprima..numero.\n"'. Si piensa en la palabra cout como el nombre de la pantalla (el dispositivo de salida), entonces las flechas le indican a la computadora que envíe la cadena entre comillas hacia la pantalla. Como muestra el diálogo de ejemplo, esto ocasiona que el texto entre comillas aparezca en la pantalla. La \n al final de la cadena entre comillas le dice a la computadora que inicie una nueva línea, después de haber escrito el texto. La siguiente línea del programa también comienza con cout, y hace que la siguiente línea de texto se escriba en la pantalla:

\n

```
introduzca el numero de vainas:
```

La siguiente línea del programa inicia con la palabra cin, por lo que se trata de una instrucción de entrada. Veamos esa línea:

```
cin >> numero_de_vainas;
```

Esta línea puede leerse como 'obtén el numero_de_vainas desde cin' o simplemente 'introduce numero_de_vainas'.

Si piensa en la palabra cin como el equivalente al teclado (el dispositivo de entrada), entonces las flechas dicen que la entrada debe enviarse desde el techado hacia la variable numero_de_vainas. Revise nuevamente el diálogo de ejemplo. La siguiente línea tiene un 10 escrito en negritas. Utilizamos las negritas para indicar algo escrito desde el teclado. Si tecleamos el número 10, éste aparecerá en la pantalla. Si después oprime la tecla Enter, esto ocasiona que el 10 esté disponible para el programa. La instrucción que comienza con cin le indica a la computadora que envíe ese valor de entrada (10) hacia la variable numero_de_vainas. A partir de ese momento, numero_de_vainas tiene el valor 10; cuando más adelante en el programa vemos numero_de_vainas, podemos considerarla como sinónimo de 10.

Considere las dos siguientes líneas del programa:

```
cout << "Introduzca el numero de chicharos en una vaina:\n";
cin >> chicharos_por_vaina;
```

Estas líneas son similares a las dos anteriores. La primera envía un mensaje hacia la pantalla, solicitando un número. Cuando tecleamos un número y oprimimos la tecla Enter, ese número se convierte en el valor de la variable chicharos_por_vaina. En el diálogo de ejemplo, asumimos que el usuario teclea el número 9. Después de teclear el número 9 y oprimir la tecla Enter, el valor de la variable chicharos_por_vaina es 9.

La siguiente línea (que no está en blanco) del programa, la cual mostramos enseguida, realiza todos los cálculos que se llevan a cabo con este sencillo programa:

```
chicharos_totales = numero_de_vainas * chicharos_por_vaina;
```

En C++, el símbolo asterisco, *, se utiliza para multiplicar. Entonces, esta instrucción dice que se multiplique `numero_de_vainas` por `chicharos_por vaina`. En este caso, `10` se multiplica por `9`, para dar como resultado `90`. El signo igual dice que la variable `chicharos_totales` debe igualarse con el resultado `90`. Éste es un uso especial del signo igual; su significado aquí difiere en otros contextos matemáticos. Aquí asigna a la variable del lado izquierdo un valor (probablemente nuevo); en este caso hace de `90` el valor de `chicharos_totales`.

El resto del programa contiene básicamente el mismo tipo de salidas. Considere las siguientes tres líneas que no están en blanco:

```
cout << "Si tiene ";
cout << numero_de_vainas;
cout << " vainas de chicharos\n";
```

Éstas son sólo tres instrucciones más de salida que funcionan básicamente de la misma forma que las instrucciones anteriores que comienzan con `cout`. La única nueva es la segunda de estas tres instrucciones, la cual dice que arroje la variable `numero_de_vainas`. Cuando la salida es una variable, dicha salida es el valor de la variable. Entonces, esta instrucción ocasiona que `10` sea la salida. (Recuerde que en esta ejecución de ejemplo, el usuario estableció el valor de la variable `numero_de_vainas` como `10`.) Por lo tanto, la salida producida por estas tres líneas es:

```
Si tiene 10 vainas de chicharos
```

Observe que la salida aparece en una sola línea. Sólo se inicia una nueva línea cuando se envía como salida la instrucción especial `\n`.

El resto del programa no contiene nada nuevo, y si comprende lo que hemos explicado hasta el momento, debe poder comprender el resto del programa.

RIESGO *Uso erróneo de la diagonal invertida* `\n`

Cuando utilice una `\n` en una instrucción `cout`, asegúrese de que utiliza la **diagonal invertida**, `\`. Si se equivoca y utiliza `/n`, en lugar de `\n`, el compilador no le dará un mensaje de error. Su programa correrá, pero la salida lucirá extraña.

diagonal invertida

TIP DE PROGRAMACIÓN

Sintaxis de entrada y salida

Si consideramos a `cin` como el nombre del teclado o del dispositivo de entrada, y a `cout` como el nombre de la pantalla o del dispositivo de salida, entonces será sencillo recordar la

dirección de las flechas >> y <<. Ellas apuntan en la dirección en que se mueven los datos. Por ejemplo, consideremos la instrucción:

```
cin >> numero_de_vainas;
```

En la instrucción anterior, los datos se mueven desde el teclado hasta la variable `numero_de_vainas`, por lo que la flecha apunta de `cin` hacia la variable.

Por otra parte, considere la instrucción de salida:

```
cout << numero_de_vainas;
```

En esta instrucción, los datos se mueven desde la variable `numero_de_vainas` hasta la pantalla, por lo que la flecha apunta de la variable `numero_de_vainas` hacia `cout`.

Diseño de un programa sencillo en C++

saltos de línea y espacios

En el cuadro 1.9 se muestra la forma general de un programa sencillo en C++. Una vez que el compilador está en lo suyo, los **saltos de línea** y el **espacio** no necesariamente deben ser como aparecen ahí o en nuestros ejemplos. El compilador aceptará cualquier patrón razonable de saltos de línea y sangrías. De hecho, aceptará aun la mayoría de los patrones no razonables de saltos de línea o sangría. Sin embargo, un programa siempre debe diseñarse de tal manera que resulte fácil leerlo. Colocar una llave de apertura, `{`, o una llave de cierre, `}`, en una línea, facilitará encontrar estos patrones. Dar una sangría a cada instrucción y colocarlas en líneas separadas facilita detectar las instrucciones del programa. Más adelante, algunas de nuestras instrucciones serán demasiado largas y no cabrán en una sola línea, y entonces usaremos una variable de este patrón de sangrías y saltos de línea. Le recomendamos seguir el patrón que aparece en los ejemplos de este libro, o el especificado por su maestro, si está tomando una clase.

En el cuadro 1.8, las declaraciones de variables están en la línea que comienza con la palabra `int`. Como veremos en el siguiente capítulo, no necesita colocar todas las declaraciones de variables al principio de su programa, aunque ése es un buen lugar para ubicarlas. A menos que tenga alguna razón para ubicarlas en otro lugar, colóquelas al principio de su programa, como muestra el cuadro 1.9, y el programa de ejemplo del cuadro 1.8. Las **instrucciones** son las órdenes que sigue la computadora. En el cuadro 1.8, las instrucciones son las líneas que inician con `cout` o `cin`, y la línea que comienza con `chicharos_totales` seguida por un signo igual. Con frecuencia, a estas instrucciones se les llama **instrucciones ejecutables**. Nosotros utilizaremos de manera indistinta los términos *instrucción* e *instrucción ejecutable*. Observe que cada una de las instrucciones que hemos visto termina con un signo de punto y coma. En las instrucciones este signo se utiliza casi de la misma manera que en los enunciados en español; éste indica el final de una instrucción.

instrucción

instrucción ejecutable

`#include`

Por el momento puede considerar a las primeras líneas como una forma amena de decir "éste es el comienzo del programa", aunque podemos explicarlas con un poco más de detalle. La primera línea

```
#include <iostream>
```

directiva `include`

se conoce como **directiva** `include`. Ésta le indica al compilador en dónde encontrar la información sobre ciertos elementos que utiliza su programa. En este caso `iostream` es el nombre de una biblioteca que contiene las definiciones de las rutinas que manejan las entradas desde el teclado y el despliegue en la pantalla; `iostream` es un archivo que contiene cierta

CUADRO 1.9 *Esquema de un programa sencillo en C++*

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

int main()
{
    Declaraciones_de_Variables
    Instruccion_1
    Instruccion_2
     ...
    Ultima_Instruccion

    return 0;
}
```

información básica sobre esta biblioteca. El programa enlazador que explicamos anteriormente en este capítulo combina el código objeto de la biblioteca `iostream`, y el código objeto del programa que escribimos. En el caso de la biblioteca `iostream` esto probablemente ocurrirá automáticamente en su sistema. En algún momento utilizaremos otras bibliotecas, y cuando lo hagamos tendremos que nombrar sus directivas al inicio del programa. En el caso de otras bibliotecas tal vez sea necesario hacer algo más que simplemente colocar la directiva `include` en el programa, pero para poder utilizar cualquier biblioteca siempre será necesario colocar al menos una directiva `include`. Las directivas siempre comienzan con el símbolo #. Algunos compiladores requieren que las directivas no tengan espacios alrededor del #, por lo que siempre es más seguro colocarlo al principio de la línea y no incluir ningún espacio entre el # y la palabra `include`.

La siguiente línea muestra la directiva `include` que acabamos de explicar.

```
using namespace std;
```

Esta línea dice que los nombres definidos en `iostream` se interpretarán de "manera estándar" (`std` es una abreviatura de *estándar*). Más adelante diremos algunas cosas más sobre esta línea.

La tercera y cuarta líneas corridas que mostramos a continuación, simplemente dicen que la parte principal del programa inicia aquí:

`int main()`

```
int main()
{
```

El término correcto es *función principal,* no *parte principal*, pero la razón de esta sutil diferencia no la abordaremos hasta el capítulo 3. Las llaves `{` y `}` marcan el principio y el fin de la parte principal del programa. Éstas no necesitan estar por sí mismas en una línea, pero ello permite encontrarlas fácilmente, es por ello que emplearemos esa convención.

`return 0;`

La penúltima línea

```
return 0;
```

dice que "terminar el programa al llegar aquí". Esta línea no necesariamente debe ser lo último del programa, pero en un programa muy sencillo, no tiene sentido colocarla en otra parte. Algunos compiladores permiten omitir esta línea, y asumirán que el programa ha terminado cuando no hay más instrucciones a ejecutar. Sin embargo, otros compiladores insistirán en que se incluya esta línea, por lo que es mejor crear el hábito de incluirla, aunque a nuestro compilador no le haga falta. A esta línea se le conoce como **instrucción return**, y se le considera como ejecutable porque le indica a la computadora que haga algo; específicamente le indica que finalice el programa. En este momento, el número `0` no tiene un significado importante para nosotros, pero lo tendrá; su significado será claro cuando aprenda algunas cosas más sobre C++. Observe que aunque la instrucción `return` indica que se termine el programa, usted tiene que incluir una llave de cierre `}` al final de la parte principal de su programa.

instrucción return

RIESGO *Colocar un espacio antes del nombre de archivo* `include`

Asegúrese de que no hay un espacio entre el símbolo `<` y el nombre de archivo `iostream` (cuadro 1.9), o entre el final del nombre del archivo y el símbolo `>`. La directiva `include` del compilador no es muy inteligente: ¡buscará un nombre de archivo que inicie o termine con un espacio! No encontrará el nombre del archivo y producirá un error que es muy difícil de encontrar. Es recomendable que cometa este error deliberadamente en un programa pequeño, y que compile nuevamente el código. Guarde el mensaje que produzca su compilador, para que la próxima vez que ocurra sepa lo que significa el mensaje de error.

Compilación y ejecución de un programa en C++

En la sección anterior vimos qué sucedería si ejecutara el programa que aparece en el cuadro 1.8. Pero, ¿en dónde está ese programa, y cómo hacer para ejecutarlo?

Escribimos un programa en C++ utilizando un editor de texto, de la misma forma en que escribimos cualquier otro documento, como por ejemplo un convenio, una carta de amor, una lista de compras, etcétera. El programa se mantiene en un archivo, tal y como sucede con cualquier documento que se prepara usando un editor de texto. Existen diferentes tipos de editores de texto, y los detalles de cómo utilizarlos varían de uno a otro, por lo que no podemos decir mucho sobre el suyo en particular. Es recomendable que consulte la documentación de su editor.

La forma de compilar y ejecutar un programa en C++ también dependerá del sistema en particular que estemos empleando, por lo que analizaremos estos puntos de una manera muy general. Es necesario aprender cómo declarar en su sistema los comandos compilar, enlazar y ejecutar un programa en C++. Puede encontrar estos comandos en los manuales del sistema, o podemos preguntarle a alguien que ya utiliza C++ en su sistema. Cuando emitamos el comando compilar nuestro programa, se producirá una traducción a lenguaje de máquina del programa en C++. Esta versión traducida se conoce como el *código objeto* del programa. Este código objeto debe enlazarse (es decir, combinarse) con el código objeto de las rutinas (por ejemplo, de entrada y salida) que ya están escritas. Es muy probable que este enlace se realice de manera automática, por lo que no tenemos

por qué preocuparnos por él. Sin embargo, en algunos sistemas es necesario efectuar una llamada al enlazador. De nuevo, consulte sus manuales o a un experto local. Por último, declararemos el comando para ejecutar el programa; la forma de hacerlo también dependerá del sistema que utilice, por lo que debe verificar los manuales o consultar a un experto.

TIP DE PROGRAMACIÓN

Cómo hacer que su programa corra

Es probable que diferentes compiladores y ambientes requieran ligeras variaciones en algunos detalles, por ejemplo en la manera de crear un archivo con su programa en C++. (Lea el Prefacio para obtener una copia del programa que aparece en el cuadro 1.10, el cual está disponible en Internet, en su versión en inglés.) O bien, cuidadosamente capture usted mismo el programa, compílelo y si obtiene un mensaje de error, revise su captura, corrija cualquier error tipográfico y vuelva a compilar el archivo. Una vez que el programa se compile sin errores, intente ejecutarlo.

Si logra compilar el programa y ejecutarlo correctamente, tiene todo listo; no necesita hacer algo diferente a lo que hemos presentado en los ejemplos. Si este programa no se compila o no se ejecuta normalmente, entonces vuelva a leer. Más adelante le presentaremos algunas pistas para manejar su sistema C++. Una vez que logre ejecutar normalmente este programa sabrá qué pequeños cambios hacer a sus archivos de programa en C++ para hacer que corran en su sistema.

Si su programa aparentemente corre, pero usted no ve la línea de salida

```
Probando 1, 2, 3
```

CUADRO 1.10 *Cómo probar su sistema C++*

Si no puede compilar y ejecutar este programa, revise el tip de programación "Cómo hacer que su programa corra". Éste le sugiere algunas cosas que debe hacer para que su programa se ejecute en su sistema de cómputo en particular.

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

int main()
{
       cout << "Probando 1, 2, 3\n";
       return 0;
}
```

Diálogo de ejemplo

```
Probando 1, 2, 3
```

entonces es muy probable que el programa haya dado esa respuesta, pero desapareció antes de que usted pudiera verla. Intente agregando lo siguiente al final de su programa, justo antes de la línea `return 0`; estas líneas deben detener su programa para permitirle leer la salida.

```
char letra;
cout << "Escriba una letra para finalizar el programa:\n";
cin >> letra;
```

La parte entre llaves entonces debe leerse como:

```
cout << "Probando 1, 2, 3\n";
char letra;
cout << "Escriba una letra para finalizar el programa:\n";
cin >> letra;
return 0;
```

Por el momento, usted no necesita entender estas líneas adicionales, aunque las comprenderá bien al final del capítulo 2.

Si el programa no se compila ni se ejecuta, entonces intente modificando

```
#include <iostream>
```

añadiendo `.h` al final de `iostream`, de la siguiente manera:

```
#include <iostream.h>
```

Si su programa requiere `iostream.h` en lugar de `iostream`, entonces cuenta con un compilador viejo de C++, y debe conseguir uno más reciente.

Si su programa aún no se compila y ejecuta, intente borrando

```
using namespace std;
```

Si su programa aún no se compila y ejecuta, entonces verifique la documentación de la versión de C++, para ver si la "consola" de entrada/salida necesita alguna otra "directiva".

Si todo esto falla consulte a su maestro, si usted está en un curso, y si no lo está o no está utilizando la computadora del curso, revise la documentación de su compilador de C++, o compare con un amigo que tenga un sistema de cómputo similar. El cambio que necesitará sin duda es muy pequeño y, una vez que descubra cuál es será muy sencillo.

Ejercicios de AUTOEVALUACIÓN

16. Si la siguiente instrucción se utilizara en un programa de C++, ¿qué aparecería escrito en pantalla?

```
cout << "C++ es facil de entender.";
```

17. ¿Cuál es el significado de `\n`, en la siguiente instrucción (la cual aparece en el cuadro 1.8)?

```
cout << "Introduzca el numero de chicharos en una vaina:\n";
```

18. ¿Qué significa la siguiente instrucción (aparece en el cuadro 1.8)?

```
cin >> chicharos_por_vaina;
```

19. ¿Qué significa la siguiente instrucción (aparece en el cuadro 1.8)?

```
chicharos_totales = numero_de_vainas * chicharos_por_vaina
```

20. ¿Qué significa la siguiente directiva?

```
#include <iostream>
```

21. ¿Qué está mal, si es que lo está, en las siguientes directivas `#include`?

```
a. #include <iostream >
b. #include < iostream>
c. #include <iostream>
```

1.4 Prueba y depuración

"Y ¿si a trescientos sesenta y cinco le quitas uno, cuánto queda?"
"Trescientos sesenta y cuatro, por supuesto".
Humpty Dumpty miró dudoso. "Preferiría verlo en papel", dijo.

Lewis Carroll, *Alicia a través del espejo*

bug: error
debugging: depuración

En inglés, a un error de programa se le conoce como ***bug*** (insecto), y al proceso de eliminación de errores se le llama ***debugging***, es decir, depuración. Existe una historia pintoresca sobre cómo es que este término comenzó a utilizarse. Ocurrió en los primeros días de las computadoras, cuando el hardware era extremadamente sensible. La Contraalmirante Grace Murray Hopper (1906-1992) fue "la tercera programadora de la primera computadora digital de gran escala en el mundo". (Denise W. Gurer, "Mujer pionera en la ciencia de la computación" CACM 38(1):45-54, enero de 1995.) Mientras Hopper trabajaba en la computadora Harvard Mark I, bajo la dirección del profesor de Harvard H. Aiken, una palomilla ocasionó que la transmisión fallara. Hopper y los otros programadores registraron la muerte de la palomilla en la bitácora, con la nota "Primer caso real de un ***bug*** (insecto) hallado". La bitácora actualmente se encuentra en exhibición en el Museo Naval en Dahlgren, Virginia. En computación, éste fue el primer *bug* documentado. El profesor Aiken entró a las instalaciones durante un descanso y preguntó si se habían encontrado algunos otros *números*, y los programadores respondieron que estaban depurando (*debugging)* la computadora. Para mayor información sobre la Contraalmirante Hopper y otras personas de computación, vea *Portraits in Silicon* de Robert Slater, MIT Press, 1987. En la actualidad, en computación un *bug* es un error. En esta sección describiremos los tres tipos principales de errores de programación y proporcionaremos algunos consejos para corregirlos.

Tipos de errores de programación

error de sintaxis

El compilador atrapará cierto tipo de errores y enviará un mensaje de error cuando encuentre uno. Detectará lo que se conoce como **errores de sintaxis** porque son en gran medida violaciones de sintaxis (es decir, de reglas gramaticales) del lenguaje de programación, como por ejemplo, omitir un punto y coma.

Si el compilador descubre que su programa contiene un error de sintaxis, le indicará en dónde es probable que se encuentre dicho error y qué clase de error es. Si el compilador dice que su programa contiene un error de sintaxis, tenga por seguro que así es. Sin embargo, el compilador puede equivocarse con respecto a la ubicación o a la naturaleza de un error. Éste realiza un mejor trabajo cuando determina la ubicación de un error, entre una línea o dos, que cuando determina la fuente de dicho error. Esto se debe a que el compilador intenta interpretar lo que usted quiso escribir, y con facilidad puede equivocarse. Después de todo, el compilador no puede leer nuestra mente. Los mensajes de error subsiguientes al primero tienen una probabilidad mayor de estar incorrectos, ya sea con respecto a la ubicación o a la naturaleza del error. De nuevo, esto se debe a que el compilador trata de interpretar nuestra intención. Si la primera interpretación del compilador fue incorrecta, esto afectará el análisis de futuros errores, ya que dicho análisis estará basado en una suposición incorrecta.

mensajes de error en comparación con los de advertencia

Si nuestro programa contiene algo que viola directamente las reglas de sintaxis del lenguaje de programación, el compilador enviará un **mensaje de error**. Sin embargo, algunas veces el compilador sólo genera un **mensaje de advertencia**, el cual indica que hemos hecho algo que no es, técnicamente hablando, una violación de las reglas de sintaxis del lenguaje de programación, pero que es lo suficientemente inusual para indicar un probable error. Cuando recibimos un mensaje de advertencia, lo que el compilador le dice es, "¿está seguro de que quiso decir esto?" En esta etapa de su desarrollo, debe tratar cualquier advertencia como un error, hasta que el maestro autorice que la ignore.

errores en tiempo de ejecución

Existe cierto tipo de errores que los sistemas de cómputo sólo pueden detectar cuando se ejecuta el programa. Por ello, se les conoce como **errores en tiempo de ejecución**. La mayoría de los sistemas de cómputo detectará ciertos tipos de errores de este tipo, y desplegarán un mensaje de error adecuado. Muchos de estos errores están relacionados con cálculos numéricos. Por ejemplo, si la computadora intenta dividir un número entre cero, ése normalmente es un error en tiempo de ejecución.

error lógico

El hecho de que el compilador apruebe el programa y éste se ejecute una vez sin mensajes de error en tiempo de ejecución, no garantiza que el programa sea correcto. Recordemos, el compilador sólo nos indica si hemos escrito un programa en C++ gramaticalmente correcto; no nos dice si el programa hace lo que queremos que haga. Los errores en el algoritmo subyacente o en la traducción del algoritmo hacia el lenguaje C++ se conocen como **errores lógicos**. Por ejemplo, si usáramos por equivocación el signo de suma +, en lugar del signo de multiplicación *, en el programa del cuadro 1.8, sería un error lógico. El programa se compilaría y ejecutaría sin contratiempo, pero arrojaría una respuesta incorrecta. Si el compilador aprobó nuestro programa y no hay errores en tiempo de ejecución, pero el programa no se comporta adecuadamente, entonces indudablemente contiene un error lógico. Los errores lógicos son más difíciles de diagnosticar, porque la computadora no le envía mensajes para ayudarle a encontrar el error. No es razonable esperar obtener mensajes de error pues la computadora no tiene forma de saber que lo que escribimos no es lo que realmente queríamos escribir.

RIESGO *Asumir que su programa es correcto*

Con el fin de evaluar un programa nuevo en busca de errores lógicos, debemos ejecutar el programa con conjuntos de datos representativos, y verificar su desempeño con esas entradas. Si el programa pasa esas pruebas podemos confiar más en él, pero esto no representa una garantía absoluta de que el programa es correcto. Aún es probable que no haga lo que queremos que haga, si lo ejecutamos con otros datos. La única forma de justificar la confianza en un programa es programar cuidadosamente para evitar la mayoría de los errores.

Ejercicios de AUTOEVALUACIÓN

22. ¿Cuáles son los tres tipos principales de errores de programación?

23. ¿Qué tipo de errores descubre el compilador?

24. Si en un programa omite un símbolo de puntuación (por ejemplo, un punto y coma), se produce un error. ¿De qué tipo?

25. Omitir la llave final, `}`, en un programa, produce un error. ¿De qué tipo?

26. Suponga que debido a cierta situación el compilador genera una advertencia en el programa. ¿Qué tendría que hacer? Proporcione la respuesta del libro y la suya propia, si se trata de una respuesta diferente a la del texto. Identifique sus respuestas como respuesta de este libro y como respuesta derivada de sus propias reglas.

27. Suponga que escribe un programa que calcula el interés de una cuenta bancaria en un banco que calcula el interés en una base diaria, y suponga que incorrectamente escribe su programa para que calcule el interés en base anual. ¿De qué tipo de error se trata?

Resumen del capítulo

- La colección de programas utilizados por una computadora se conoce como **software** de esa computadora. La parte física de las máquinas, la cual permite la instalación de la computadora, se conoce como **hardware**.

- Los cinco componentes principales de una computadora son: dispositivos de entrada, dispositivos de salida, procesador (CPU), memoria principal y memoria secundaria.

- Una computadora tiene dos tipos de memoria: la memoria principal y la secundaria. La memoria principal se utiliza sólo durante la ejecución de un programa. La memoria secundaria se utiliza para guardar los datos que permanecerán en la computadora antes y/o después de la ejecución de un programa.

- La memoria principal de una computadora se divide en una serie de ubicaciones numeradas llamadas **bytes**. El número asociado con uno de estos bytes se conoce como la **dirección** del byte. Con frecuencia, muchos de estos bytes se agrupan para formar una ubicación de memoria más grande. En ese caso, la dirección del primer byte se utiliza como la dirección de esta ubicación de memoria más grande.

- Un **byte** consiste en ocho dígitos binarios, ya sea cero o uno. Un dígito que sólo puede ser cero o uno se conoce como **bit**.

- Un **compilador** es un programa que traduce un programa escrito en un lenguaje de alto nivel, como C++, en un programa escrito en un lenguaje máquina que la computadora puede entender y ejecutar directamente.

- A la secuencia de instrucciones precisas que lleva a una solución se le conoce como **algoritmo**. Los algoritmos pueden escribirse en español o en un lenguaje de programación como C++. Sin embargo, en la actualidad la palabra algoritmo se utiliza para indicar una secuencia de instrucciones escritas en español (o en algún otro idioma, como inglés o árabe).

- Antes de escribir un programa en C++, debe diseñar el algoritmo (método de solución) que el programa utilizará.
- Los errores de programación pueden clasificarse en tres grupos: errores de sintaxis, errores en tiempo de ejecución y errores lógicos. la computadora normalmente le avisará sobre los errores de las dos primeras categorías y usted debe descubrir los errores lógicos.
- A las órdenes individuales de un programa en C++ se les conoce como **instrucciones**.
- Una variable de un programa en C++ puede utilizarse para nombrar un número. (En el siguiente capítulo explicaremos ampliamente a las variables.)
- Una instrucción de un programa en C++ que comience con `cout <<` es una instrucción de salida, que le indica a la computadora que muestre en la pantalla lo que sea que siga al símbolo `<<`.
- Una instrucción de un programa en C++ que comience con `cin >>` es una instrucción de entrada.

Respuestas a los ejercicios de autoevaluación

1. Los cinco componentes principales de una computadora son: dispositivos de entrada, dispositivos de salida, procesador (CPU), memoria principal y memoria secundaria.
2. Los dos números a sumarse.
3. Las calificaciones de cada alumno en cada examen y en cada materia.
4. Un programa en lenguaje máquina está escrito en una forma que la computadora puede ejecutar directamente. Un programa en lenguaje de alto nivel está escrito en una forma que un humano puede leer y escribir fácilmente. Un programa en un lenguaje de alto nivel se debe traducir a un programa en lenguaje máquina antes de que la computadora pueda ejecutarlo.
5. Un compilador traduce un programa en lenguaje de alto nivel a un programa en lenguaje máquina.
6. El programa en lenguaje de alto nivel que se introduce en un compilador se conoce como programa fuente. El programa traducido a lenguaje máquina que arroja el compilador se conoce como programa objeto.
7. Un sistema operativo es un programa o varios programas en cooperación, pero es mejor considerarlos como el sirviente en jefe del usuario.
8. El propósito de un sistema operativo es asignar los recursos de la computadora para que ésta realice las diferentes tareas asignadas.
9. Entre las posibilidades se encuentran el sistema operativo de Macintosh, Mac OS, Windows 2000, Windows XP, VMS, Solaris, SunOS, UNIX (o tal vez uno de los sistemas operativos como UNIX, por ejemplo Linux), y existen muchos otros.
10. El código objeto de su programa en C++ debe combinarse con el código objeto de las rutinas (por ejemplo, las de entrada y salida) que su programa utiliza. A este proceso de combinación de código objeto se le conoce como enlace. En programas sencillos, este enlace puede llevarse a cabo automáticamente.

11. La respuesta varía, dependiendo del compilador que use. La mayoría de los compiladores UNIX y similares enlazan automáticamente, como lo hacen la mayoría de los compiladores de ambientes integrados de desarrollo para sistemas operativos Windows y Macintosh.

12. Las siguientes instrucciones son demasiado vagas como para utilizarlas en un algoritmo:

```
Agregue extracto de vainilla al gusto.
Bata hasta que la mezcla sea homogénea.
Vierta en un vaso hermoso.
Esparza nuez moscada.
```

    Los términos “al gusto”, “homogénea”, y “hermoso” no son precisos. La instrucción “esparza” es demasiado vaga, ya que no especifica cuánta nuez moscada se debe esparcir. Las otras instrucciones son razonables para usarlas en un algoritmo.

13. El primer paso que debe realizar para crear un programa es asegurarse de que la tarea que debe desarrollar el programa está total y precisamente definida.

14. La fase de solución del problema y la fase de implementación.

15. Porque la experiencia ha demostrado que el proceso de dos fases produce más rápido un programa funcionalmente correcto.

16.
```
C++ es facil de entender.
```

17. Los símbolos `\n` indican a la computadora que inicie una nueva línea en la salida, para que el siguiente elemento de salida aparezca en la siguiente línea.

18. Esta instrucción indica a la computadora que lea el siguiente número que se introduzca desde el teclado, y que lo envíe a la variable llamada `chicharos_por_vaina`.

19. Esta instrucción dice que multiplique los dos números que se encuentran en las variables `numero_de_vainas` y `chicharos_por_vaina`, y que coloque el resultado en la variable llamada `chicharos_totales`.

20. La directiva `#include <iostream>` le indica al compilador que busque el archivo `iostream`. Este archivo contiene declaraciones de `cin`, `cout`, los operadores de inserción (`<<`) y de extracción (`>>`) para E/S (entrada y salida). Esto permite enlazar correctamente el código objeto de la biblioteca iostream con las instrucciones de E/S del programa.

    a) El espacio adicional después del nombre de archivo `iostream` ocasiona un mensaje de *error archivo no encontrado*.

    b) El espacio adicional antes del nombre de archivo `iostream` ocasiona un mensaje de *error archivo no encontrado*.

    c) Ésta es correcta.

21. Los tres principales tipos de errores de programación son: errores de sintaxis, errores en tiempo de ejecución y errores lógicos.

22. El compilador detecta errores de sintaxis. Existen otros errores que técnicamente no son de sintaxis pero que agrupamos con ellos. Más adelante aprenderá sobre esto.

23. Error de sintaxis.

24. Error de sintaxis.

25. El texto establece que debe considerar las advertencias como si fueran reportadas como errores. Es recomendable que consulte a su maestro sobre las reglas locales para manejo de advertencias.

26. Error lógico.

Proyectos de programación

1. Utilizando su editor de texto, introduzca (es decir, escriba) el programa en C++ que aparece en el cuadro 1.8, Asegúrese de escribir la primera línea exactamente como muestra dicho cuadro. En particular, asegúrese de que la primera línea comience al principio del extremo izquierdo, sin espacios antes o después del símbolo #. Compile y ejecute el programa. Si el compilador le envía un mensaje de error, corrija el programa y recompílelo. Haga esto hasta que el compilador no le envíe mensajes de error. Después, ejecute su programa.

2. Modifique el programa en C++ que introdujo en el proyecto de programación 1. Cambie el programa para que la primera línea escriba en la pantalla la palabra `Hola`, y después continúe con el resto del programa del cuadro 1.8. Sólo tiene que agregar una línea al programa para que esto ocurra. Recompile el programa modificado y ejecútelo. Después modifique el programa todavía más. Agregue una línea más que haga que el programa escriba en la pantalla la palabra `Adios`, al final del programa. Asegúrese de agregar los símbolos `\n` en la última instrucción de salida, de tal manera que se lea como:

   ```
   cout << "Adios\n";
   ```

   (Algunos sistemas requieren esa \n, y es posible que el suyo sea uno de ellos.) Recompile el programa y ejecútelo.

3. Modifique aún más el programa que cambió en el proyecto de programación 2. Cambie el signo de multiplicación * por uno de división /. Recompile el programa modificado y ejecútelo. Cuando aparezca la línea "Introduzca el numero de chicharos en una vaina: escriba cero. Observe que el mensaje de error en tiempo de ejecución se debe a una división entre cero.

4. Modifique el programa que introdujo en el proyecto de programación 1. Cambie el signo * por un signo de suma +. Recompile y ejecute el programa modificado. Observe que el programa se compila y ejecuta perfectamente bien, pero la salida es incorrecta. Esto se debe a que esta modificación es un error lógico.

5. Escriba un programa en C++ que lea dos enteros y que después arroje su suma y su producto. Una forma de hacerlo es iniciar con el programa del cuadro 1.8 y después modificarlo para producir el programa para este proyecto. Asegúrese de escribir la primera línea de su programa exactamente de la misma forma en que se escribió la del cuadro 1.8. En particular, asegúrese de que la primera línea comienza en el lado izquierdo de la línea, sin espacios antes o después del símbolo #. Además, asegúrese de agregar los símbolos `\n` en la última instrucción de salida. Por ejemplo, la última instrucción de salida podría ser la siguiente:

   ```
   cout << "Este es el final del programa.\n";
   ```

   (Algunos sistemas requieren esa `\n` final, y es posible que el suyo sea uno de ellos.) Recompile el programa y ejecútelo.

6. El objetivo de este ejercicio es producir un catálogo de errores típicos de sintaxis y de mensajes de error con los que se encontrará un principiante, y también que se familiarice con el ambiente de programación. Este ejercicio debe proporcionarle conocimientos sobre qué tipo de error buscar, dados cierto tipo de mensajes de error.

Es probable que su maestro tenga un programa para que lo use con este ejercicio. Si no es así, utilice uno de los programas de los proyectos de programación anteriores.

Deliberadamente introduzca errores al programa, compile y guarde el error y el mensaje de error; arregle el error, compílelo nuevamente (asegúrese de que tiene el programa corregido), después introduzca otro error. Conserve el catálogo de errores y agréguele errores y sus mensajes, mientras continúe este curso.

La secuencia sugerida de errores a introducir es:

a) Coloque un espacio adicional entre el símbolo < y el nombre de archivo `iostream`.

b) Omita uno de los símbolos < o > en la directiva `include`.

c) Omita el `int` de `int main()`.

d) Omita o escriba mal la palabra `main`.

e) Omita uno de los `()`, después omita ambos.

f) Continúe de esta manera, escribiendo mal los identificadores (`cout`, `cin`, etcétera). Omita uno o ambos símbolos << de la instrucción `cout`; no incluya la llave final `}`.

7. Escriba un programa que imprima C S ! en un gran bloque de letras dentro de un borde de *, seguidos por dos líneas en blanco, y después el mensaje La Ciencia de la Computación es Increíble. La salida debe verse de la siguiente forma:


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

          C C C             S S S S        !!
        C       C         S         S      !!
       C                 S                !!
      C                   S               !!
      C                    S S S S        !!
      C                               S    !!
       C                               S   !!
        C       C         S         S
          C C C             S S S S        OO

**************************************************

La Ciencia de la Computación es Increíble!!!
```

2 Fundamentos de C++

2 Fundamentos de C++

No creas que sabes lo que es una terminal de computadora. Ésta no es una vieja y rara televisión con una máquina de escribir frente a ella; es una interfaz en donde la mente y el cuerpo pueden conectarse con el universo y mover pedacitos de él de aquí para allá.

DOUGLAS ADAMS, *Mostly Harmless*
(el quinto volumen de la trilogía de The Hitchhiker)

Introducción

En este capítulo explicaremos algunos ejemplos de programas en C++, y presentaremos muchos detalles acerca del lenguaje C++ para que pueda escribir programas sencillos en C++.

Prerrequisitos

En el capítulo 1 presentamos una breve descripción de un programa de ejemplo en C++. (Si no lo ha leído, sería útil que lo hiciera antes de leer este capítulo.)

2.1 Variables y asignaciones

Una vez que una persona entiende cómo utilizar las variables en la programación, entiende también la quintaesencia de la programación.

E. W. Dijkstra, *Notas sobre la programación estructurada*

Los programas manipulan datos como números y letras. C++ y la mayoría de los lenguajes de programación utilizan construcciones de programación llamadas *variables*, para nombrar y almacenar datos. Las variables son el corazón de un lenguaje de programación como C++, de modo que es ahí donde iniciaremos nuestra descripción de C++. Utilizaremos el programa del cuadro 2.1 para nuestra demostración, y explicaremos todos los elementos de dicho programa. Aunque la idea general de dicho programa debiera ser clara, algunos de los detalles son nuevos y requerirán alguna explicación.

Variables

Una variable en C++ puede almacenar un número o un dato de otro tipo. Por el momento, nos enfocaremos en las variables que sólo almacenan números. Estas variables son como pequeños pizarrones en donde es posible escribir los números. Así como los números escritos en un pizarrón pueden modificarse, también es posible modificar el número de una variable en C++. A diferencia de un pizarrón que puede no contener un número en absoluto, una variable en C++ debe tener algún valor, aunque sea un número inservible olvidado en la memoria de la computadora por la ejecución de un programa anterior.

valor de una variable

Al número u otro tipo de dato almacenado en una variable se le llama **valor**; esto es, el valor de la variable es el elemento escrito en el pizarrón imaginario. En el programa del cuadro 2.1, `numero_de_barras`, `un_peso`, y `peso_total` son variables. Por ejemplo, cuando el programa se ejecuta con la entrada que muestra el diálogo de ejemplo, a `numero_de_barras` se le asigna un valor igual a 11 mediante la instrucción

```
cin >> numero_de_barras;
```

Más adelante, el valor de la variable `numero_de_barras` cambia a 12, cuando se ejecuta una segunda copia de la misma instrucción. Más adelante explicaremos precisamente cómo sucede esto.

las variables son ubicaciones de memoria

Por supuesto, las variables no son pizarrones. En los lenguajes de programación, las variables se implementan como ubicaciones de memoria. El compilador asigna una ubicación de memoria (del tipo que explicamos en el capítulo 1) a cada nombre de variable del programa. El valor de la variable, en forma de ceros y unos, se almacena en la ubicación asignada a dicha variable. Por ejemplo, a las tres variables del programa que se muestran en el cuadro 2.1 se les puede asignar ubicaciones de memoria con las direcciones 1001, 1003, y 1007. Los números exactos dependerán de su computadora, su compilador y de otros factores. No sabemos, ni nos interesa, cuáles direcciones elegirá el compilador para las variables de nuestro programa. De hecho, podemos pensar que las ubicaciones de memoria fueron etiquetadas con los nombres de las variables.

¿No logra que sus programas se ejecuten?

Si no puede lograr que sus programas se compilen y ejecuten, lea la sección de riesgos del capítulo 1 titulada "Cómo hacer que su programa corra". Esta sección contiene tips para lidiar con diferencias entre los compiladores y los ambientes de C++.

CUADRO 2.1 *Un programa en C++ (parte 1 de 2)*

```
#include <iostream>
using namespace std;
int main( )
{
     int numero_de_barras;
     double un_peso, peso_total;

     cout << "Introduzca el numero de barras de dulce en un
             paquete \n";
     cout << "y el peso en kilos de una barra de dulce.\n";
     cout << "Y presione Intro.\n";
     cin >> numero_de_barras;
     cin >> un_peso;

     peso_total = un_peso * numero_de_barras;

     cout << numero_de_barras << " barras de dulce \n";
     cout << un_peso << " kilos cada una \n";
     cout << "El peso total es de " << peso_total << " kilos.\n";

     cout << "Intente con otra marca.\n";
     cout << "Introduzca el numero de barras de dulce dentro
             de un paquete \n";
     cout << "y el peso en kilos de cada barra de dulce.\n";
     cout << "Y presione Intro.\n";
     cin >> numero_de_barras;
     cin >> un_peso;

     peso_total = un_peso * numero_de_barras;

     cout << numero_de_barras << " barras de dulce\n";
     cout << un_peso << " kilos cada una\n";
     cout << "El peso total es de " << peso_total << " kilos.\n";

     cout << "Quiza una manzana seria mas saludable.\n";

     return 0;}
```

Nombres: Identificadores

Lo primero que notará al examinar los nombres de las variables de nuestros programas de ejemplo es que son más largos que los nombres que generalmente se utilizan para las variables en las clases de matemáticas. Para hacer que su programa sea fácil de comprender, siempre utilice nombres significativos para las variables. Al nombre de una variable

CUADRO 2.1 *Programa en C++ (parte 2 de 2)*

Diálogo de ejemplo

```
Introduzca el numero de barras de dulce en un paquete
y el peso en kilos de una barra de dulce.
Y presione Intro.
11 2.1
11 barras de dulce
2.1 kilos cada una
El peso total es de 23.1 kilos.
Intente con otra marca.
Introduzca el numero de barras de dulce dentro de un paquete
y el peso en kilos de cada barra de dulce.
Y presione Intro.
12 1.8
12 barras de dulce
1.8 kilos cada una
El peso total es de 21.6 kilos.
Quiza una manzana seria mas saludable.
```

identificador

(o a otro elemento que pudiera definir en un programa) se le llama **identificador**. Un identificador debe comenzar con una letra o con el símbolo de guión bajo, y el resto de los caracteres deben ser letras, dígitos o el guión bajo. Por ejemplo, los siguientes son identificadores válidos:

```
x x1 x_1 _abc ABC123z7 suma TASA cuenta dato2 Gran_Bono
```

Todos los nombres mencionados previamente son válidos y serían aceptados por el compilador, pero los primeros cinco son malas opciones para identificadores, ya que no describen el uso del identificador. Ninguno de los siguientes es un identificador válido y todos serían rechazados por el compilador:

```
12 3X %cambio dato-1 miprimer.c PROG.CPP
```

Los primeros tres no son válidos, debido a que no comienzan con una letra o un guión bajo. Los tres restantes no son identificadores, ya que contienen símbolos diferentes a letras, dígitos, y el símbolo de guión bajo.

mayúsculas y minúsculas

C++ es un lenguaje sensible a mayúsculas y minúsculas; es decir, distingue entre letras mayúsculas y minúsculas al deletrear los identificadores. Entonces, los siguientes son tres identificadores distintos y pueden utilizarse para nombrar tres variables distintas:

```
tasa TASA Tasa
```

Sin embargo, no es una buena idea utilizar dos de estas variantes en el mismo programa, ya que podría generar confusión. Aunque C++ no lo requiere, las variables frecuentemente

se escriben solamente con letras minúsculas. Los identificadores predefinidos, como `main`, `cin`, `cout`, y otros, deben escribirse con letras minúsculas. Más adelante veremos usos de identificadores escritos con letras mayúsculas.

Un identificador en C++ puede tener cualquier longitud, aunque algunos compiladores ignorarán los caracteres posteriores a un cierto número específico de caracteres iniciales.

Identificadores

Los **identificadores** se utilizan como nombres de variables y de otros elementos de un programa en C++. Un identificador debe comenzar con una letra o con un guión bajo, y el resto de los caracteres deben ser letras, dígitos o el símbolo de guión bajo.

palabras reservadas

Existe una clase especial de identificadores, llamada **palabras clave** o **palabras reservadas**, que tienen un significado predefinido en C++ y que no podemos utilizar como nombres para variables o para cualquier otra cosa. En este libro, las palabras reservadas aparecen con un tipo de fuente diferente como: *`int`*, *`double`*. (Ahora ya sabe por qué estas palabras se escribieron con un tipo de fuente distinta). En el apéndice 1 encontrará una lista completa de palabras reservadas.

Se preguntará por qué las demás palabras que definimos como parte del lenguaje C++ no se encuentran en la lista de palabras reservadas. ¿Qué hay de palabras como `cin` y `cout`? La respuesta es que podemos redefinir estas palabras, aunque sería confuso hacerlo. Estas palabras predefinidas no son palabras reservadas; sin embargo, están definidas en las bibliotecas requeridas por el estándar del lenguaje C++. Más adelante explicaremos las bibliotecas; por ahora, no debemos preocuparse por ellas. No es necesario decir que el uso de un identificador predefinido para otra cosa que no sea su significado estándar puede ser confuso y peligroso, y por lo tanto se debe evitar. La manera más segura y fácil es tratar a los identificadores predefinidos como si fueran palabras reservadas.

Declaración de variables

declaraciones

Todas las variables de un programa en C++ deben *declararse*. Cuando **declaramos** una variable le decimos al compilador (y al final, a la computadora) qué tipo de datos almacenaremos en dicha variable. Por ejemplo, las siguientes dos declaraciones del programa del cuadro 2.1 declaran las tres variables que se utilizan en ese programa:

```
int numero_de_barras;
double un_peso, peso_total;
```

Cuando existe más de una variable dentro de una declaración, las variables se separan con comas. Además, observe que cada declaración termina con un signo de punto y coma.

La palabra *`int`* en la primera de estas dos declaraciones es una abreviatura de la palabra en inglés *integer*. (Pero en un programa de C++ debe utilizar la forma abreviada *`int`*. No escriba la palabra completa *integer*). Esta línea declara el identificador `numero_de_barras` para que sea una variable de *tipo* *`int`*. Esto significa que el valor de `numero_de_barras` debe ser un número completo, como 1, 2, −1, 0, 37, o −288.

La palabra *`double`* en la segunda de estas líneas declara los identificadores `un_peso` y `peso_total` como variables de tipo *`double`*. Estas variables pueden contener números con una parte fraccionaria, como 1.75 o −0.55. Al tipo de dato que se almacena en una variable se le llama **tipo** y al nombre para el tipo, por ejemplo *`int`* o *`double`*, se le llama **nombre de tipo**.

tipo

Declaración de variables

Todas las variables deben declararse para poder utilizarlas. La sintaxis para la declaración de variables es la siguiente:

Sintaxis *Nombre_Tipo Nombre_Variable_1*, *Nombre_Variable_2*, . . .;

Ejemplo
```
int cuenta, numero_de_dragones, numero_de_duendes;
double distancia;
```

lugar para colocar la declaración de variables

Toda variable de un programa en C++ se debe declarar antes de poder utilizarla. Hay dos lugares naturales para declarar una variable: inmediatamente antes de utilizarla por primera vez o bien al inicio de la parte `main` de nuestro programa, justo después de las líneas

```
int main()
{
```

Elija lo necesario para que nuestro programa sea más claro.

La declaración de variables proporciona la información que el compilador necesita para implementar las variables. Recuerde que el compilador implementa variables como ubicaciones de memoria y que el valor de una variable se almacena en la ubicación de memoria asignada a esa variable. El valor se codifica como una cadena de ceros y unos. Diferentes tipos de variables requieren diferentes tipos de ubicaciones de memoria y diferentes métodos para codificar sus valores como una cadena de ceros y unos. La computadora utiliza un código para codificar enteros como una cadena de ceros y unos, un código diferente para codificar números que tienen una parte fraccionaria y otro código para codificar letras como cadenas de ceros y unos. La declaración de variables le indica al compilador (y por último, a la computadora) qué tamaño de ubicación de memoria utilizar para la variable y cuál código emplear para representar el valor de la variable como una cadena de ceros y unos.

Sintaxis

La **sintaxis** de un lenguaje de programación (o cualquier otra clase de lenguaje) es el conjunto de reglas gramaticales de dicho lenguaje. Por ejemplo, cuando hablamos sobre la sintaxis de una declaración de variable (como en el cuadro titulado "declaración de variables"), nos referimos a las reglas para escribir una declaración de variable bien formada. Si sigue las reglas de sintaxis de C++, entonces el compilador aceptará su programa. Por supuesto, esto solamente garantiza que lo que escribe es correcto, que su programa hará algo; pero no garantiza que su programa hará lo que desea que haga.

Instrucciones de asignación

instrucción de asignación

La manera más directa de modificar el valor de una variable es utilizar una *instrucción de asignación*. Una **instrucción de asignación** es una orden para la computadora que dice, "establece el valor de esta variable a lo que está escrito". La siguiente línea del programa en el cuadro 2.1 es un ejemplo de una instrucción de asignación:

```
peso_total = un_peso * numero_de_barras;
```

Esta instrucción de asignación le indica a la computadora que establezca el valor de la variable `peso_total` igual al número en la variable `un_peso` multiplicada por el número en `numero_de_barras`. (Como explicamos en el capítulo 1, `*` es el signo empleado en C++ para la multiplicación.)

Una instrucción de asignación siempre consta de una variable en la parte izquierda del signo igual y una expresión del lado derecho. Una instrucción de asignación termina con punto y coma. La expresión en la parte derecha del signo igual puede ser una variable, un número o una expresión más complicada compuesta por variables, números y operadores aritméticos como `*` y `+`. Una instrucción de asignación instruye a la computadora para que evalúe (esto es, que calcule el valor de) la expresión en la parte derecha del signo igual y que establezca el valor de la variable en el lado izquierdo del signo igual con el valor de la expresión. Algunos ejemplos harán más clara la manera en que funcionan las instrucciones de asignación.

Podemos utilizar cualquier operador aritmético en lugar del signo de multiplicación. Por ejemplo, la siguiente también es una instrucción de asignación válida:

```
peso_total = un_peso + numero_de_barras;
```

Esta instrucción es igual a las instrucciones de asignación de nuestro programa de ejemplo, excepto porque realiza una suma en lugar de una multiplicación. Esta instrucción cambia el valor de `peso_total` por la suma de los valores de `un_peso` y `numero_de_barras`. Por supuesto, si efectuamos este cambio en el programa del cuadro 2.1, el programa arrojará una salida incorrecta, pero aun así funcionará.

En una instrucción de asignación, la expresión a la derecha del signo igual puede ser sencillamente otra variable. La instrucción

```
peso_total = un_peso;
```

modifica el valor de la variable `peso_total` para que tenga el mismo valor que `un_peso`. Si empleáramos la instrucción anterior en el programa del cuadro 2.1, arrojaría valores incorrectos, demasiado bajos para el peso total de un paquete (suponiendo que existe más de una barra de dulce dentro de un paquete), pero podría tener sentido en algún otro programa.

Como otro ejemplo, la siguiente instrucción de asignación modifica el valor de `numero_de_barras` a `37`:

```
numero_de_barras = 37;
```

constante

A un número, como el `37` de nuestro ejemplo, se le conoce como **constante**, debido que a diferencia de una variable su valor no se puede modificar.

la misma variable en ambos lados del =

Dado que el valor de las variables puede modificarse con el tiempo, y dado que el operador de asignación es un medio para modificar sus valores, existe un elemento de tiempo involucrado en el significado de la instrucción de asignación. Primero se evalúa la expresión en la parte derecha del signo igual. Después de eso, se modifica el valor en el lado izquierdo del signo igual, por el valor que se obtuvo de la expresión. Esto significa que una variable puede aparecer de manera significativa en ambos lados de una asignación. Por ejemplo, considere la instrucción de asignación

```
numero_de_barras = numero_de_barras + 3;
```

En principio, la instrucción de asignación podría parecer extraña. Si la leyéramos como una oración en español, parece decir "numero_de_barras es igual a numero_de_barras más tres". Pareciera decir eso, pero lo que en realidad dice es, "Haz que el *nuevo* valor de numero_de_barras sea igual al valor *anterior* de numero_de_barras más tres". El signo igual en C++ no se utiliza de la misma manera que en español o en matemáticas sencillas.

Instrucciones de asignación

En una instrucción de asignación, primero se evalúa la expresión del lado derecho del signo igual y luego se establece el valor del lado izquierdo con dicho valor.

Sintaxis *Variable* = *Expresión*;

Ejemplos
```
distancia = velocidad * tiempo;
cuenta = cuenta + 2;
```

RIESGO *Variables no inicializadas*

Una variable no tiene un valor significativo hasta que un programa le asigna uno. Por ejemplo, si a la variable numero_minimo no se le ha asignado un valor determinado, ya sea como el lado izquierdo de una asignación o mediante otros medios (como la entrada de un valor mediante la instrucción cin), entonces lo siguiente es un error:

```
numero_deseado = numero_minimo + 10;
```

variables no inicializadas

Esto se debe a que numero_minimo no tiene un valor significativo, de modo que la expresión completa del lado derecho del signo igual no tiene un valor significativo. A una variable como numero_minimo, a la cual no se le ha asignado valor alguno, se le conoce como variable **no inicializada**. De hecho, esta situación es peor que si numero_minimo no tuviera valor alguno. Una variable sin inicializar, como numero_minimo, simplemente contendrá un "valor basura". El valor de una variable no inicializada se determina por el patrón de unos y ceros que dejó en esa ubicación de memoria el último programa que utilizó esa porción de memoria. De modo que si el programa se ejecuta dos veces, una variable no inicializada podría recibir un valor diferente en cada ejecución del programa. Si un programa produce diferentes salidas con los mismos datos de entrada *exactamente* y *sin* que hayamos modificado el programa, debemos sospechar que hay una variable no inicializada.

Una manera de evitar una variable no inicializada es inicializarlas en el momento de declararlas. Esto puede hacerse agregando un signo de igual y el valor de la siguiente forma:

```
int numero_minimo = 3;
```

Esta instrucción declara la variable `numero_minimo` como una variable de tipo entero (*int*) y establece su valor en 3. Podemos utilizar una expresión más complicada que involucre operaciones como suma o multiplicación al inicializar de esta manera una variable dentro de la declaración.

Sin embargo, con frecuencia se utiliza solamente una simple constante. En una declaración que enumera más de una variable podemos inicializar algunas de las variables, todas o ninguna de ellas. Por ejemplo, lo que sigue declara tres variables e inicializa dos de ellas:

```
double tasa = 0.07, tiempo, saldo = 0.0;
```

C++ permite una notación alterna para inicializar variables cuando se declaran. En el siguiente ejemplo explicamos esta notación alterna, el cual es equivalente a la declaración anterior:

```
double tasa(0.07), tiempo, saldo(0.0);
```

El hecho de inicializar una variable al declararla, o en algún punto posterior en el programa, dependerá de las circunstancias. Haga lo que permita que su programa sea más fácil de comprender.

Inicialización de variables dentro de declaraciones

Podemos inicializar una variable (esto es, asignarle un valor) al momento de declararla.

Sintaxis

Nombre_Tipo Nombre_Variable_1 = Expresion_para_Valor_1,
Nombre_Variable_2 = Expresion_para_Valor_2 , . . . ;

Ejemplos

```
int cuenta = 0, limite = 10, factor_sensibilidad = 2;
double distancia = 999.99;
```

Sintaxis alternativa para la inicialización de declaraciones

Nombre_Tipo Nombre_Variable_1 (Expresion_para_Valor_1),
Nombre_Variable_2 (Expresion_para_Valor_2), . . . ;

Ejemplos

```
int cuenta(0), limite(10), factor_sensibilidad(2);
double distancia(999.99);
```

TIP DE PROGRAMACIÓN

Utilice nombres significativos

Los nombres de variables y otros nombres dentro de un programa deben al menos dar una clave del significado o uso de lo que se está nombrando. Es mucho más fácil comprender un programa si las variables tienen nombres significativos. Compare lo siguiente:

```
x = y * z;
```

con esta forma más sugerente:

```
distancia = velocidad * tiempo;
```

Las dos instrucciones cumplen la misma función, pero la segunda es más fácil de comprender.

Ejercicios de AUTOEVALUACIÓN

1. Escriba la declaración de dos variables llamadas `pies` y `pulgadas`. Ambas variables son de tipo *`int`* y deben inicializarse en cero en la declaración. Utilice ambas alternativas de inicialización.

2. Escriba una declaración para dos variables llamadas `cuenta` y `distancia`, `cuenta` es de tipo *`int`* y se inicializa en cero. `distancia` es de tipo *`double`* y se inicializa en `1.5`.

3. Escriba una instrucción en C++ que modifique el valor de la variable `suma` a la suma de los valores en las variables `n1` y `n2`. Todas las variables son de tipo *`int`*.

4. Escriba una instrucción en C++ que incremente el valor de la variable `longitud` en 8.3. La variable `longitud` es de tipo *`double`*.

5. Escriba una instrucción que modifique el valor de la variable `producto` a su antiguo valor multiplicado por el valor de la variable `n`. Las variables son todas de tipo *`int`*.

6. Escriba un programa que contenga instrucciones que muestren el valor de seis variables que estén declaradas, pero no inicializadas. Compile y ejecute el programa. ¿Cuál es la salida? Explique.

7. Escriba nombres significativos para cada una de las siguientes variables:

   a) Una variable para almacenar la velocidad de un automóvil.

   b) Una variable para almacenar la tarifa de pago por hora para un empleado.

   c) Una variable para almacenar el valor más alto en un examen.

2.2 Entrada y salida

Si lo que entra es basura, lo que sale es basura.
Dicho de un programador

Existen muchas formas diferentes en las que un programa en C++ puede efectuar una entrada y salida. Ahora explicaremos lo que llamamos *flujos*. Un **flujo de entrada** es simplemente el flujo que se introduce para el uso de la computadora. La palabra *flujo* sugiere que el programa debe procesar la entrada de la misma manera, independientemente de dónde provenga. La idea de la palabra *flujo* es que el programa ve solamente el flujo de entrada y no la fuente de dicha entrada, como una corriente de agua en una montaña en donde el agua fluye y pasa cerca de nosotros pero cuya fuente desconocemos. En esta sección asumiremos que la entrada proviene del teclado. En el capítulo 5 explicaremos la manera en que un programa puede leer la entrada desde un archivo; como verá ahí, puede utilizar los mismos tipos de instrucciones de entrada, para leer desde un archivo o desde un teclado. De manera similar, un **flujo de salida** es el flujo que genera el programa. En esta sección asumiremos que la salida va hacia la pantalla del monitor; en el capítulo 5 explicaremos la salida que va hacia un archivo.

flujo de entrada

flujo de salida

Salida mediante el uso de `cout`

Los valores de las variables, así como las cadenas de texto, pueden dirigirse a la pantalla mediante el uso de `cout`. Puede darse cualquier combinación de variables y cadenas para la salida. Por ejemplo, consideremos la línea del programa en el cuadro 2.1:

```
cout << numero_de_barras << " barras de dulce\n";
```

Esta instrucción le indica a la computadora que dé salida a dos elementos: el valor de la variable `numero_de_barras` y la cadena entrecomillada "`barras de dulce\n`". Observe que no necesita una copia separada de la palabra `cout` para cada elemento de la salida. Podemos simplemente listar todos los elementos de salida precedidos por los símbolos de flecha <<. La instrucción individual `cout` de arriba es equivalente a las dos siguientes instrucciones `cout`:

```
cout << numero_de_barras;
cout << " barras de dulce\n";
```

En una instrucción `cout` podemos incluir expresiones aritméticas como lo muestra el siguiente ejemplo, en donde `precio` e `impuesto` son variables:

expresión dentro de una instrucción `cout`

```
cout << "El costo total es de $" << (precio + impuesto);
```

Algunos compiladores requieren los paréntesis alrededor de las expresiones aritméticas, como `precio + impuesto`, de modo que es mejor incluirlos.

El símbolo < por sí solo significa "menor que"; por lo tanto, los dos símbolos < deben escribirse sin espacio entre ellos. Con frecuencia, a la notación de flecha << se le conoce como **operador de inserción**. La instrucción `cout` completa termina con un signo de punto y coma.

operador de inserción

Siempre que tenga dos instrucciones `cout` en una línea, puede combinarlas dentro de una sola instrucción `cout`. Por ejemplo, considere las siguientes líneas del cuadro 2.1:

```
cout << numero_de_barras << " barras de dulce\n";
cout << un_peso << " kilos cada una\n";
```

Estas dos instrucciones pueden reescribirse como la siguiente instrucción individual, y el programa hará exactamente lo mismo:

```
cout << numero_de_barras << " barras de dulce\n" << un_peso << " kilos cada una\n";
```

Si desea que las líneas del programa no se salgan de la pantalla, puede colocar una instrucción `cout` larga en dos o más líneas. La mejor forma de escribir la instrucción `cout` anterior es:

```
cout << numero_de_barras << "barras de dulce\n"
     << un_peso << "kilos de cada una\n";
```

No debe dividir una cadena entrecomillada en dos líneas, pero puede comenzar una nueva línea en cualquier parte en donde pueda insertar un espacio. La computadora aceptará cualquier patrón razonable de espacios y cambios de línea, pero el ejemplo previo y los programas de ejemplo son buenos modelos a seguir. Una buena política es utilizar un `cout` para cada grupo de salida que se considere de manera intuitiva como una unidad. Observe que solamente hay un signo de punto y coma para cada `cout`, incluso si la instrucción `cout` se prolonga varias líneas.

Ponga especial atención a las cadenas entrecomilladas que se encuentran en el programa del cuadro 2.1. Observe que las cadenas deben incluirse entre comillas dobles. El símbolo de comillas dobles utilizado es una sola tecla en el teclado; no escriba dos comillas seguidas. Además, observe que el mismo símbolo de comilla doble se utiliza en cada fin de línea de la cadena; no existen símbolos por separado para comillas izquierdas o derechas.

espacios en la salida

Además, observe los espacios dentro de las comillas. La computadora no inserta espacios extras antes o después de los elementos de salida de una instrucción `cout`. Con frecuencia, ésta es la razón por la que las cadenas entrecomilladas de los ejemplos comienzan con un espacio en blanco. Los espacios en blanco evitan que las cadenas y los números queden pegados. Si todo lo que necesita es un espacio y no existe una cadena con comillas en donde desee insertar el espacio, entonces utilice una cadena que únicamente contenga un espacio, como en la siguiente instrucción:

```
cout << primer_numero << "  " << segundo_numero;
```

nuevas líneas en la salida

Como vimos en el capítulo 1, `\n` le indica a la computadora que inicie una nueva línea en la salida. A menos que le indique a la computadora que vaya a la siguiente línea, colocará toda la salida en la misma línea. Dependiendo de la manera como se encuentre configurado su monitor, esto puede producir cualquier cosa, desde una nueva línea arbitraria hasta una línea que salga de la pantalla. Observe que `\n` va dentro de las comillas. En C++, ir a la siguiente línea se considera como un carácter especial (un símbolo especial), y la manera en que deletreamos este carácter especial dentro de la cadena entrecomillada es `\n`, sin espacios intermedios entre los dos símbolos `\n`. Aunque se escribe como dos símbolos, C++ considera a `\n` como un carácter individual que se denomina **carácter de nueva línea**.

carácter de nueva línea

Directivas include y espacios de nombres

Todos nuestros programas han comenzado con el siguiente par de líneas:

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

Estas dos líneas hacen que la biblioteca `iostream` esté disponible. Ésta es una biblioteca que incluye, entre otras cosas, las definiciones `cin` y `cout`. De modo que si nuestro progra-

ma contiene ya sea `cin` o `cout`, debemos incluir las dos líneas anteriores al principio del archivo que contiene el programa.

A la siguiente línea se le conoce como **directiva include**. Ésta "incluye" la biblioteca `iostream` en nuestro programa, de manera que podamos disponer de `cin` y `cout`.

directiva include

```
#include <iostream>
```

Los operadores `cin` y `cout` se definen dentro de un archivo llamado `iostream` y la directiva `include` anterior es equivalente a copiar dicho archivo en nuestro programa. La segunda línea es un poco más complicada de explicar.

C++ divide los nombres dentro de **espacios de nombres**. Un espacio de nombre es una colección de nombres, como los nombres `cin` y `cout`. Una instrucción que especifica un espacio de nombre de la manera en que se muestra a continuación, se llama **directiva** *`using`*.

espacio de nombre

directiva *`using`*

```
using namespace std;
```

Esta directiva *`using`*, en particular dice que nuestro programa utilizará el espacio de nombre `std` ("standard"). Esto significa que los nombres que utilicemos tendrán el significado definido para ellos en el espacio de nombres `std`. En este caso, lo importante es que se dice que cuando los nombres como `cin` y `cout` se definieron en `iostream`, sus definiciones se incluyeron en el espacio de nombres `std`. De manera que para utilizar nombres como `cin` y `cout` necesitamos indicar al compilador que utilizará *`using namespace`* `std;`

Esto es todo lo que necesitamos saber (por ahora) acerca de los espacios de nombres, pero una breve observación eliminará parte del misterio que pudiera rodear el uso de *`namespace`*. La razón por la que existen los espacios de nombres en C++ se debe a que existen demasiadas cosas que nombrar. Por lo tanto, hay ocasiones en que dos o más elementos reciben el mismo nombre; esto es, un nombre individual puede tener dos definiciones diferentes. Para eliminar estas ambigüedades, C++ divide los elementos en colecciones, de tal suerte que no existan dos o más elementos (del mismo espacio de nombres) con el mismo nombre.

Observe que un espacio de nombres no es simplemente una colección de nombres. Es toda una porción de código que especifica el significado de algunos nombres, como definiciones y/o declaraciones. La función de los espacios de nombres es la de dividir todas las especificaciones de nombres de C++ en colecciones (llamadas *`namespaces`*), de tal forma que cada nombre en el espacio de nombres contenga solamente una especificación (una "definición") en el espacio de nombres. Un espacio de nombres divide los nombres, pero implica mucho código en C++ junto con los nombres.

Pero, ¿qué sucede si desea utilizar dos elementos en dos espacios de nombres diferentes, de manera que ambos elementos tengan el mismo nombre? Esto puede hacerse, y no es muy complicado, pero éste es un tema que veremos más adelante. Por ahora, no necesitamos hacerlo.

Algunas versiones de C++ utilizan lo siguiente, una vieja forma de la directiva `include` (sin el uso de *`using namespace`*):

forma alternativa de la directiva `include`

```
#include <iostream.h>
```

Si sus programas no compilan o no se ejecutan con

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

entonces intente con la siguiente línea, en lugar de las dos líneas previas:

```
#include <iostream.h>
```

Si su programa requiere `iostream.h` en lugar de `iostream`, entonces tiene una versión vieja del compilador de C++, y debe obtener un compilador más reciente.

Secuencias de escape

La diagonal invertida, \, que precede a un carácter le indica al compilador que el carácter que sigue a \ no tiene el mismo significado que el carácter que aparece sólo. A dicha secuencia se le llama **secuencia de escape**. La secuencia se digita como dos caracteres sin espacios intermedios entre los símbolos. En C++, existen muchas secuencias de escape.

secuencia de escape

Si queremos colocar una diagonal invertida (\) o una comilla (") en una constante de tipo cadena, es preciso quitarle a la comilla (") la función que tiene de terminar una constante de cadena; y para ello utilizaríamos lo siguiente \". Ahora bien, si lo que queremos es colocar la diagonal invertida (\), debemos introducir \\. El símbolo \\ le indica al compilador que deseamos representar en realidad una diagonal invertida, \, y no una secuencia de escape, y \" significa comillas reales, no el fin de una constante de cadena.

Una \ cualquiera, digamos \z, con una constante de cadena solamente, con cierto compilador regresará una z; y en otro producirá un error. El estándar ANSI contempla el hecho de que las secuencias de escape tienen un comportamiento indefinido. Esto significa que el compilador puede hacer cualquier cosa que su autor considere conveniente. La consecuencia es que el código que utiliza secuencias de escape indefinidas no es portable. No debe utilizar secuencias de escape diferentes a las que ya estén proporcionadas. Aquí listaremos algunas.

nueva línea	\n
tabulador horizontal	\t
alerta	\a
diagonal invertida	\\
comilla doble	\"

Si deseamos insertar un espacio en blanco en la salida, podemos usar el carácter de nueva línea, \n, solo:

```
cout << "\n";
```

Otra manera de evitar a la salida una línea en blanco es utilizar `endl`, lo que esencialmente significa lo mismo que "\n". De modo que también podemos desplegar una línea en blanco de la siguiente manera:

```
cout << endl;
```

Aunque "\n" y `endl` significan lo mismo, se utilizan de manera ligeramente diferente; \n siempre debe ir entrecomillado, y `endl` no.

decidir entre \n **y** `endl`

Una buena regla para decidir cuándo utilizar \n o `endl` es la siguiente: utilice \n al final de una cadena larga (de 2 o más líneas), como en la siguiente instrucción:

```
cout << "La eficiencia del combustible es "
     << mpg << " kilometros por litro\n";
```

Por otro lado, si la cadena es corta (una sola línea) utilice `endl`, como se muestra a continuación:

```
cout << "Usted introdujo " << numero << endl;
```

Cómo comenzar nuevas líneas en la salida

Para comenzar una nueva línea, podemos incluir `\n` dentro de una cadena entrecomillada, como en el siguiente ejemplo:

```
cout << "Usted gano definitivamente\n"
     << "uno de los siguientes premios:\n";
```

Recuerde que `\n` se digita como dos símbolos sin espacios intermedios entre los dos símbolos.

De manera alternativa, puede comenzar una nueva línea mediante el uso de `endl`. Una manera equivalente de escribir la instrucción `cout` anterior es la siguiente:

```
cout << "Usted gano definitivamente" << endl
     << "uno de los siguientes premios:" << endl;
```

TIP DE PROGRAMACIÓN

Finalice cada programa con una `\n` ***o*** `endl`

Una buena idea es colocar una instrucción de nueva línea al final de cada programa. Si el último elemento es una cadena, entonces incluya una \n al final de la cadena, si no, coloque un `endl` como la última acción de su programa. Esto tiene dos propósitos. Algunos compiladores no envían a salida la última línea de su programa, a menos que incluya el símbolo de nueva línea al final. En otros sistemas, su programa podría funcionar bien sin la instrucción de nueva línea al final, pero el siguiente programa que se ejecute mezclará la primera salida en la misma línea que la última del programa anterior. Incluso si ninguno de estos dos problemas se presenta en su sistema, colocar la instrucción de fin de línea al final hace que nuestro programa sea más portable.

Formato de números con un punto decimal

formato para valores *double*

Cuando la computadora envía a la salida un valor de tipo `double`, el formato podría no ser el que deseado. Por ejemplo, la siguiente instrucción `cout` puede producir cualquiera de las siguientes salidas:

```
cout << "El precio es $" << precio << endl;
```

Si `precio` tiene el valor 78.5, la salida podría ser

```
El precio es $78.500000
```

o podría ser

```
El precio es $78.5
```

o se podría desplegar usando la siguiente notación (la cual explicaremos en la sección 2.3):

```
El precio es $7.850000e01
```

Pero es muy poco probable que la salida sea la siguiente, aunque este es el formato que tiene más sentido:

```
El precio es $78.50
```

Para asegurarnos de que la salida tenga la forma que queremos, nuestro programa deberá contener algún tipo de instrucciones que le digan a la computadora cómo desplegar los números.

fórmula mágica

Existe una "fórmula mágica" que podemos insertar en nuestro programa para provocar que los números contengan un punto decimal, como los números de tipo *double* se desplieguen en la notación común con los números exactos de dígitos después del punto decimal que especifiquemos. Si queremos dos dígitos después del punto decimal, podemos utilizar la siguiente fórmula mágica:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

salida de cifras en moneda

Si insertamos las tres instrucciones anteriores en nuestro programa, entonces la instrucción `cout` subsecuente desplegará los valores de tipo `double` en notación ordinaria, con exactamente dos dígitos después del punto decimal. Por ejemplo, supongamos que la siguiente instrucción `cout` aparece en algún lugar después de la fórmula mágica y que el valor de `precio` es de `78.5`:

```
cout << "El precio es $ " << precio << endl;
```

Entonces la salida será la siguiente:

```
El precio es $78.50
```

Podemos utilizar cualquier número entero no negativo en lugar de `2` para especificar un número de dígitos diferente después del punto decimal. Incluso podemos utilizar una variable de tipo *int* en lugar del `2`.

En el capítulo 5 explicaremos con detalle esta fórmula mágica. Por ahora, solamente debemos pensar en esta fórmula mágica como una sola y larga instrucción que le indica a la computadora cómo queremos desplegar los números que contienen un punto decimal.

Si deseamos modificar el número de dígitos después del punto decimal, de modo que valores diferentes del programa se desplieguen con diferente número de dígitos, podemos repetir la fórmula mágica con algún otro número en lugar del `2`. Sin embargo, al repetir la fórmula mágica, sólo necesitamos repetir la última línea. Si la fórmula mágica ya se incluyó en algún punto anterior del programa, entonces la siguiente línea cambiará a `5` el número de dígitos después del punto decimal para todos los valores de tipo *double* subsecuentes que se envíen a la salida:

```
cout.precision(5);
```

Entrada mediante `cin`

Empleamos `cin` para las entradas, más o menos de la misma manera que empleamos `cout` para las salidas. La sintaxis es similar, excepto que `cin` se utiliza en lugar de `cout` y las flechas apuntan en la dirección opuesta. Por ejemplo, en el programa del cuadro 2.1,

Despliegue de valores de tipo *double*

Si insertamos la siguiente "fórmula mágica" en un programa, entonces todos los números de tipo *double* (o de cualquier otro tipo que permita dígitos después del punto decimal) se desplegarán en la notación común que lleva dos dígitos después del punto decimal.

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

Podemos utilizar cualquier otro número entero no negativo en lugar del 2 para especificar un número diferente de dígitos después del punto decimal. Incluso se puede utilizar una variable de tipo *int* en lugar del 2.

las variables numero_de_barras y un_peso se introdujeron mediante las siguientes instrucciones cin (que se muestran junto con las instrucciones cout que le indican al usuario qué hacer):

```
cout << "Introduzca el numero de barras de dulce dentro de un
        paquete\n";
cout << "y el peso en kilos de una barra de dulce.\n";
cout << "Entonces oprima Enter.\n";
cin >> numero_de_barras;
cin >> un_peso;
```

En una sola instrucción cin podemos listar más de una variable. Así pues, las líneas anteriores podrían reescribirse como sigue:

```
cout << "Introduzca el numero de barras de dulce dentro de un
        paquete\n";
cout << "y el peso en kilos de una barra de dulce.\n";
cout << "Entonces oprima Enter.\n";
cin >> numero_de_barras >> un_peso;
```

Si lo prefiere, la instrucción cin anterior puede escribirse en dos líneas de la siguiente forma:

```
cin >> numero_de_barras
    >> un_peso;
```

Observe que, así como en la instrucción cout, solamente existe un signo de punto y coma para cada ocurrencia de cin.

cómo funciona cin

Cuando un programa alcanza una instrucción cin, espera a que la entrada del programa sea introducida desde el teclado. Asigna a la variable el valor de la primera cifra que se introduce desde el teclado, la segunda variable igual al segundo valor digitado, y así sucesivamente. Sin embargo, el programa no lee la entrada sino hasta que el usuario oprime la tecla Intro (Enter). Esto permite al usuario retroceder y corregir los errores al introducir una línea de entrada.

separe los números con espacios en blanco

Los números en las entradas se deben separar por uno o más espacios en blanco, o por un símbolo de cambio de línea. Por ejemplo, si queremos introducir los números **12** y **5**, y en lugar de eso escribimos los números sin espacios intermedios, entonces la computadora pensará que introdujimos el número **125**. Cuando usamos instrucciones cin, la computadora no tomará en cuenta el número de espacios en blanco o cambios de línea hasta que encuentre el siguiente valor de entrada. Así, no importa si los números que introduce están separados por uno o más espacios o incluso un fin de línea.

Instrucciones `cin`

Una instrucción `cin` hace que el contenido de las variables sea igual a los valores introducidos desde el teclado.

Sintaxis

```
cin >> Variable_1 >> Variable_2 >> . . . ;
```

Ejemplo

```
cin >> numero >> tamanio;
cin >> hora_de_partida
    >> puntos_necesarios;
```

Diseño de entrada y salida

E/S

La entrada y la salida, o como con frecuencia se le conoce `E/S`, es la parte del programa que el usuario ve, de tal modo que el usuario no será feliz con un programa a menos que dicho programa tenga una `E/S` muy bien diseñada.

Indicadores de entrada

Cuando la computadora ejecuta una instrucción `cin`, espera la entrada de cierta información desde el teclado. Si no se teclea nada, la computadora simplemente se mantendrá en espera. El programa debe indicarle al usuario cuándo digitar un número (u otro elemento de dato). La computadora no le pedirá al usuario de manera automática que introduzca un dato. Ésta es la razón por la cual los programas de ejemplo contienen instrucciones de salida como la siguiente:

```
cout << "Introduzca el número de barras de dulce de un paquete\n";
cout << "y el peso en kilos de cada barra de dulce.\n";
cout << "Y presione enter.\n";
```

Estas instrucciones de salida le indican al usuario que introduzca los datos de entrada. Nuestros programas siempre deben solicitar los datos de entrada.

El eco de la entrada

Al introducir datos desde una terminal, la entrada aparece en la pantalla conforme se teclean. Sin embargo, el programa siempre debe escribir los valores de entrada en algún momento antes de terminar. Esto se denomina **eco de la entrada**, y permite verificar si la entrada se leyó correctamente. No sólo por el hecho de que la entrada se vea bien en la pantalla cuando se digita significa que la computadora la leyó de manera correcta. Podría existir alguna anomalía o equivocación que no se nota. El eco de la entrada sirve como verificación de la integridad de los datos de entrada.

TIP DE PROGRAMACIÓN

Fin de línea en la E/S

Es posible mantener la entrada y la salida en la misma línea, y en ocasiones puede producir una interfaz más atractiva para el usuario. Si sencillamente omitimos un `\n` o `endl` al final de la última línea de solicitud, entonces la entrada del usuario aparecerá en la misma línea que el indicador. Por ejemplo, supongamos que utiliza las siguientes instrucciones de solicitud y de entrada:

```
cout << "Introduzca el costo por persona: $";
cin >> costo_por_persona;
```

Cuando se ejecuta la instrucción `cout`, en la pantalla aparecerá lo siguiente:

```
Introduzca el costo por persona: $
```

Cuando el usuario escriba la entrada, aparecerá en la misma línea de la siguiente manera:

```
Introduzca el costo por persona: $1.25
```

Ejercicios de AUTOEVALUACIÓN

8. Escriba una instrucción de salida que despliegue en la pantalla el siguiente mensaje:

```
La respuesta a la pregunta de la
vida, el Universo, y todo lo demas es 42.
```

9. Escriba una instrucción de entrada que coloque en la variable `el_numero` (de tipo *`int`*) un número escrito desde el teclado. Anteceda a la instrucción de entrada una instrucción que solicite al usuario que introduzca un número entero.

10. ¿Qué instrucciones debe incluir en el programa para asegurarnos de que, cuando se envíe a la salida un número de tipo *`double`*, éste aparezca en la notación ordinaria con tres dígitos después del punto decimal?

11. Escriba un programa completo en C++ que despliegue en la pantalla la frase `Hola mundo`. El programa no hace nada más.

12. Escriba un programa completo en C++ que lea dos números enteros y que despliegue la suma de ambos. Asegúrese de indicar la entrada de los números, haga eco de la entrada y de que lo envíe a la salida.

13. Escriba una instrucción de salida que produzca un carácter de nueva línea y el de tabulación.

14. Escriba un programa pequeño que declare e inicialice las variables *`double`*, `uno`, `dos`, `tres`, `cuatro` y `cinco` con los valores `1.000`, `1.414`, `1.732`, `2.000` y `2.236`, respectivamente. Luego escriba las instrucciones de salida para generar la siguiente leyenda en la tabla. Utilice la secuencia de escape de tabulador `\t` para alinear las columnas. Si no está familiarizado con el carácter tabulador, primero experimente con él mientras realiza este ejercicio. Un tabulador actúa como el alto mecánico en una máquina de escribir. Un tabulador provoca el comienzo de una nueva columna, por lo general un número de espacios hacia adelante. Muchos editores y procesadores de palabras cuentan con tabuladores ajustables. Nuestra salida no.

    La salida deberá ser:

```
N    Raíz cuadrada
1    1.000
2    1.414
3    1.732
4    2.000
5    2.236
```

2.3 Tipos de datos y expresiones

Ellos no serán felices jamás. Él no es su tipo.

Conversación en una reunión

Los tipos `int` y `double`

Conceptualmente los números `2` y `2.0` son el mismo. Pero C++ los considera como números de diferente tipo. El número entero `2` es de tipo `int`, el número `2.0` es de tipo `double`, debido a que contiene una parte fraccionaria (aunque la fracción es `0`). Una vez más, las matemáticas de la computadora son un poco diferentes a lo que probablemente habíamos aprendido en nuestras clases de matemáticas. Algunas cosas relacionadas con la propiedad práctica de las computadoras hacen que los números de la computadora difieran de las definiciones abstractas de dichos números. Todos los números en C++ se comportan como esperamos que lo hagan. El tipo `int` no contiene sorpresas. Pero los valores de tipo `double` son un poco más problemáticos. Debido a que solamente pueden almacenar un número limitado de dígitos significativos, almacena los números de tipo `double` como valores aproximados. La precisión con la que un valor `double` se almacena, varía de una computadora a otra, pero podemos suponer que se almacenen con 14 o más dígitos de precisión. Para la mayoría de las aplicaciones esto es más que suficiente, pero pueden ocurrir problemas aún en los casos más sencillos. Además, si sabemos que los valores de algunas de las variables siempre serán valores enteros en un rango permitido por nuestra computadora, lo mejor será declarar la variable de tipo `int`.

¿Qué es double?

¿Por qué a los números que contienen una parte fraccionaria se les llama `double`? ¿Existe un tipo llamado "single" que sea de la mitad de su tamaño? No, pero algo de eso es cierto. Muchos lenguajes de programación utilizan tradicionalmente dos tipos de números con una parte fraccionaria. Un tipo utiliza menos espacio y es más impreciso (esto es, no permite muchos dígitos significativos). El segundo tipo usaba el doble (`double`) de espacio de almacenamiento y por lo tanto era mucho más preciso; además admitía números mucho más grandes (aunque los programadores tienden a interesarse más por la precisión que por el tamaño). Al tipo de números que utilizaban más espacio de almacenamiento se les llamaba números de *doble precisión*; aquellos que utilizaban menos espacio de almacenamiento eran llamados de *simple precisión*. Siguiendo esta tradición, en C++ a los tipos que corresponden a este tipo de doble precisión se les llamó `double`. Al tipo que corresponde a los números de simple precisión en C++ se les llamó `float`. C++ contiene además un tercer tipo de números con una parte fraccionaria, al que se le llama `long double`. Estos tipos se describen en la subsección titulada "Otros tipos de números". Sin embargo, en este libro no utilizaremos los tipos `float` y `long double`.

Las constantes numéricas de tipo `double` se escriben diferente de aquellas de tipo `int`. Las constantes de tipo `int` no deben contener punto decimal. Las constantes de tipo `double` pueden escribirse de las dos formas. La forma sencilla para las constantes `double` se parece a la forma común de escribir fracciones decimales. Cuando se escribe de esta forma, una debe contener punto decimal. Sin embargo, algo que las constantes de tipo `double` y las constantes de tipo `int` tienen en común es que ningún número en C++ debe contener comas.

La notación más complicada para las constantes de tipo *double* es con frecuencia llamada **notación científica** o **notación de punto flotante**, y es particularmente útil para escribir números grandes y fracciones realmente pequeñas. Por ejemplo,

notación e

3.67×10^{17}

que es lo mismo que

367000000000000000.0

en C++ se expresa con la constante 3.67e17. El número

5.89×10^{-6}

que es lo mismo que

0.00000589

en C++ se expresa mejor como la constante 5.89e-6. La e significa *exponente* y significa "multiplica por 10 a la potencia que sigue".

La notación e se utiliza debido a que por lo general los teclados no tienen manera de escribir exponentes como superíndices. Podemos considerar que el número después de la *e* nos dice en qué dirección y cuántos dígitos debemos mover el punto decimal. Por ejemplo, para modificar 3.49e4 a un número sin e, mueva el punto decimal cuatro posiciones a la derecha para obtener 34900.0, la cual es otra manera de escribir el mismo número. Si el número después de la e es negativo, movemos el punto decimal el número de posiciones indicadas hacia la izquierda, insertando ceros adicionales si es necesario. Así, 3.49e-2 es lo mismo que 0.0349.

El número antes de e puede contener un punto decimal, aunque no se requiere. Sin embargo, en definitiva el exponente después de e no debe contener punto decimal.

Dado que las computadoras tienen limitaciones en el tamaño de memoria, por lo general los números se almacenan en un número limitado de bytes (esto es, un espacio de almacenamiento limitado). De modo que existe una limitación en el tamaño que puede tener un número, y este límite es diferente para diferentes tipos de números. El número permitido más grande de tipo *double* siempre es mucho más grande que el número más grande permitido de tipo *int*. Cualquier implementación de C++ permitirá valores de tipo *int* tan grandes como 32767 y valores de tipo *double* de hasta 10^{308}.

Otros tipos de números

C++ tiene otro tipo de números además de *int* y *double*. Algunos aparecen en el cuadro 2.2. Los distintos tipos de números permiten números de diferentes tamaños y para una mayor o menor precisión (esto es, más o menos dígitos después del punto decimal). En el cuadro 2.2, los valores dados para el uso de memoria, rango de tamaño y precisión son solamente conjuntos de valores, con la intención de darle una idea general de la diferencia entre los tipos. Los valores varían de un sistema a otro, y puede ser diferente para su sistema.

Aunque algunos de estos tipos de valores numéricos se deletrean como dos palabras, declaramos las variables de estos tipos adicionales igual que las variables de tipo *int* o *double*. Por ejemplo, la siguiente instrucción declara una variable de tipo *long double*:

long double

```
long double numero_grande;
```

CUADRO 2.2 *Algunos tipos de números*

Nombre de tipo	Memoria utilizada	Rango de tamaño	Precisión
`short` (también llamado `short int`)	2 bytes	–32,767 a 32,767	(no aplica)
`int`	4 bytes	–2,147,483,647 a 2,147,483,647	(no aplica)
`long` (también llamado `long int`)	4 bytes	–2,147,483,647 a 2,147,483,647	(no aplica)
`float`	4 bytes	aproximadamente 10^{-38} a 10^{38}	7 dígitos
`double`	8 bytes	aproximadamente 10^{-308} a 10^{308}	15 dígitos
`long double`	10 bytes	aproximadamente 10^{-4932} a 10^{4932}	19 dígitos

Estos son solamente ejemplos de valores para darle una idea de cómo difieren los tipos. Los valores para cada una de estas entradas podrían ser diferentes en su sistema. La precisión hace referencia al número de dígitos significativos, que incluye los dígitos al frente del punto decimal. Los rangos para los tipos `float`, `double` y `long double` son rangos de números positivos. Los números negativos tienen un rango similar, pero con un signo negativo al frente de cada número.

long

Los nombres de tipo `long` y `long int` son nombres diferentes para el mismo tipo. Por lo tanto, el siguiente par de declaraciones es equivalente:

```
long gran_total;
     y su equivalente
long int gran_total;
```

Por supuesto, en cualquier programa solamente debe utilizar una de las dos declaraciones para la variable `gran_total`, pero no importa cuál utilice. Además, recuerde que el nombre de tipo `long` por sí mismo significa lo mismo que `long int`, pero no lo mismo que `long double`.

tipos enteros
tipos de punto flotante

Los tipos para números enteros, como `int` y similares, se llaman **tipos enteros**. Los tipos de números con punto decimal, como el tipo `double` y similares, se llaman **tipos de punto flotante**. Se llaman de *punto flotante* debido a que cuando la computadora almacena un número escrito de la manera usual, como `392.123`, primero convierte el número a algo como la notación `e`; en este caso algo como `3.92123e2`. Cuando la computadora realiza esta conversión, el punto decimal *flota* (esto es, se mueve) a una nueva posición.

Debe estar conciente de que existen otros tipos numéricos en C++. Sin embargo, en este libro solamente utilizaremos los tipos `int`, `double` y ocasionalmente `long`. Para aplicaciones más sencillas, no debe necesitar ningún tipo excepto `int` y `double`. Sin embargo, si escribe un programa que utiliza números enteros muy grandes, entonces podría tener que utilizar el tipo `long`.

long double

El tipo `char`

No queremos darle la impresión de que las computadoras y C++ se utilizan solamente para cálculos numéricos, de modo que ahora introduciremos los tipos no numéricos. Los valores de tipo `char`, el cual es la abreviación de *character*, son símbolos especiales como letras, dígitos o signos de puntuación. En los libros y en las conversaciones, a los valores de este tipo con frecuencia se les llama *caracteres*, pero en un programa en C++ a este tipo siempre se le debe deletrear con la abreviación `char`. Por ejemplo, las variables `simbolo` y `letra` de tipo `char` se declaran de la siguiente manera:

```
char simbolo, letra;
```

Una variable de tipo `char` puede almacenar cualquier carácter individual del teclado. Así, por ejemplo, la variable `simbolo` puede contener una `'A'` o un `'+'` o una `'a'`. Observe que las versiones en mayúscula y en minúscula de una letra se consideran caracteres diferentes.

cadenas y caracteres

Existe un tipo para cadenas de más de un carácter, pero por el momento no explicaremos ese tipo, aunque ya hemos visto, e incluso utilizado valores que son cadenas. Las cadenas entrecomilladas dobles que se despliegan mediante el uso de `cout` son valores de cadena. Por ejemplo, la siguiente cadena, que ocurre en el programa del cuadro 2.1, es una cadena:

```
"Introduzca el numero de barras de dulce en un paquete\n"
```

comillas

Asegúrese de notar que dicha cadena se coloca dentro de comillas dobles, mientras que las constantes de tipo `char` se colocan dentro de comillas simples. Los dos tipos de comillas tienen significados diferentes. En particular, `'A'` y `"A"` son cosas diferentes. `'A'` es un valor de tipo `char` y se puede almacenar en una variable de tipo `char`. `"A"` es una cadena de caracteres. El hecho de que una cadena que contiene solamente un carácter no *hace* de `"A"` un valor de tipo `char`. Observe además que, para ambas cadenas y caracteres, las comillas izquierda y derecha son la misma.

En el programa del cuadro 2.3 explicamos el uso del tipo `char`. Observe que el usuario digita un espacio entre la primera y la segunda inicial. Aunque el programa ignora el espacio en blanco y lee la letra **B** como segundo carácter de entrada. Cuando utiliza `cin` para leer la entrada dentro de una variable de tipo `char`, la computadora ignora los espacios en blanco y saltos de línea hasta que obtiene el primer carácter no blanco y lee el carácter no blanco dentro de la variable. No existe diferencia alguna entre si existen o no espacios en blanco en la entrada. El programa del cuadro 2.3 dará la misma salida si el usuario digita un espacio en blanco entre las iniciales, como lo muestra el diálogo de ejemplo, o si el usuario digita las dos iniciales sin un espacio en blanco, como en:

JB

El tipo `bool`

El último tipo que explicaremos aquí es `bool`. Este tipo fue recientemente agregado al lenguaje C++ por el comité de la ISO/ANSI (International Standards Organization/American National Standards Organization). Las expresiones de tipo `bool` se llaman *Booleanas* en ho-

CUADRO 2.3 *El tipo* `char`

```
#include <iostream>
using namespace std;
int main( )
{
      char simbolo1, simbolo2, simbolo3;

      cout << "Introduzca dos iniciales, sin puntos:\n";
      cin >> simbolo1 >> simbolo2;

      cout << "Las dos iniciales son:\n";
      cout << simbolo1 << simbolo2 << endl;

      cout << "Una vez mas pero con espacios:\n";
      simbolo3 = ' ';
      cout << simbolo1 << simbolo3 << simbolo2 << endl;

      cout << "Eso es todo.";

      return 0;
}
```

Diálogo de ejemplo

```
Introduzca dos iniciales, sin puntos:
J B
Las dos iniciales son:
JB
Una vez mas pero con espacios:
J B
Eso es todo.
```

nor al matemático inglés George Boole (1815-1864) quien formuló las reglas para la lógica matemática.

Las expresiones booleanas evalúan uno de los siguientes valores, `true` (verdadero) o `false` (falso). Las expresiones booleanas se utilizan como marcas y dentro de los ciclos que explicamos en la sección 2.4. Hablaremos más acerca de las expresiones booleanas y el tipo *`bool`* en esta sección.

Compatibilidad de tipos

Como regla general, no puede almacenar un valor de un tipo dentro de una variable de otro tipo. Por ejemplo, la mayoría de los compiladores objetarán la siguiente instrucción:

```
int variable_int;
variable_int = 2.99;
```

El problema es una inconsistencia de tipos. La constante `2.99` es de tipo *`double`* y la variable `variable_int` es de tipo *`int`*. Desafortunadamente, no todos los compiladores reaccionarán de la misma manera con la instrucción de arriba. Algunos arrojarán un mensaje de error, otros arrojarán solamente un mensaje de advertencia, y algunos compiladores ni siquiera objetan. Sin embargo, aunque el compilador nos permita usar la asignación anterior, lo más probable es que asigne a `variable_int` el valor *`int`* `2`, y no el valor `3`. Dado que no podemos confiar en que el compilador acepte esta asignación, no debemos asignar un valor *`double`* a una variable de tipo *`int`*.

El mismo problema surge si utilizamos una variable de tipo *`double`* en lugar de la constante `2.99` La mayoría de los compiladores también objetarán lo siguiente:

```
int variable_int;
double variable_double;
variable_double = 2.00;
variable_int = variable_double;
```

El hecho de que el valor `2.00` "sea par" no hace diferencia alguna. El valor `2.00` es de tipo *`double`*, no de tipo *`int`*. Como veremos en breve, podemos reemplazar a `2.00` con `2` en la instrucción anterior para la variable `variable_double`, pero incluso esto no es suficiente para que dicha asignación sea aceptable. Las variables `variable_int` y `variable_double` son de tipos diferentes, y ésta es la causa del problema.

Incluso si el compilador nos permite mezclar tipos dentro de una instrucción de asignación, lo más recomendable es no hacerlo. Esto hace a nuestro programa menos portable, y puede causar confusión. Por ejemplo, si nuestro compilador nos permite asignar `2.99` a una variable de tipo *`int`*, ésta recibirá el valor `2`, en lugar de `2.99`, lo cual puede provocar confusión porque al parecer el programa dice que el valor es `2.99`.

asignar valores *`int`* a variables *`double`*

Existen algunos casos especiales en donde se permite asignar un valor de un tipo a una variable de otro tipo. Es aceptable asignar un valor de tipo *`int`* a una variable de tipo *`double`*. Por ejemplo, la siguiente instrucción es válida y además tiene un estilo aceptable:

```
double variable_double;
variable_double = 2;
```

La instrucción anterior establece el valor de la variable llamada `variable_double` igual a `2.0`.

Aunque por lo general no es recomendable hacerlo, en una variable de tipo *`char`* podemos almacenar un valor *`int`* como `65` y en una variable de tipo *`int`*, una letra como `'Z'`. Por muchas razones el lenguaje C considera a los caracteres como enteros cortos y, quizá de manera desafortunada, C++ heredó esta característica de C. La razón para permitir esto es que las variables de tipo *`char`* consumen menos memoria que las variables de tipo *`int`* y hacer aritmética con variables de tipo *`char`* puede provocar algún ahorro de memoria. Sin embargo, es más claro el uso del tipo *`int`* cuando estamos manejando enteros y usar el tipo *`char`* sólo cuando estemos manejando caracteres.

mezcla de tipos

La regla general es que no podemos colocar un valor de un tipo en una variable de otro tipo (aunque pareciera que existen más excepciones a la regla que casos que obedecen la regla). Incluso si el compilador no nos obliga a seguir esta regla de manera muy estricta, es una buena regla a seguir. Colocar los datos de un tipo en una variable de otro tipo puede provocar problemas, ya que el valor se debe modificar a un valor del tipo apropiado y el valor podría no ser el que espera.

Los valores de tipo *`bool`* se pueden asignar a variables de un tipo entero (*`short`*, *`int`*, *`long`*) y los enteros se pueden asignar a valores de tipo *`bool`*. Sin embargo, es de poco

estilo hacer esto y no debe utilizar estas características. Con el fin de que nuestra explicación sea completa y para facilitar la lectura de los programas de otras personas, daremos los detalles: Cuando se asigna a una variable de tipo *bool*, cualquier entero distinto de cero se almacenará con valor *true* (verdadero). El cero se almacenará como el valor *false* (falso). Cuando asignamos un valor *bool* a una variable entera, *true* se almacenará como 1 y *false* como 0.

Operadores aritméticos y expresiones

Dentro de un programa en C++, podemos combinar variables y/o números mediante el uso de los operadores aritméticos + para la suma, − para la resta, * para la multiplicación y / para la división. Por ejemplo, la siguiente instrucción de asignación, la cual aparece en el programa del cuadro 2.1, utiliza el operador * para multiplicar los números de dos variables. (Entonces el resultado se coloca en la variable del lado izquierdo del signo igual.)

```
peso_total = un_peso * numero_de_barras;
```

mezcla de tipos

Todos los operadores aritméticos se pueden utilizar con números de tipo int, números de tipo *double* e incluso con un número de cada tipo. Sin embargo, el tipo de valor producido y el valor exacto del resultado depende de los tipos de números que se combinan. Si ambos operandos (esto es, ambos números) son de tipo *int*, entonces el resultado de la combinación de ambos con un operador aritmético es de tipo *int*. Si uno, o ambos de los operandos es de tipo double, entonces el resultado es de tipo double. Por ejemplo, si las variables monto_base e incremento son de tipo *int*, entonces el número producido por la siguiente expresión es de tipo *int*:

```
monto_base + incremento
```

división

Sin embargo, si una de las dos variables es de tipo *double*, entonces el resultado es de tipo *double*. Esto también es verdad si reemplaza el operador + con cualquiera de los operadores –,*, o /.

El tipo del resultado puede ser más significativo de lo que podemos pensar. Por ejemplo, 7.0/2 contiene un operando de tipo *double*, a saber 7.0. Así, el resultado es el tipo *double* 3.5. Sin embargo, 7/2 tiene dos operandos de tipo *int* y por lo tanto produce como resultado el número 3. Incluso si el resultado "es un par", existe una diferencia. Por ejemplo, 6.0/2 tiene un operando de tipo *double*, a saber 6.0. Así, el resultado es el número de tipo *double* 3.0, el cual es solamente una cantidad aproximada. Sin embargo, 6/2 tiene dos operandos de tipo *int*, de modo que el resultado es 3, el cual es de tipo *int* y es una cantidad exacta. El operador de división es el operador que más se ve afectado por el tipo de sus argumentos.

la división entera

Cuando se utiliza con uno o con ambos operadores de tipo *double*, el operador de división, /, se comporta como cabría esperar. Sin embargo, cuando lo utiliza con dos operandos de tipo *int*, el operador de división, /, produce la parte entera del cociente. En otras palabras, la división entera descarta la parte fraccionaria después del punto decimal. Así, 10/3 es 3 (no 3.3333), 5/2 es 2 (no 2.5), y 11/3 es 3 (no 3.6666). Observe que el número no está *redondeado*; la parte después del punto decimal se descarta sin importar qué tan grande sea.

el operador %

El operador % se puede utilizar con los operandos de tipo *int* para recuperar información perdida cuando utilizamos / para hacer la división con números de tipo *int*. Cuando se utiliza con valores de tipo *int*, los dos operadores / y % producen los dos números producidos cuando se ejecute el algoritmo que aprendimos en la primaria. Por ejemplo, 17 di-

vidido por 5 da 3 con un residuo de 2. El operador / produce el número de veces que un número "cabe en" otro. El operador % arroja el residuo. Por ejemplo, las instrucciones

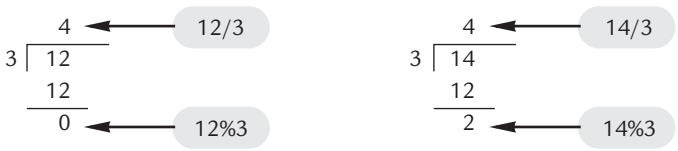
```
cout << "17 dividido por 5 es " << (17/5) << endl;
cout << "con un residuo de " << (17%5) << endl;
```

produce la siguiente salida:

```
17 dividido por 5 es 3
con un residuo de 2
```

El cuadro 2.4 ilustra como / y % trabajan con valores de tipo *int*.

CUADRO 2.4 *División entera*



enteros negativos en una división

Cuando los operadores / y % se emplean con valores negativos de tipo *int*, el resultado puede ser diferente para la diferentes implementaciones de C++. Por eso, recomendamos emplear / y % con *int* cuando sabemos que ambos valores son positivos.

espaciado

Cualquier espacio razonable servirá en las expresiones aritméticas. Podemos insertar espacios antes y después de los operadores y los paréntesis, u omitirlos. Haga lo que sea necesario para producir un resultado que sea fácil de leer.

paréntesis

Mediante la inserción de paréntesis podemos especificar el orden de las operaciones, como explicamos en las siguientes dos expresiones:

```
(x + y) * z
x + (y * z)
```

Para evaluar la primera expresión, la computadora primero suma x y y y después multiplica el resultado por z. Para evaluar la segunda expresión, multiplica y y z y después le suma el resultado a x. Aunque probablemente esté acostumbrado al uso de fórmulas matemáticas que contienen corchetes y varios tipos de paréntesis de fórmulas, en C++ esto no está permitido. C++ solamente permite un tipo de paréntesis dentro de las expresiones aritméticas. Las otras variedades de paréntesis están reservadas para otros propósitos.

reglas de precedencia

Si omitimos los paréntesis, la computadora seguirá las reglas conocidas como **reglas de precedencia**, las cuales determinan el orden en el cual se ejecutan los operadores como + y *. Estas reglas de precedencia son similares a las reglas utilizadas en álgebra y en otras clases de matemáticas. Por ejemplo,

```
x + y * z
```

se evalúa efectuando primero la multiplicación y posteriormente la adición. Excepto en algunos casos comunes, como cadenas o sumas o una simple multiplicación incrustada dentro de una suma, es mejor incluir los paréntesis, incluso si el orden deseado para las operaciones es el que dictan las reglas de precedencia. Los paréntesis hacen que la expresión sea más fácil de leer y menos susceptible a un error del programador. En el apéndice 2 presentamos el conjunto completo de reglas de precedencia para C++.

El cuadro 2.5 muestra ejemplos de algunos tipos comunes de expresiones y cómo se expresan en C++.

CUADRO 2.5 *Expresiones aritméticas*

Fórmula matemática	Expresión en C++
$b^2 - 4ac$	`b*b - 4*a*c`
$x(y + z)$	`x*(y + z)`
$\frac{1}{x^2 + x + 3}$	`1/(x*x + x + 3)`
$\frac{a + b}{c - d}$	`(a + b)/(c d)`

RIESGO *Números enteros en la división*

Cuando utilizamos el operador de división / con dos números enteros, el resultado es un número entero. Esto puede ser un problema si esperamos una fracción. Además es fácil que el problema pase inadvertido, provocando un programa que aparentemente es correcto pero que produce resultados incorrectos sin que nos demos cuenta del problema. Por ejemplo, suponga que es un arquitecto proyectista que destina $5000 por milla a una vía rápida en construcción, y suponga que sabe la longitud de dicha vía en pies. El precio que destina a toda la obra puede ser calculado fácilmente mediante la siguiente instrucción en C++:

```
precio_total = 5000 * (pies/5280.0);
```

Esto funciona debido a que existen 5280 pies en una milla. Si el largo de la vía rápida que planea es de 15,000 de largo, esta fórmula le indicará que el precio total es de:

```
5000 * (15000/5280.0)
```

Su programa en C++ obtiene el valor final de la siguiente manera: `15000/5280.0` da como resultado `2.84`. Entonces su programa multiplica `5000` por `2.84` para producir el valor `14200.00`. Con la ayuda de su programa en C++, sabe que debe destinar $14,200 para el proyecto.

Ahora suponga que la variable `pies` es de tipo *`int`*, y olvida colocar el punto decimal y el cero, de modo que la instrucción de asignación en su programa queda así:

```
precio_total = 5000 * (pies/5280);
```

Aún así se ve bien, pero provocará serios problemas. Si utiliza esta segunda forma de la instrucción de asignación, está dividiendo dos valores de tipo *`int`*, de modo que el resultado de la división `pies/5280` es `15000/5280`, el cual es el valor *`int`* `2` (en lugar del valor `2.84`, que creemos estar obteniendo). Así el valor asignado a `costo_total` es `5000*2`, es decir, `10000.00`. Si olvida el punto decimal, destinará $10,000. Sin embargo, como ya vimos, el valor correcto es $14,200. La omisión de un punto decimal le costará $4,200. Observe que esto será verdadero si el tipo de `precio_total` es *`int`* o es *`double`*; el daño ya está hecho antes de que el valor sea asignado a `precio_total`.

Ejercicios de AUTOEVALUACIÓN

15. Convierta cada una de las siguientes fórmulas matemáticas a una expresión en C++:

    $3x$ $\quad 3x + y$ $\quad \frac{x + y}{7}$ $\quad \frac{3x + y}{z + 2}$

16. ¿Cuál es la salida de las siguientes líneas de programa, incorporadas en un programa que declara todas las variables de tipo *`char`*?

```
a = 'b';
b = 'c';
c = a;
cout << a << b << c << 'c';
```

17. ¿Cuál es la salida de las siguientes líneas de programa (cuando se insertan dentro de un programa correcto que declara `numero` de tipo *`int`*?

```
numero = (1/3) * 3;
cout << "(1/3) * 3 es igual a " << numero;
```

18. Escriba un programa completo en C++ que lea dos números enteros y los coloque en dos variables de tipo *`int`*, y luego despliegue tanto la parte entera como el residuo cuando el primer número se divide entre el segundo. Esto puede hacerse mediante el uso de los operadores / y `%`.

19. Dado el siguiente fragmento que propone convertir de grados Celsius a grados Fahrenheit, responda las siguientes preguntas:

```
double c = 20;
double f;
f = (9/5) * c + 32.0;
```

a) ¿Cuál es el valor asignado a `f`?

b) Explique ¿qué sucede realmente, y qué es lo que el programador probablemente desea?

c) Reescriba el código para que haga lo que el programador quería.

Más instrucciones de asignación

Existe una notación abreviada que combina el operador de asignación (=) y el operador aritmético de tal manera que una variable dada puede modificar su valor al sumar, restar, multiplicar o dividir un valor específico. El formato general es:

Variable Operador = Expresión

la cual es equivalente a

Variable = Variable Operador (Expresión)

Operador es un operador como +, — o *. La *expresión* puede ser otra variable, una constante, o una expresión aritmética más complicada. A continuación unos cuantos ejemplos:

Ejemplo:	Equivalente a:
`cuenta +=2;`	`cuenta = cuenta + 2;`
`total -= descuento;`	`total = total - descuento;`
`bono *= 2;`	`bono = bono *2;`
`tiempo /= factor_prisa`	`tiempo = tiempo/factor_prisa;`
`cambio %= 100;`	`cambio = cambio %100;`
`monto *= cnt1 + cnt2;`	`monto = monto * (cnt1 + cnt2);`

2.4 Control de flujo sencillo

"Si crees que somos figuras de cera", dijo, "entonces debías pagar, ¿no crees? La gente no hace figuras de cera para exhibirlas gratis, ¡no señor!"

"Por otra parte", añadió el rotulado con "DEE", "si crees que estamos vivos, deberías de hablar".

Lewis Carroll, *Alicia a través del espejo*

Los programas que hemos visto hasta este punto solamente constan de una lista de instrucciones que se deben ejecutar en un orden determinado. Sin embargo, para escribir programas

más sofisticados, necesitaremos además alguna manera de variar el orden en el cual se ejecutan las instrucciones. Con frecuencia, al orden en el cual se ejecutan las instrucciones se le conoce como **flujo de control**. En esta sección explicaremos dos formas sencillas de agregar algo de flujo de control de nuestros programas. Explicaremos un mecanismo de bifurcación que le permitirá a nuestros programas elegir entre dos acciones alternativas, seleccionando una o la otra dependiendo de los valores de las variables. También presentaremos un mecanismo de ciclo que permite a nuestro programa repetir una acción varias veces.

flujo de control

Mecanismo de flujo sencillo

Algunas veces es necesario que un programa elija entre dos alternativas, dependiendo de la entrada. Por ejemplo, suponga que desea diseñar un programa que calcule el salario semanal de un empleado al que se le paga por hora. Asuma que la empresa paga el tiempo extra con una tarifa una y media veces mayor que la tarifa normal, y que dichas horas son todas las que excedan las primeras 40 horas trabajadas. Si el empleado trabaja 40 horas o más, el pago será igual a

```
tarifa*40 + 1.5*tarifa*(horas – 40)
```

Sin embargo, si existe la posibilidad de que el empleado haya trabajado menos de 40 horas, esta fórmula le pagará injustamente un monto negativo de tiempo extra. (Para ver esto, solamente sustituya `10` en `horas`, `1` en `tarifa` y haga el cálculo aritmético. Al pobre empleado se le pagará con un cheque en números negativos). La fórmula de pago correcta para un empleado que trabaja menos de 40 horas es simplemente:

```
tarifa*hora
```

Si es posible trabajar tanto más de 40 horas como menos de 40 horas, entonces el programa necesitará elegir entre dos fórmulas. Con el fin de calcular el pago del empleado, la acción del programa deberá ser:

Decidir si (`horas > 40`) es verdadera

Si lo es, realizar las siguientes instrucciones de asignación:

```
pago_bruto = tarifa*40 + 1.5*tarifa(horas – 40);
```

Si no lo es, hacer lo siguiente:

```
pago_bruto = tarifa*horas;
```

Existe una instrucción en C++ que hace exactamente esta clase de acción de marcado. La **instrucción** *`if-else`* elige entre dos acciones alternativas. Por ejemplo, el cálculo de sueldo del que hemos hablado se puede efectuar en C++ mediante la siguiente instrucción:

instrucciones *`if-else`*

```
if (horas > 40)
   pago_bruto = tarifa*40 + 1.5*tarifa*(horas – 40);
else
   pago_bruto = tarifa*horas;
```

En el cuadro 2.6 aparece un programa completo que utiliza esta instrucción.

En el cuadro 2.7 se describen dos formas de la instrucción *`if-else`*. La primera es la forma simple de una instrucción *`if-else`*; la segunda forma la explicaremos en la subsección titulada “Instrucciones compuestas”. En la primera forma, las dos instrucciones pueden ser cualquier instrucción ejecutable. La *expresion_booleana* es una prueba que se puede verificar

CUADRO 2.6 *Instrucción* `if-else`

```
#include <iostream>
using namespace std;
int main( )
{
    int horas;
    double pago_bruto, tarifa;

    cout << "Introduzca la tarifa de pago por hora: $";
    cin >> tarifa;
    cout << "Introduzca el numero de horas trabajadas,\n"
         << "redondeada a un numero entero de horas: ";
    cin >> horas;

    if (horas > 40)
        pago_bruto = tarifa*40 + 1.5*tarifa*(horas - 40);
    else
        pago_bruto = tarifa*horas;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Horas = " << horas << endl;
    cout << "Pago por hora = $" << tarifa << endl;
    cout << "Pago bruto = $" << pago_bruto << endl;

    return 0;
}
```

Diálogo de ejemplo 1

```
Introduzca la tarifa de pago por hora: $20.00
Introduzca el numero de horas trabajadas,
redondeada a un numero entero de horas: 30
Horas = 30
Pago por hora = $20.00
Pago bruto = $600.00
```

Diálogo de ejemplo 2

```
Introduzca la tarifa de pago por hora: $10.00
Introduzca el numero de horas trabajadas,
redondeada a un numero entero de horas: 41
Horas = 41
Pago por hora = $10.00
Pago bruto = $415.00
```

CUADRO 2.7 *Sintaxis para una instrucción* `if-else`

Una instrucción individual para cada alternativa:

```
if (Expresion_Booleana)
   instruccion_Si
else
   instruccion_No
```

Una secuencia de instrucciones para cada alternativa:

```
if (Expresion_Booleana)
{
   instruccion_Si_1
   instruccion_Si_2
     ...
   instruccion_Si_Ultima
}
else
{
   instruccion_No_1
   instruccion_No_2
     ...
   instruccion_No_Ultima
}
```

para ver si es verdadera o falsa, esto es, para ver si se satisface el if o no. Por ejemplo, *expresion_booleana* en la instrucción `if-else` anterior es

```
horas > 40
```

Cuando el programa alcanza la instrucción `if-else`, se ejecuta una de las dos instrucciones incrustadas. Si la *expresion_booleana* es verdadera (esto es, se satisface), entonces se ejecuta la instrucción *instruccion_Si*; si *expresion_booleana* es falsa (esto es, no se satisface), entonces se ejecuta la *instruccion_No*. Observe que la *expresion_booleana* debe estar encerrada entre paréntesis. (Las reglas de sintaxis de la instrucción `if-else` en C++ exigen esto.) Observe además que una instrucción `if-else` contiene dos instrucciones más pequeñas dentro de ella.

expresión booleana

Una **expresion booleana** es cualquier expresión que puede ser verdadera o falsa. Una instrucción `if-else` siempre contiene una *expresion_booleana*. La forma más sencilla de *expresion_booleana* consta de dos expresiones, como números y variables, que se comparan con uno de los operadores de comparación mostrados en el cuadro 2.8. Observe que algunos de los operadores se deletrean con dos símbolos: por ejemplo, ==, !=, <=, >=. Asegúrese de observar que utiliza el doble signo de igual == para el signo de igual, y utiliza los dos símbolos != para el de diferente. Estos operadores de dos símbolos no deben tener espacios inter-

CUADRO 2.8 *Operadores de comparación*

Símbolo matemático	Español	Notación C++	Ejemplo en C++	Equivalente en matemáticas
=	igual a	==	`x + 7 == 2*y`	x + 7 = 2y
π	no igual a	!=	`resp != 'n'`	resp π 'n'
<	menor que	<	`cuenta < m + 3`	cuenta < m + 3
£	menor que o igual a	<=	`tiempo <= límite`	tiempo ≤ límite
>	mayor que	>	`tiempo > límite`	tiempo > límite
≥	mayor que o igual a	>=	`edad >= 21`	edad ≥ 21

medios entre los dos símbolos. La parte del compilador que separa los caracteres dentro de C++ en nombres y símbolos verá !=, por ejemplo, y le indicará al resto del compilador que el programador quiso decir DESIGUALDAD. Cuando se ejecuta una instrucción *if-else*, las dos expresiones que se comparan son evaluadas y comparadas mediante el operador. Si la comparación resulta ser verdadera, entonces se ejecuta la primera instrucción. Si la comparación falla, entonces se ejecuta la segunda instrucción.

&& significa "and" (y)

Podemos combinar dos comparaciones mediante el uso del operador "and" (y), el cual se digita && en C++. Por ejemplo, la siguiente expresión booleana es verdadera (esto es, se satisface) ya que x es mayor que 2 y x es menor que 7.

```
(2 < x) && (x < 7)
```

Cuando se conectan dos comparaciones mediante el uso de &&, la expresión completa es verdadera si las dos comparaciones son verdaderas (esto es, las dos comparaciones se satisfacen); de lo contrario, la expresión completa es falsa.

|| significa "or" (o)

También podemos combinar dos comparaciones mediante el uso del operador "or", el cual se escribe || en C++. Por ejemplo, la siguiente expresión es verdadera si y es menor que 0 o si y es mayor que 12:

```
(y < 0) || (y > 12)
```

Cuando dos comparaciones se conectan mediante el uso de ||, la expresión completa es verdadera si una o ambas comparaciones son verdaderas (esto es, se satisfacen); de lo contrario, la expresión completa es falsa.

paréntesis

Recuerde que al emplear expresiones booleanas dentro de una instrucción *if-else* éstas deben encerrarse entre paréntesis. Por lo tanto una instrucción *if-else* que utiliza el operador && y dos comparaciones se debe encerrar entre paréntesis de la siguiente manera:

```
if ( (temperatura >= 95) && (humedad >= 90) )
     ...
```

El operador "and" &&

Podemos formar expresiones booleanas más elaboradas mediante la combinación de dos pruebas sencillas y el uso del operador &&.

Sintaxis (para una expresión booleana que utiliza &&)

(*comparacion_1*) && (*comparacion_2*)

Ejemplo (dentro de una instrucción *if-else*)

```
if ( (marcador > 0) && (marcador < 10))
  cout << "el marcador se encuentra entre 0 y 10\n";
else
  cout << "el marcador no se encuentra entre 0 y 10.\n";
```

Si el valor de `marcador` es mayor que `0` y además es menor que `10`, entonces se ejecuta la primera instrucción `cout`, de lo contrario, se ejecutará la segunda instrucción.

Los paréntesis interiores alrededor de la comparación no son requeridos, pero hacen que el significado sea más claro, por lo que por lo general los incluiremos.

Podríamos negar cualquier expresión booleana mediante el operador `!`. Si queremos negar una expresión booleana, colocamos la expresión entre paréntesis y colocamos el operador `!` precediendo la expresión. Por ejemplo, `!(x < y)` significa "`x` *no* es menor que `y`". Dado que la expresión booleana de un *`if-else`* debe estar entre paréntesis, es preciso colocar un segundo par de paréntesis alrededor de la expresión de negación si ésta se emplea dentro de una instrucción *`if-else`*. Por ejemplo, una instrucción *`if-else`* podría comenzar de la siguiente manera:

```
if(!(x < y))
   . . .
```

Por lo general, se puede evitar el operador `!`. Por ejemplo, nuestra instrucción *`if-else`* hipotética puede comenzar de la siguiente manera, la cual es equivalente y más fácil de leer:

```
if(x >= y)
    . . .
```

Tendremos oportunidad de utilizar el operador `!` más adelante en el libro, de modo que pospondremos cualquier explicación detallada hasta entonces.

omisión de *else*

Algunas veces necesitaremos que alguna de las alternativas de la instrucción *`if-else`* no realice acción alguna. En C++ esto puede efectuarse omitiendo la parte *`else`*. Esta clase de instrucciones son conocidas como **instrucciones *`if`*** para distinguirlas de las instrucciones *`if-else`*. Por ejemplo, la primera de las dos instrucciones es una instrucción *`if`*:

instrucciones *if*

```
if (ventas >= minimo)
    salario = salario + bono;
cout << "salario = $" << salario;
```

Si el valor de `ventas` es mayor o igual que el valor de `minimo`, se ejecuta la instrucción de asignación y posteriormente se ejecuta la instrucción `cout`. Por otra parte, si el valor de `ventas`

es menor que `minimo`, entonces no se ejecuta la instrucción de asignación incrustada, de modo que la instrucción *`if`* no provoca cambios (esto es, no se suma bono alguno al salario base) y el programa procede directamente con la instrucción `cout`.

El operador "or" ||

Podemos formar una expresión booleana más elaborada mediante la combinación de dos simples pruebas con el uso del operador "or" ||

Sintaxis (para una expresión booleana con el uso ||)

(*Comparacion_1*) || (*Comparacion_2*)

Ejemplo (Dentro de una instrucción *if-else*)

```
if ( (x == 1) || (x == y) )
    cout << "x es 1 o x es igual a y.\n";
else
    cout << "x no es ni 1 ni es igual a y.\n";
```

Si el valor de x es igual a 1 o el valor de x es igual al valor de y (o ambos), entonces se ejecuta la primera instrucción `cout`; de lo contrario, se ejecuta la segunda instrucción `cout`.

RIESGO *Cadenas de desigualdades*

No utilice en su programa una cadena de desigualdades como la siguiente:

```
if (x < z < y) <-------------------------- ¡No haga esto!
  cout << "z se encuentra entre x y y.";
```

Si utiliza este tipo de expresión, probablemente su programa compilará y se ejecutará, pero sin duda producirá una salida incorrecta. Explicaremos por qué sucede esto después de que aprendamos más detalles acerca del lenguaje C++. El mismo problema ocurrirá con una cadena de comparaciones mediante el uso de cualquiera de los operadores de comparación; el problema no está limitado a comparaciones <. La forma correcta de expresar una cadena de desigualdades es utilizar el operador "and" && de la siguiente manera:

```
if ( (x < z) && (z < y) )<------- forma correcta
   cout << "z esta entre x y y.";
```

RIESGO *Uso de = en lugar de ==*

Desafortunadamente, podemos escribir muchas cosas en C++ que están formadas incorrectamente pero resulta que tienen algún significado poco común. Esto significa que si por error escribe algo que espera que produzca un mensaje de error, podría darse cuenta de que el programa compila y se ejecuta sin mensaje de error alguno, pero produce una salida incorrecta. Pero tal vez no nos demos cuenta de que escribimos algo de manera incorrecta, esto puede provocar serios pro-

blemas. En el momento en el que nos damos cuenta de que algo anda mal, el error podría ser muy difícil de encontrar. Un error común es utilizar el símbolo = cuando en realidad debemos utilizar ==. Por ejemplo, consideremos una instrucción *if-else* que comienza de la siguiente manera:

```
if (x = 12)
    Haz_algo
else
    Haz_otra_cosa
```

Supongamos que deseamos probar si el valor de x es igual a 12 de modo que en realidad quisimos usar == en lugar de =. Podríamos pensar que el compilador puede detectar este error. La expresión

```
x = 12
```

no es algo que se pueda satisfacer o no. Es una instrucción de asignación, así que con seguridad el compilador generará un mensaje de error. Desafortunadamente, éste no es el caso. En C++ la expresión `x = 12` es una expresión que devuelve (o que tiene) un valor, como `x + 12` o `2 + 3`. El valor de una expresión de asignación es el valor que se transfiere a la variable de la izquierda. Por ejemplo, el valor de `x=12` es `12`. Como vimos al hablar acerca de la compatibilidad de valores booleanos, los valores *int* se pueden convertir a *true* (verdadero) y *false* (falso). Ya que `12` no es igual a cero, se convierte a *true*. Si en una instrucción *if* utilizamos `x = 12` como la expresión booleana siempre se ejecutará la "cláusula si".

El error es muy difícil de encontrar, debido a que ¡*no parece error*! El compilador puede encontrar el error sin instrucciones especiales si colocamos el `12` a la izquierda de la comparación, como en:

```
if (12 == x)
  Haz_algo;
else
  Haz_otra_cosa;
```

Entonces, el compilador producirá un mensaje de error si utilizamos = en lugar de ==.

Recuerde que omitir un = de un == es un error común que muchos compiladores no detectan, es mucho más difícil de ver, y ciertamente no es lo que se desea. En C++, muchas instrucciones ejecutables se pueden utilizar como cualquier tipo de expresión, incluso como expresiones booleanas dentro de una instrucción *if-else*. Si colocamos una instrucción de asignación en donde debiera ir una expresión booleana, la instrucción de asignación se interpretará como una expresión booleana. Por supuesto el resultado de la "prueba" indudablemente no será lo que espera de una expresión booleana. La instrucción *if-else* anterior se ve bien con un rápido vistazo y compilará y se ejecutará. Pero, con toda probabilidad, producirá resultados inesperados durante la ejecución.

Instrucciones compuestas

if-else **con múltiples instrucciones**

Con frecuencia querrá que las bifurcaciones de una instrucción *if-else* ejecuten más de una instrucción cada una. Para realizar esto, encerramos las instrucciones para cada bifurcación en un par de llaves, { y }, como indicamos en la segunda plantilla de sintaxis en el cuadro 2.7. Esto se ilustra en el cuadro 2.9. A la lista de instrucciones encerradas en un par de llaves se le conoce como **instrucción compuesta**. C++ trata a una instrucción compuesta como una instrucción individual y se puede utilizar en cualquier parte en donde se pue-

instrucción compuesta

CUADRO 2.9 *Instrucciones compuestas con el uso de* `if-else`

```
if (mi_marcador > su_marcador)
{
    cout << "¡Yo gane!\n";
    apuesta = apuesta + 100;
}
else
{
    cout << "Desearia que estos fueran marcadores de golf.\n";
    apuesta = 0;
}
```

de utilizar una instrucción individual. (Así, la segunda plantilla de sintaxis en el cuadro 2.7 es en realidad solamente un caso especial del primero.) El cuadro 2.9 contiene dos instrucciones compuestas, incrustadas dentro de una instrucción `if-else`. Las instrucciones compuestas están en color.

Las reglas de sintaxis `if-else` requieren que la instrucción *Si* y la instrucción *No* se encuentren dentro de una sola instrucción. Si se requieren más instrucciones dentro de una marca, las instrucciones se deben encerrar entre llaves para convertirlas en una instrucción compuesta. Si se colocan dos o más instrucciones sin encerrar entre llaves entre el `if` y el `else`, entonces el compilador arrojará un mensaje de error.

Ejercicios de AUTOEVALUACIÓN

20. Escriba una instrucción `if-else` que despliegue la palabra `Alto` si el valor de la variable `marcador` es mayor a `100` y `Bajo` si el valor de `marcador` es a lo más `100`. La variable `marcador` es de tipo `int`.

21. Suponga que `ahorros` y `gastos` son variables de tipo `double` a las que se han asignado valores. Escriba una instrucción `if-else` que despliegue la palabra `Solvente`, reste al valor de `ahorros` al valor de `gastos`, y establezca el valor de `gastos` a `0`, previendo que `ahorros` sea menor que `gastos` entonces la instrucción `if-else` simplemente despliega la palabra `Bancarrota`, y no modifica el valor de variable alguna.

22. Escriba una instrucción `if-else` que despliegue la palabra `Autorizado` si el valor de la variable `examen` es mayor o igual a `60` y el valor de la variable `programas_realizados` es mayor o igual que `10`. De lo contrario, la instrucción `if-else` arroja la palabra `Falla`. Las variables `examen` y `programas_realizados` son ambas de tipo `int`.

23. Escriba una instrucción `if-else` que despliegue la palabra `Advertencia` si el valor de la variable `temperatura` es mayor o igual que `100`, o el valor de la variable `presion` es mayor o igual que `200`, o ambos. De lo contrario, la instrucción `if-else` despliega la palabra `OK`. Las variables `temperatura` y `presion` son ambas de tipo `int`.

24. Considere una expresión cuadrática, como la siguiente

    $x^2 - x - 2$

    Describir si esta es una función cuadrática positiva (esto es, mayor que 0), implica describir un conjunto de números que pudieran ser menores que la raíz cuadrada más pequeña (la cual es −1) o más grande que la mayor raíz (la cual es +2). Escriba una expresión booleana en C++ que sea verdadera cuando la fórmula contenga valores positivos.

25. Considere la expresión cuadrática

    $x^2 - 4x + 3$

    Describir si esta función cuadrática es negativa significa describir un conjunto de números que sean simultáneamente mayores que la raíz cuadrada más pequeña (+1) y menores que la raíz cuadrada más grande (+3). Escriba una expresión booleana que sea verdadera cuando el valor de esta función cuadrática es negativa.

26. ¿Cuál es la salida de las siguientes instrucciones `cout` incrustadas en estas instrucciones *`if-else`*? Asuma que éstas están incrustadas dentro de un programa completo y correcto. Explique su respuesta.

    a)
    ```
    if(0)
      cout << "0 es verdadero";
    else
      cout << "0 es falso";
    cout << endl;
    ```
    b)
    ```
    if(1)
      cout << "1 es verdadero";
    else
      cout << "1 es falso";
    cout << endl;
    ```
    c)
    ```
    if(-1)
      cout << "-1 es verdadero";
    else
      cout << "-1 es falso";
    cout << endl;
    ```

Nota: Éste solamente es un ejercicio. *No* tiene la intención de mostrarle el estilo de programación que debe seguir.

Mecanismos sencillos de ciclos

La mayoría de los programas incluyen algunas acciones que se repiten un número de veces. Por ejemplo, el programa del cuadro 2.6 calcula el pago bruto de un empleado. Si la empresa emplea a 100 empleados, entonces un programa más completo debe repetir el cálculo 100 veces. A la porción de un programa que repite una instrucción o un grupo de instrucciones se le llama **ciclo**. El lenguaje C++ cuenta con un número de maneras para crear ciclos. A una de estas construcciones se le llama **instrucción** *`while`* o **ciclo** *`while`*. Primero explicaremos su uso con un ejemplo pequeño de juguete y posteriormente haremos un ejercicio más realista.

instrucción *`while`*

El programa del cuadro 2.10 contiene una instrucción *`while`* sencilla en color. A la porción de código entre las llaves, `{` y `}`, se le llama **cuerpo** del ciclo *`while`*; es la acción que se repite. Las instrucciones dentro de las llaves se ejecutan en orden, y luego se ejecutan de nuevo, y de nuevo, y así sucesivamente hasta que el ciclo *`while`* termina. En el primer diálogo del ejemplo, el cuerpo se ejecuta tres veces antes de que termine el ciclo, así que el programa despliega `Hola` tres veces. Cada repetición del cuerpo del ciclo se denomina una **iteración** del ciclo, así que el primer ejemplo del diálogo muestra tres iteraciones del mismo.

cuerpo del ciclo

iteración

CUADRO 2.10 *Un ciclo* `while`

```
#include <iostream>
using namespace std;
int main( )
{
    int cuenta_regresiva;

    cout << "¿Cuantos saludos desea? ";
    cin >> cuenta_regresiva;

    while (cuenta_regresiva > 0)
    {
        cout << "Hola ";
        cuenta_regresiva = cuenta_regresiva - 1;
    }

    cout << endl;
    cout << "¡Eso es todo!\n";

    return 0;
}
```

Diálogo de ejemplo 1

```
¿Cuantos saludos desea? 3
Hola Hola Hola
¡Eso es todo!
```

Diálogo de ejemplo 2

```
¿Cuantos saludos desea? 1
Hola
¡Eso es todo!
```

Diálogo de ejemplo 3

```
¿Cuantos saludos desea? 0
¡Eso es todo!
```

El cuerpo del ciclo se ejecuta cero veces.

La palabra inglesa *while* sugiere el significado de la instrucción `while`. El ciclo se repite *mientras se satisface la expresión booleana en el paréntesis*. En el cuadro 2.10 esto significa que el cuerpo del ciclo se repite mientras el valor de la variable `cuenta_regresiva` es mayor que `0`. Consideremos el primer diálogo de ejemplo y veamos cómo se ejecuta el ciclo `while`. El usuario digita un `3` de tal forma que establece el valor de `cuenta_regresiva` en `3`. Así, en este caso, cuando el programa alcance a la instrucción `while`, con certeza es verdad que `cuenta_regresiva` es mayor que 0, así que se ejecutan las instrucciones en el cuerpo del ciclo. Cada vez que se repite el cuerpo del ciclo, se ejecuta la siguiente instrucción:

```
cout << "Hola";
cuenta_regresiva = cuenta_regresiva - 1;
```

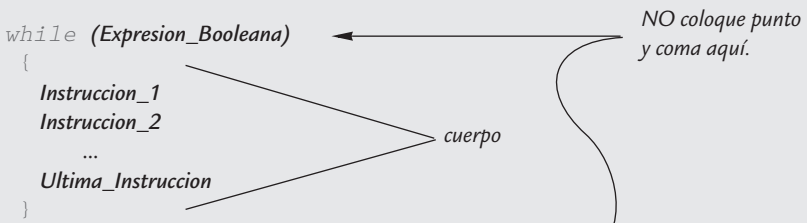
Por lo tanto, cada vez que se repite el cuerpo del ciclo, despliega `"Hola"` y el valor de la variable `cuenta_regresiva` disminuye en uno. Después de que la computadora repite el cuerpo del ciclo tres veces, el valor de `cuenta_regresiva` disminuye a `0` y la expresión booleana entre paréntesis ya no se satisface. Así, esta instrucción `while` termina después de repetir el cuerpo del ciclo tres veces.

La sintaxis para la instrucción `while` se da en el cuadro 2.11. Las *expresiones booleanas* permitidas son exactamente las mismas que las expresiones booleanas permitidas en la instrucción `if-else`. Tal como las instrucciones `if-else`, la expresión booleana en una expresión `while` debe estar encerrada entre paréntesis. En el cuadro 2.11 explicamos las plantillas de sintaxis para dos casos: el caso en donde existe más de una instrucción en el cuerpo del ciclo y el caso en donde existe una sola instrucción en el cuerpo del ciclo. Observe que cuando existe solamente una instrucción en el cuerpo del ciclo, no necesita incluir las llaves `{` y `}`.

Volvamos a las acciones realizadas por la instrucción `while` con más detalle. Cuando la instrucción `while` se ejecuta, la primera cosa que sucede es que se evalúa la expresión

CUADRO 2.11 *Sintaxis de la instrucción* `while`

Cuerpo del ciclo con varias instrucciones:



Cuerpo del ciclo con una sola instrucción:

booleana que se encuentra a continuación de la palabra *while*. Puede ser falsa o verdadera. Por ejemplo, la comparación

```
cuenta_regresiva > 0
```

es verdadera si el valor de cuenta_regresiva es positiva. Si es falsa, entonces no se toma acción alguna y el programa procede con la siguiente instrucción después de la instrucción *while*. Si la comparación es verdadera, entonces se ejecuta el cuerpo entero del ciclo. Por lo general, al menos una de las expresiones que se comparan contiene algo que podría modificar el cuerpo del ciclo, tal como el valor de cuenta_regresiva en la instrucción *while* del cuadro 2.10. Después de la ejecución del cuerpo del ciclo, se evalúa de nuevo la comparación. Este proceso se repite una y otra vez mientras la comparación sea verdadera. Después de cada iteración del cuerpo del ciclo, se evalúa de nuevo la comparación y si es verdadera, entonces el cuerpo del ciclo se ejecuta de nuevo. Cuando la comparación ya no es verdadera, la instrucción *while* llega a su fin.

ejecución del cuerpo del ciclo cero veces

Lo primero que sucede cuando se ejecuta un ciclo *while* es que se evalúa la expresión booleana. Si la expresión booleana no es verdadera cuando inicia el ciclo *while*, entonces nunca se ejecuta el cuerpo del ciclo. Esto es exactamente lo que sucede en el diálogo de ejemplo del cuadro 2.10. En muchas situaciones de programación querrá tener la posibilidad de ejecutar el cuerpo del ciclo cero veces. Por ejemplo, si su ciclo *while* lee todos los puntajes insuficientes para aprobar un examen y nadie reprobó dicho examen, entonces querrá que el cuerpo del ciclo se ejecute cero veces.

instrucción *do-while*

Como ya notamos, un ciclo *while* puede ejecutar su cuerpo de ciclo cero veces, lo cual deseará hacer con frecuencia. Si por otra parte sabe que *bajo cualquier circunstancia* el cuerpo del ciclo se debe ejecutar al menos una vez, entonces puede utilizar una instrucción *do-while*. Una instrucción *do-while* es similar a la instrucción *while* excepto que el ciclo siempre se ejecuta una vez. La sintaxis para la instrucción *do-while* aparece en el cuadro 2.12. En el cuadro 2.13 aparece un ejemplo del ciclo *do-while*. En ese ciclo *do-while*, lo primero que sucede es que se ejecutan las instrucciones en el cuerpo del ciclo. Después de la primera iteración del cuerpo del ciclo, la instrucción *do-while* se comporta de la misma manera que el ciclo *while*. Se evalúa la expresión booleana. Si la expresión booleana es verdadera, el cuerpo del ciclo se ejecuta de nuevo; se evalúa la expresión booleana de nuevo y así sucesivamente.

Operadores de incremento y decremento

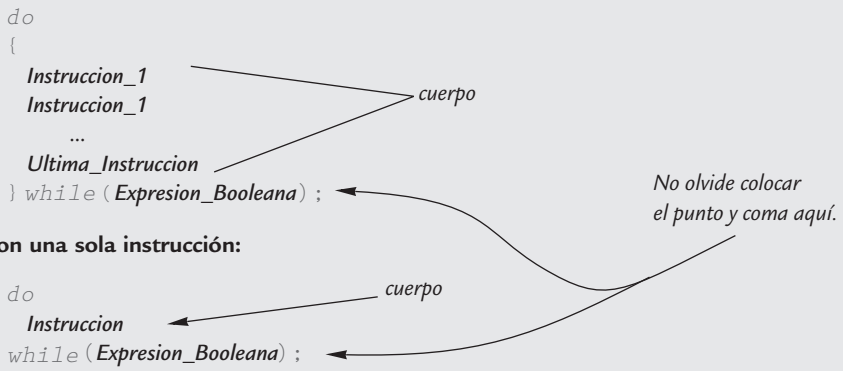
++ y −−

Explicaremos los operadores binarios en la sección titulada "expresiones y operadores aritméticos". Los operadores binarios tienen dos operandos. Los operadores unitarios contienen solamente un operando. Ya conocemos los dos operadores unitarios, + y −, tal como se utilizan en las expresiones +7 y −7. El lenguaje C++ tiene otros dos operadores unitarios muy comunes, ++ y−−. El operador ++ se llama **operador de incremento** y el operador −− se llama operador de decremento. Por lo general se utilizan con las variables de tipo *int*, entonces n++ incrementa el valor de n en uno y n−− decrementa el valor de n en uno. Así, n++ y n−− (cuando van seguidas de un punto y coma) son instrucciones ejecutables. Por ejemplo, las instrucciones

```
int n = 1, m = 7;
n++;
cout << "El valor de n se modifica a " << n << endl;
m--;
cout << "El valor de m se modifica a " << m << endl;
```

CUADRO 2.12 *Sintaxis de la instrucción* `do-while`

Cuerpo del ciclo con varias intrucciones:



Cuerpo del ciclo con una sola instrucción:

Cuadro 2.13 *Un ciclo* `do-while` *(parte 1 de 2)*

```
#include <iostream>
using namespace std;
int main( )
{
     char ans;

     do
     {
          cout << "Hola\n";
          cout << "Desea otro saludo?\n"
               << "Presione s para si, n para no,\n"
               << "y presione Intro: ";
          cin >> ans;
     } while (ans == 's' || ans == 'S');

     cout << "Adios \n";

     return 0;
}
```

CUADRO 2.13 *Un ciclo* `do-while` *(parte 2 de 2)*

Diálogo de ejemplo

```
Hola
Desea otro saludo?
Presione s para si, n para no,
y presione Intro: s
Hola
Desea otro saludo?
Presione s para si, n para no,
y presione Intro: S
Hola
Desea otro saludo?
Presione s para si, n para no,
y presione Intro: n
Adios
```

despliega la siguiente salida:

```
El valor de n se modifica a 2
El valor de m se modifica a 6
```

Y bien, ahora ya sabe de dónde proviene el nombre de "C++".

Con frecuencia las instrucciones de incremento y decremento se utilizan dentro de los ciclos. Por ejemplo, utilizamos las siguientes instrucciones en el ciclo `while` del cuadro 2.10:

```
cuenta_regresiva = cuenta_regresiva - 1;
```

Sin embargo, la mayoría de los programadores en C++ utilizan los operadores de incremento y decremento en lugar de instrucciones de asignación, de modo que el ciclo `while` completo se leerá de la siguiente manera:

```
while (cuenta_regresiva > 0)
{
    cout << "Hola";
    cuenta_regresiva--;
}
```

EJEMPLO DE PROGRAMACIÓN
Saldo de una tarjeta de débito

Suponga que tiene una tarjeta de débito con un saldo de $50 y que el banco carga a su cuenta 2% de interés mensual. ¿Cuántos meses puede dejar pasar sin hacer pago alguno antes de que su cuenta exceda los $100? Una manera de resolver este problema es simplemen-

te leer la instrucción cada mes y contar el número de meses que pasan hasta que el `saldo` alcanza el $100 o más. Pero sería mejor calcular los saldos mensuales con un programa en lugar de esperar a que las instrucciones lleguen. De esta forma obtendrá una respuesta sin tener que esperar tanto (y sin poner en peligro su historial crediticio).

Después de un mes el saldo será de $50 más 2%, que son $51. Después de dos meses el saldo será de $51 más 2%, que es $52.02. Después de tres meses el saldo será $52.02 más 2%, y así sucesivamente. En general, cada mes el saldo aumenta el %2. El programa podría dar seguimiento al saldo al almacenarlo en una variable llamada `saldo`. El cambio en el valor de `saldo` por un mes se puede calcular de la siguiente manera:

```
saldo = saldo + 0.02*saldo;
```

Si repetimos esta acción hasta que el valor de `saldo` alcance (o exceda) `100.00` y contamos el número de repeticiones, entonces conoceremos el número de meses que tomará al saldo alcanzar los 100.00. Para hacer esto necesitamos otra variable que cuente el número de veces que el saldo cambia. Llamemos a esta nueva variable `cuenta`. El cuerpo final de nuestro ciclo *`while`* también contendrá las siguientes instrucciones:

```
saldo = saldo + 0.02*saldo;
cuenta++;
```

Con el objeto de hacer que el ciclo se realice de forma correcta, debemos dar los valores apropiados a las variables `saldo` y `cuenta` antes de ejecutar el ciclo. En este caso, podemos iniciar las variables cuando las declaramos. El programa completo aparece en el cuadro 2.14.

RIESGO *Ciclos infinitos*

Un ciclo *`while`* o un ciclo *`do-while`* no termina mientras la expresión booleana *`while`* sea verdadera. Por lo general la expresión booleana contiene una variable que es modificada por el cuerpo del ciclo, y con frecuencia el valor de esta variable se modifica de tal forma que la expresión booleana sea falsa y por lo tanto termine el ciclo. Sin embargo, si comete un error y escribe su programa de forma que la expresión booleana sea siempre verdadera, entonces el ciclo se ejecutará infinitamente. Un ciclo que se ejecuta de modo infinito se conoce como **ciclo infinito**.

ciclo infinito

Primero describamos un ciclo que sí termina. El siguiente código en C++ escribe los números positivos pares menores a 12. Esto es, desplegará los números 2, 4, 6, 8, y 10, uno por línea, y luego el ciclo terminará.

```
x = 2;
while (x != 12)
{
   cout << x << endl;
   x = x +2;
}
```

CUADRO 2.14 *Programa de saldos de una tarjeta de débito*

```
#include <iostream>
using namespace std;
int main( )
{
     double saldo = 50.00;
     int cuenta = 0;

     cout << "Este programa le indica cuanto tardara usted \n"
          << "en acumular una deuda de $100, si su \n"
          << "saldo inicial es $50 de deuda.\n"
          << "La tasa de interes es de 2% mensual.\n";

     while (saldo < 100.00)
     {
          saldo = saldo + 0.02 * saldo;
          cuenta++;
     }

     cout << "Despues de " << cuenta << " meses,\n";
     cout.setf(ios::fixed);
     cout.setf(ios::showpoint);
     cout.precision(2);
     cout << "su saldo en contra sera de $" << saldo << endl;

     return 0;
}
```

Diálogo de ejemplo

```
Este programa le indica cuanto tardara usted
en acumular una deuda de $100, si su saldo inicial es $50 de deuda.
La tasa de interes es de 2% mensual.
Despues de 36 meses,
su saldo en contra sera de $101.99
```

El valor de x se incrementa en 2 durante cada iteración del ciclo hasta que alcanza 12. En ese punto, la expresión booleana después de la palabra *while* ya no es verdadera, por lo que el ciclo termina.

Suponga ahora que desea desplegar los números nones menores que 12, en lugar de los números pares. Podríamos pensar erróneamente que todo lo que necesita hacer es modificar la instrucción de inicio a

```
x = 1;
```

pero esta equivocación creará un ciclo infinito. Debido a que el valor de x va de 11 a 13, el valor de x nunca será igual a 12, por lo que el ciclo nunca terminará.

Esta clase de problema es común cuando los ciclos terminan al evaluar una cantidad mediante el uso de == o !=. Cuando se trate de números, siempre será más seguro evaluar un valor de paso. Por ejemplo, la siguiente expresión estará bien como la primera línea de nuestro ciclo *while*:

```
while (x < 12)
```

Con esta modificación, x se puede inicializar con cualquier valor y el ciclo de todas maneras terminará.

Un programa que se encuentra en un ciclo infinito se ejecutará por siempre a menos que alguna fuerza externa lo detenga. Dado que no puede escribir programas que contengan un ciclo infinito, es una buena idea aprender cómo forzar a un programa para que termine. El método para forzar a un programa a detenerse varía de un sistema a otro. Por lo general, la combinación de teclas Control+C terminarán el programa. (Oprima la tecla Control mientras presiona la tecla C.)

Ejercicios de AUTOEVALUACIÓN

27. ¿Cuál es la salida que produce la siguiente instrucción (cuando se encuentra incrustada en un programa correcto con *x* declarada como tipo *int*)?

```
x = 10;
while (x > 0)
{
   cout << x << endl;
   x = x 2 – 3;
}
```

28. ¿Cuál será la salida producida en el ejercicio previo si el signo > se remplaza con <?

29. ¿Cuál será la salida producida por el siguiente código (cuando se encuentra incrustada en un programa correcto con *x* declarada como tipo *int*)?

```
x = 10;
do
{
    cout << x << endl;
    x = x - 3;
} while (x > 0)
```

30. ¿Cuál será la salida producida por el siguiente código (cuando se encuentra incrustada en un programa correcto con *x* declarada como tipo *int*)?

```
x = -42;
do
{
    cout << x << endl;
    x = x - 3;
} while (x > 0)
```

31. ¿Cuál es la diferencia más importante entre una instrucción *while* y una instrucción *do-while*?

32. ¿Cuál será la salida producida por el siguiente código (cuando se encuentra incrustada en un programa correcto con *x* declarada como tipo *int*)?

```
x = 10;
while (x > 0)
{
   cout << x << endl;
   x = x + 3;
}
```

33. Escriba un programa completo en C++ que despliegue los números de 1 a 20, uno por línea. El programa no hace nada más.

2.5 Estilo de programación

"En asuntos de gran importancia, el estilo, no la sinceridad, es algo vital."
Oscar Wilde, *La importancia de llamarse Ernesto*

Todos los nombres de variables en nuestros programas de ejemplo se eligieron para sugerir su uso. Nuestros programas de ejemplo se escribieron con un formato en particular. Por ejemplo, todas las declaraciones y las instrucciones aparecen con sangrías iguales. Estas y otras características de estilo tienen algo más que sólo un interés estético. Un programa que está escrito con cuidadosa atención al estilo es más fácil de leer y más fácil de modificar.

Sangrado

Un programa debe estar diseñado de tal forma que los elementos que naturalmente son considerados un grupo estén hechos para parecer un grupo. Una manera de hacer esto es saltar una línea entre las partes que por su lógica son considerados por separado. El sangrado también puede ayudar a hacer más clara la estructura de un programa. Una instrucción dentro de otra instrucción debe sangrarse. En especial, las instrucciones *if-else*, los ciclos *while* o los ciclos *do-while* se deben sangrar en la forma en que están en nuestros programas de ejemplo o de alguna manera similar.

en dónde colocar las llaves { }

Las llaves { } determinan una parte larga de la estructura de un programa. Colocar cada llave en una línea independiente, como lo hemos hecho, hace más fácil localizar cada par de llaves. Observe que algunos pares de llaves tienen sangrías. Cuando un par de llaves se encuentra incrustado dentro de otro par de llaves, las llaves incrustadas tienen sangrías más grandes que las llaves exteriores. Vea el programa en el cuadro 2.14. Las llaves para el cuerpo del ciclo *while* tienen sangrías más grandes que las llaves para la parte *main* del programa.

Existen al menos dos escuelas para el uso correcto de las llaves. La primera, que utilizamos en el libro, es reservar una línea individual para cada llave. Esta forma es más fácil de leer. La segunda sostiene que la llave izquierda o de apertura de un par no necesita estar en una línea individual si se utiliza con cuidado, este segundo método puede ser efectivo y ahorra espacio. El punto importante es utilizar un estilo que muestre la estructura del programa. El formato exacto no existe, pero debe ser consistente dentro de cualquier programa.

Comentarios

// Comentarios

Con el objeto de hacer que un programa sea más claro, debe incluir algunas notas aclaratorias en lugares clave del programa. A dichas notas se les conoce como **comentarios**. C++ y la mayoría de los lenguajes de programación contemplan la inclusión de dichos comentarios dentro del texto de un programa. En C++ los símbolos // se utilizan para indicar el inicio de un comentario. Todo el texto entre // el final de la línea es un comentario. El compilador simplemente ignora cualquier cosa que siga a // dentro de una línea. Si desea un comentario que cubra más de una línea, coloque un // en cada línea de comentario. Los símbolos // son dos diagonales (sin espacio intermedio).

En la presente obra, los comentarios se escriben en cursivas de manera que resalten del texto del programa. Algunos editores de texto resaltan los comentarios mediante un color diferente del resto del texto del programa.

/* Comentarios */

Existe otra manera de insertar comentarios dentro de un programa en C++. Cualquier cosa entre el par de símbolos /* y el par de símbolos */ se considera un comentario, y el compilador lo ignora. A diferencia de los comentarios con //, que requieren un // por cada línea adicional, los comentarios de /* a */ se pueden prolongar a lo largo de varias líneas como en:

```
/* Éste es un comentario que se prolonga
a lo largo de tres líneas. Observe que no
existe un símbolo de comentario en la segunda línea.*/
```

Los comentarios de tipo /* */ se pueden insertar en cualquier parte del programa en donde se permita un espacio o un cambio de línea. Sin embargo, no se deben insertar en cualquier parte excepto en donde sean fáciles de leer y no distraigan la atención del resto del programa. Por lo general los comentarios se colocan solamente al final de las líneas o en líneas separadas por sí mismas.

Existen diversas opiniones acerca de qué tipo de comentario es mejor utilizar. Cualquier tipo (el // o el /* */) puede ser efectivo si se utiliza con precaución. Nosotros utilizaremos el tipo //.

cuándo insertar comentarios

Es difícil decir cuando o cuántos comentarios debe contener un programa. La única respuesta correcta es "suficientes", lo cual por supuesto le dice poco al programador novato. Le tomará algo de experiencia para adquirir la noción acerca de la mejor forma de incluir un comentario. Todo aquello que sea importante y no sea obvio, merece un comentario. Sin embargo, demasiados comentarios son tan malos como muy pocos. Un programa que tiene un comentario en cada línea estará tan saturado de comentarios que la estructura del programa quedará oculta en un mar de observaciones obvias. Los comentarios como el que aparece a continuación no contribuyen a la claridad del programa y no deben aparecer:

```
distancia = velocidad * tiempo; // Calcula la distancia recorrida
```

Observe el comentario que aparece al inicio del programa del cuadro 2.15. Todos los programas deben comenzar con un comentario similar al que ahí se muestra. Brinda la información básica relativa al programa: el nombre del archivo de programa, quién lo escribió, cómo ponerse en contacto con la persona que escribió el programa, qué hace el programa, la fecha en que lo modificaron por última vez y muchas otras particularidades que son apropiadas, tales como el números de asignación, si el programa es una tarea escolar. Lo que incluya exactamente en el programa dependerá en su situación particular. No incluiremos comentarios tan largos como esos en el resto del libro, pero siempre debe comenzar sus programas con dichos comentarios.

Nombres de constantes

Existen dos problemas con los números dentro de un programa de computadora. El primero es que no tienen un valor mnemónico. Por ejemplo, cuando se encuentra el número `10` dentro de un programa, no da clave alguna respecto a su significado. Si el programa es bancario, podría ser el número de oficinas o el número de ventanillas de caja de la oficina principal. Con el objeto de comprender el programa, debe conocer el significado de cada constante. El segundo problema es que cuando el programa necesita modificar algunas cifras, dicha modificación tiende a provocar errores. Suponga que el `10` aparece `12` veces en el programa bancario, de esas, cuatro veces representa el número de oficinas, y que ocho representa el número de ventanillas en la oficina principal. Cuando el banco abre una nueva oficina, tenemos que actualizar el programa, esto representa una posibilidad de que alguno de esos `10` no se modifique a `11`, o que algunos se modifiquen cuando no les corresponde. La manera de evitar estos problemas es darle un nombre a cada número y utilizar dicho nombre en lugar del número dentro de su programa. Por ejemplo, un programa bancario podría tener dos constantes con los nombres `NUMERO_OFICINAS` y `NUMERO_VENTANILLAS`. Ambos números podrían tener un valor igual a `10`, pero cuando el banco abra una nueva oficina, todo lo que necesita hacer para actualizar el programa es modificar la definición `NUMERO_OFICINAS`.

¿Cómo le pone nombre a un número en C++? Una manera de nombrar un número es iniciar una variable con el valor numérico deseado, como en el siguiente ejemplo:

```
int NUMERO_OFICINAS = 10;
int NUMERO_VENTANILLAS = 10;
```

CUADRO 2.15 *Comentarios y nombres de constantes*

```
//Nombre de Archivo: salud.cpp (Su sistema pudiera requerir algún sufijo diferente
//de cpp.)
//Autor: Su nombre va aquí.
//Dirección de Email: usted@sumaquina.bla.bla
//Número de asignación: 2
//Descripción: Programa para determinar si el usuario está enfermo.
//Última modificación: Septiembre 23, 2005

#include <iostream>
using namespace std;
int main( )
{
    const double NORMAL = 98.6;//grados Fahrenheit
    double temperatura;

    cout << "Introduzca su temperatura: ";
    cin >> temperatura;

    if (temperatura > NORMAL)
    {
        cout << "Usted tiene fiebre.\n";
        cout << "Tome mucho liquido y vayase a la cama.\n";
    }
    else
    {
        cout << "Usted no tiene fiebre.\n";
        cout << "Vaya a estudiar.\n";
    }

    return 0;
}
```

Sus programas deben comenzar siempre con un comentario similar a éste.

Diálogo de ejemplo

```
Introduzca su temperatura: 98.6
Usted no tiene fiebre.
Vaya a estudiar.
```

Sin embargo, existe un problema con este método para nombrar constantes: podría de forma inadvertida modificar el valor de una de estas variables. C++ proporciona una manera de marcar e inicializar de forma que no se pueda modificar. Si su programa intenta modificar alguna de estas variables se produce una condición de error. Para marcar una declaración de variable para que no se pueda modificar, anteceda la declaración con la palabra `const` (la cual es una abreviación de *constante*). Por ejemplo,

const

```
const int NUMERO_OFICINAS = 10;
const int NUMERO_VENTANILLAS = 10;
```

si las variables son del mismo tipo, es posible combinar las líneas previas de la siguiente manera:

```
const int NUMERO_OFICINAS = 10, NUMERO_VENTANILLAS = 10;
```

Sin embargo, la mayoría de los programadores piensan que colocar cada definición en una línea separada es más claro. Con frecuencia, a la palabra `const` se le conoce como un **modificador**, debido a que modifica (restringe) a las variables que se declaran.

constantes declaradas

Con frecuencia, a una variable que se declara con el modificador `const` se le llama **constante declarada**. C++ no requiere que las constantes declaradas se escriban con mayúsculas, pero es una práctica común entre los programadores de C++.

Una vez que un número se ha nombrado de esta forma, dicho nombre se puede utilizar en cualquier parte en donde dicho número sea permitido, y tendrá exactamente el mismo significado que el número al que nombra. Para modificar una constante declarada, solamente necesita modificar el valor con el que la inicia en la declaración de la variable. Por ejemplo, el significado de todas las ocurrencias de `NUMERO_OFICINAS` puede sencillamente modificarse de `10` a `11` al modificar el valor inicial de `10` en la declaración de `NUMERO_OFICINAS`.

Aunque dentro de un programa se permite el uso de las constantes numéricas no declaradas, intente no utilizarlas. Con frecuencia tiene sentido utilizar constantes no declaradas, fácilmente reconocibles, y valores que no se modificarán, como `100` para el número de centímetros en un metro. Sin embargo, todas las demás constantes numéricas deben tener nombres al estilo de lo que describimos con anterioridad. Esto hará que sus programas sean más fáciles de modificar.

El cuadro 2.15 contiene un programa de ejemplo que muestra el uso del modificador `const` en una declaración.

Cómo nombrar constantes con el modificador *const*

Cuando inicia una variable dentro de una declaración, puede marcar de manera que no se permita al programa que modifique su valor. Para hacer esto coloque el valor `const` antes de la declaración, como lo describimos a continuación:

Sintaxis *const Nombre_de_Tipo Nombre_de_Variable = Constante;*

Ejemplos

```
const int MAX_INTENTOS = 3;
const int PI = 3.14159;
```

Ejercicios de AUTOEVALUACIÓN

34. La siguiente instrucción *if-else* compilará y se ejecutará sin problemas. Sin embargo, no se basa en una forma consistente con otras instrucciones *if-else* que utilizamos en nuestros programas. Reescríbala de modo que su organización (sangrías y saltos de línea) coincida con el estilo que utilizamos en este capítulo.

```
if (x < 0) {x = 7; cout << "x ahora es positivo.";} else
{x = -7; cout << "x ahora es negativo.";}
```

35. ¿Cuál será la salida que producen las dos siguientes líneas (cuando se incrustan dentro de un programa completo y correctamente escrito)?

```
// cout << "Hola desde";
cout << "el ejercicio de autoevaluacion";
```

36. Escriba un programa en C++ que pregunte al usuario por un número de galones y que despliegue su equivalente en litros. Utilice una constante declarada. Ya que esto es solamente un ejercicio, no necesita colocar comentarios dentro de su programa.

Resumen del capítulo

- Utilice nombres de variables que sean significativos.
- Asegúrese de verificar que las variables se declaran con el tipo correcto.
- Asegúrese de que las variables se inicializan antes de intentar utilizar su valor. Esto se puede hacer cuando la variable se declara o se asigna antes de utilizar la variable.
- Utilice suficientes paréntesis dentro de expresiones aritméticas con el objeto de hacer que las operaciones sean más claras.
- Siempre incluya en un programa una línea de indicador cuando desee que el usuario introduzca datos desde su teclado, y siempre haga eco de la entrada del usuario.
- Una instrucción *if-else* permite a su programa elegir una de dos acciones alternativas. Una instrucción *if* permite a su programa decidir si realiza o no una acción en particular.
- Un ciclo *do-while* ejecuta su cuerpo de ciclo al menos una vez. En algunas situaciones un ciclo *while* podría no ejecutarse ni siquiera una vez.
- Por lo general, todas las variables dentro de un programa deben tener un nombre significativo que se puede utilizar en lugar de los números. Esto se puede hacer con el uso del modificador *const* en la declaración de una variable.
- Utilice sangrías y patrones de cambios de línea similares a los programas de ejemplo.
- Inserte comentarios para explicar las principales subsecciones o cualquier parte poco clara de un programa.

Respuestas a los ejercicios de autoevaluación

1.
```
int pies = 0, pulgadas = 0;
int pies(0), pulgadas(0);
```

2.
```
int cuenta = 0;
double distancia = 1.5;
```
Otra opción es
```
int cuenta(0);
double distancia(1.5);
```

3.
```
suma = n1 + n2;
```

4.
```
longitud = longitud + 8.3;
```

5.
```
producto = producto*n;
```

6. La salida real de un programa como éste depende del sistema y de la historia de uso del sistema.
```
#include <iostream>
using namespace std;
int main()
{
int primero, segundo, tercero, cuarto, quinto;
cout << primero << " " << segundo << " " << tercero
     << " " << cuarto << " " << quinto << endl;
return 0;
}
```

7. No existe una sola respuesta correcta para esta pregunta. A continuación aparecen las posibles respuestas:

a) `velocidad`
b) `tasa_pago`
c) `el_mas_alto` o `marcador_max`

8.
```
cout << "La respuesta a la pregunta de\n"
     << la vida, el Universo, y de Todo es 42.\n"
```

9.
```
cout << "Introduzca un número entero y presione Intro: ";
cin >> el_numero;
```

10.
```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(3);
```

11.
```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hola mundo\n";
    return 0;
}
```

12.
```
#include <iostream>
using namespace std;

int main()
{
    int n1, n2, suma;
    cout << "Introduzca dos números enteros\n";
    cin >> n1 >> n2;
    suma = n1 + n2;
    cout << "La suma de " << n1 << " y "
         << n2 << " es " << suma << endl;
    return 0;
}
```

13.
```
cout << endl << "\t";
```

14.
```
#include <iostream>

using namesapce std;

int main()

{

  double uno(1.0), dos(1.414), tres(1.732), cuatro(2.0),

         cinco(2.236);
  cout << "\tN\tRaiz cuadrada\n";
  cout << "\t1\t" << uno << endl
       << "\t2\t" << dos << endl
       << "\t3\t" << tres << endl
       << "\t4\t" << cuatro << endl
       << "\t5\t" << cinco << endl
  return 0;
}
```

15.
```
3*x
3*x + y
(x + y)/7 Observe que x + y/7 no es correcto.
(3*x + y)/(z + 2)
```

16. `bcbc`

17. `(1/3)*3 es igual a 0`

Dado que `1` y `3` son de tipo *`int`*, el operador / realiza la división, la cual descarta el residuo, así que el valor de `1/3` es `0`, no `0.3333`. Esto hace que el valor de toda la expresión es `0 * 3`, lo cual por supuesto es `0`.

18.
```
#include <iostream>
using namespace std;

int main()
{
    int numero1, numero2;
    cout << "Introduzca dos numeros enteros: ";
    cin << numero1 << numero2;
    cout << numero1 << " dividido por " << numero2
         << " iguales " << (numero1/numero2) << endl
         << "con un residuo de " << (numero1%numero2)
         << endl;
    return 0;
}
```

19. a) `52.0`

b) `9/5` contiene un valor *`int`* igual a `1`, ya que tanto el numerador como el denominador son de tipo *`int`*, por lo que se lleva a cabo una división entera; se descarta la parte fraccionaria.

c)
```
f = (9.0/5) * c + 32.0;
```
o esto
```
f = 1.8 * c + 32.0;
```

20.
```
if (marcador > 100)
   cout << "Alto"
else
    cout << "Bajo";
```

podría agregar una `\n` al final de las cadenas entrecomilladas dependiendo de los otros detalles del programa.

21.
```
if(ahorros >= gastos)
{
   ahorros = ahorros - gastos;
   gastos = 0;
   cout << "Solvente";
}
else
{
   cout << "En bancarrota";
}
```

Puede agregar \n al final de las cadenas entrecomilladas, dependiendo de otros detalles del programa.

22.
```
if ( (examen >= 60) && (programas_hechos >= 10) )
     cout << "Aprobado";
else
     cout << "Reprobado";
```

Puede agregar \n al final de las cadenas entrecomilladas dependiendo de otros detalles del programa.

23.
```
if ( (temperatura >= 100) || (presion >= 200) )
    cout << "Advertencia";
else
     cout << "OK";
```

Puede agregar \n al final de las cadenas entrecomilladas dependiendo de otros detalles del programa.

24. `(x < -1) || ( x > 2)`

25. `(1 < x) && (x < 3)`

26. a) `0 es falso.`

En la sección acerca de compatibilidad de tipos, observamos que el valor *int* igual a 0 se convierte en falso.

b) `1 es verdadero.`

En la sección acerca de compatibilidad de tipos, observamos que el valor *int* distinto de 0 se convierte en verdadero.

c) `-1 es verdadero.`

En la sección acerca de compatibilidad de tipos, observamos que el valor *int* distinto de 0 se convierte en verdadero.

27.
```
10
7
4
1
```

28. No habrá salida, ya que la expresión booleana `(x < 0)` no se satisface, así que la instrucción *while* termina la ejecución del cuerpo del ciclo.

29. La salida es exactamente la misma que la del ejercicio 27.

30. El cuerpo del ciclo se ejecuta antes de evaluar la expresión booleana, la expresión booleana es falsa y la salida es

```
-42
```

31. Con una instrucción *do-while* el cuerpo del ciclo siempre se ejecuta al menos una vez. Con una instrucción *while* pueden existir condiciones por medio de las cuales no se ejecuta el cuerpo del ciclo.

32. Este es un ciclo infinito. La salida comienza con la siguiente serie de números y continuará indefinidamente:

```
10
13
16
19
```

(Una vez que el valor de x se hace más grande que el entero más grande permitido en su computadora, el programa podría detenerse o exhibir un comportamiento extraño, pero el ciclo es por concepto un ciclo infinito.)

33.
```
#include <iostream>
using namespace std;
int main()
{
    int n = 1;
    while (n <= 20)
    {
        cout << n << endl;
        n++;
    }
    return 0;
}
```

34.
```
if (x < 0)

{
    x = 7;
    cout << "x es ahora positivo.";
}
else
{
     x = -7;
     cout << "x es ahora negativo.";
}
```

35. La primera línea es un comentario y no se ejecuta. Entonces la salida completa es solamente la línea que se muestra a continuación:

```
Ejercicio de autoevaluacion
```

36.
```
#include <iostream>
using namespace std;
int main()
{
    const double LITROS_POR_GALON = 3.78533;
    double galones, litros;

    cout << "Introduzca el numero de galones:\n";
    cin >> galones;

    litros = galones*LITROS_POR_GALON;
    cout << "Hay " << litros << " en "
         << galones << " galones .\n";

    return 0;
}
```

Proyectos de programación

1. Una tonelada métrica equivale a 35,273.92 onzas. Escriba un programa que lea el peso de un paquete de cereal para desayunos en onzas y que obtenga como salida el peso en toneladas métricas así como el número de cajas necesarias para llenar una tonelada métrica de cereal. El programa deberá permitir al usuario repetir este cálculo las veces que desee.

2. Un laboratorio de investigación del gobierno concluyó que un endulzante artificial de uso común en gaseosas dietéticas provocó la muerte en ratones de laboratorio. Un amigo suyo está desesperado por perder peso pero no puede renunciar a dicho refresco. Su amigo quiere saber cuánta gaseosa dietética puede beber sin morir a consecuencia de ello. Escriba un programa que dé la respuesta. La entrada del programa es la cantidad de endulzante artificial necesaria para matar a un ratón, el peso del ratón y el peso del sujeto a dieta. Para que su amigo esté fuera de peligro, asegúrese de que el programa pida el peso en el que el sujeto a dieta termina con ella, en lugar del peso actual del sujeto. Asuma que la gaseosa dietética contiene 1/10 del 1% de endulzante artificial. Utilice una declaración de variable con el modificador `const` para darle nombre a esa fracción. Podría expresar el porcentaje como el valor `double` 0.001. Su programa debe permitir la repetición del cálculo las veces que desee.

3. Los trabajadores de una empresa en particular ganaron un 7.6% de incremento retroactivo por seis meses. Escriba un programa que tome como entrada el salario anual previo del empleado y que despliegue el monto retroactivo del pago al empleado, el nuevo salario anual y el salario mensual. Utilice la declaración de variable con el modificador `const` para expresar el incremento del pago. El programa deberá permitir que el usuario repita este cálculo tantas veces como lo desee.

4. Modifique el programa del proyecto de programación 3 de manera que calcule el salario retroactivo para un trabajador para cualquier número de meses, en lugar de seis meses. El número de meses es introducido por el usuario.

5. Negociar un préstamo de consumidor no siempre es sencillo. Una forma de préstamo es el préstamo a plazos con descuento que funciona como sigue. Supongamos que un préstamo tiene un valor de $1000, la tasa de interés es del 15% y la duración es de 18 meses. El interés se calcula al multiplicar el valor del préstamo de $1000 por 0.15, que es $150. Esa cantidad se multiplica entonces por el periodo del préstamo que es 1.5 para que arroje $225 como el interés total. El monto se deduce de inmediato del valor del préstamo, lo que deja al consumidor con solamente $775. El siguiente pago se hace en pagos iguales basados en el valor del préstamo. Así el pago mensual del préstamo será de $1000 dividido entre 18, el cual es de $55.56. Este método de cálculo no sería malo si el consumidor necesita $775, pero el cálculo es un poco más complicado si el consumidor necesita $1000. Escriba un programa que tome tres entradas: el monto que necesita recibir el consumidor, la tasa de interés y la duración del préstamo en meses. Entonces, el programa debe calcular el valor que requiere para que el consumidor reciba el monto necesario. Debe calcular también el pago mensual. Además, el programa debe permitir que el usuario repita este cálculo tantas veces como lo desee.

6. Escriba un programa que determine si una sala de juntas está violando los reglamentos contra incendios respecto a la capacidad máxima del recinto. El programa leerá la capacidad máxima y el número de personas que asistirán a una junta. Si el número de personas es menor o igual a la capacidad máxima de dicho recinto, el programa anunciará que está permitido celebrar la reunión e indicará cuántas personas más podrán asistir sin violar la ley. Si el número de personas excede la capacidad máxima del recinto, el programa anunciará que la reunión no puede celebrarse según lo planeado a causa de los reglamentos contra incendios e indicará cuántas personas deben excluirse para cumplir con dichos reglamentos. Para versiones más complejas, escriba un programa que permita que el usuario repita este cálculo tantas veces lo desee. Si es un ejercicio de clase, pregunte a su instructor si debe programar la versión más difícil.

7. 
Un empleado recibe un pago por hora de $16.78 por las primeras 40 horas trabajadas en una semana. Cualquier número de horas adicionales se pagan como tiempo extra a una tarifa de uno y media veces el pago por hora. Del pago bruto del trabajador, 6% se retiene para el impuesto al Seguro Social, 14% por concepto de impuesto federal, 5% por concepto de impuesto estatal y 10% semanal por concepto de pagos al sindicato. Si el trabajador tiene tres o más dependientes, entonces se retiene un monto de $35 adicionales para cubrir el costo extra del seguro de salud que el empleado ya pagó. Escriba un programa que lea como entrada el número de horas trabajadas en una semana y el número de dependientes, y que despliegue el pago bruto del trabajador, cada monto retenido y el monto neto del pago por semana. Para una versión más difícil, escriba el programa de tal manera que permita que el cálculo se repita las veces que el usuario lo desee. Si éste es un ejercicio de clase, pregunte a su instructor si debe hacer esta versión más difícil.

8. 
Es difícil hacer un presupuesto que se extienda a lo largo de varios años, debido a que los precios no son estables. Si su empresa necesita 200 lápices por año, no puede solamente utilizar el precio de este año como el costo de lápices dentro de dos años. Debido a la inflación, el costo probablemente será mayor de lo que es hoy. Escriba un programa que pronostique el costo esperado de un producto dentro de un número específico de años. El programa debe preguntar el costo del producto, el número de años a partir del actual en el que se comprará el producto y la tasa de la inflación. Entonces el programa debe desplegar el costo estimado de un producto en el periodo especificado. Que el usuario introduzca la tasa inflacionaria como un porcentaje, como 5.6 (por ciento). Entonces, su programa debe convertir el porcentaje en fracción como 0.056 y debe utilizar un ciclo para estimar el precio ajustado a la inflación. (*Sugerencia:* Esto es similar el cálculo de interés en una tarjeta de débito, el cual explicamos en este capítulo.)

9. Usted acaba de comprar un equipo de sonido que tiene un valor de $1000 con el siguiente plan de crédito: no hay pago inicial, una tasa de interés del 18% anual (es decir, un 1.5% mensual), y un pago mensual de $50. El pago mensual de $50 se utilizará para pagar el interés y el resto se utilizará para pagar parte de la deuda. Así, el primer mes paga 1.5% de $1000 de interés. Esto es $15 de interés. Así, los $35 restantes se deducen de su deuda, lo cual deja una cantidad de $965.00. El siguiente mes paga un interés de 1.5%

de $965.00, cuyo monto es de $14.48. Así, podemos deducir $35.52 (lo cual es $50 − $14.48) del monto de su deuda. Escriba un programa que le indique cuántos pagos le tomará pagar el préstamo, así como el monto total del interés pagado sobre la vida del préstamo. Utilice un ciclo para calcular el monto del interés y el tamaño de la deuda después de cada monto. (Su programa final no necesita desplegar el monto mensual del interés pagado y de la deuda restante, pero podría escribir una versión preliminar del programa que despliegue dichos valores). Utilice una variable para contar el número de iteraciones del ciclo y por lo tanto el número de meses hasta que su deuda sea cero. Además podría utilizar también otras variables. El último pago podría ser menor a $50. No olvide el interés del último pago. Si debe $50, entonces su pago mensual de $50 no pagará su deuda, aunque estará muy cerca. Un mes de interés para $50 es de solamente 75 centavos.

10. 

Escriba un programa que lea 10 números y que despliegue la suma de todos los números mayores a cero, la suma de todos los números menores que cero (la cual será un número menor o igual a cero) y la suma de todos los números, ya sean positivos, negativos o cero. El usuario debe introducir diez números una a uno y los puede introducir en cualquier orden. Su programa no debe pedir al usuario que introduzca los números positivos y negativos en forma separada.

11. Modifique el programa 10 de los proyectos de programación de modo que despliegue la suma de todos los números positivos, el promedio de todos los números positivos, la suma de todos los números negativos, el promedio de todos los números negativos, la suma de todos los números negativos y el promedio de todos los números introducidos.

3 Abstracción de procedimientos y funciones que devuelven un valor

3

Abstracción de procedimientos y funciones que devuelven un valor

Había un arquitecto de lo más ingenioso que había ideado un nuevo método para construir casas, comenzando desde el techo y trabajando hacia abajo hasta los cimientos.

JONATHAN SWIFT, *Los viajes de Gulliver*

Introducción

Podemos pensar que un programa se compone de subpartes como obtener los datos de entrada, calcular los datos de salida y desplegar los datos de salida. C++, al igual que la mayoría de los lenguajes de programación, cuenta con recursos para nombrar y codificar por separado cada una de estas subpartes. En C++ las subpartes se llaman funciones. En este capítulo presentaremos la sintaxis básica de uno de los dos tipos principales de funciones de C++, las diseñadas para calcular un solo valor. También veremos cómo dichas funciones pueden ayudarnos a diseñar programas. Comenzaremos con un análisis de un principio de diseño fundamental.

Prerrequisitos

Es aconsejable que lea el capítulo 2, y que al menos revise brevemente el capítulo 1, antes de leer el presente capítulo.

3.1 Diseño descendente

Recuerde que para escribir un programa primero diseñamos el método que el programa usará y lo escribiremos en español, como si una persona fuera a seguir las instrucciones. Como vimos en el capítulo 1, este conjunto de instrucciones se conoce como *algoritmo*. Un buen plan para enfrentarse al diseño del algoritmo es dividir en subtareas las labores por realizar, descomponer cada una de las subtareas en subtareas más pequeñas y así sucesivamente. En algún momento, las subtareas se vuelven tan pequeñas que resulta trivial implementarlas en C++. A este método se le conoce como **diseño descendente**. (Aunque también se le conoce como **refinación paso a paso**, o de forma figurada **divide y vencerás**).

diseño descendente

Mediante el método descendente, diseñamos un programa separando la labor del programa en subtareas y resolviéndolas con subalgoritmos. Si conservamos esta estructura descendente en nuestro programa en C++ el programa será más fácil de entender, modificar y, cómo veremos en breve, más fácil de escribir, probar y depurar. C++, al igual que la mayoría de los lenguajes de programación, cuenta con herramientas para incluir subtareas individuales en un programa. En otros lenguajes de programación a estas subtareas se les llama *subprogramas*, *procedimientos* o *métodos*. En C++ a estas subtareas se les conoce como **funciones**.

funciones para trabajo en equipo

Una de las ventajas de utilizar funciones para dividir una tarea de programación en subtareas, es que distintas personas pueden trabajar en diferentes subtareas. Cuando se produce un programa muy grande, como un compilador o un sistema de administración de oficina, el trabajo en equipo es indispensable para que el programa esté listo en un tiempo razonable. Comenzaremos nuestro tratamiento de las funciones explicando cómo podemos emplear las funciones que fueron escritas por alguien más.

3.2 Funciones predefinidas

C++ contiene bibliotecas de funciones predefinidas que podemos utilizar en nuestros programas. Antes de explicar cómo definir funciones, mostraremos cómo utilizar las que ya lo están.

Uso de funciones predefinidas

Utilizaremos la función `sqrt` para ilustrar cómo utilizar funciones predefinidas. La función `sqrt` calcula la raíz cuadrada de un número. (La raíz cuadrada de un número es aquel que, cuando se multiplica por sí mismo, produce el número con el que inició. Por ejemplo, la raíz cuadrada de 9 es 3, porque 3^2 es igual a 9.) La función `sqrt` inicia con un número, como 9.0, y calcula su raíz cuadrada, que en este caso es 3.0. El valor con que inicia la función se conoce como **argumento**. Al valor que calcula se le llama **valor devuelto**. Algunas funciones pueden tener más de un argumento, pero ninguna función devuelve más de un valor. Si piensa en la función como un pequeño programa, entonces los argumentos serían el análogo de los datos de entrada, y el valor devuelto sería análogo a los datos de salida.

argumento
valor devuelto

La sintaxis para utilizar funciones en un programa es sencilla. Si queremos hacer que una variable llamada `la_raiz` contenga la raíz cuadrada de `9.0`, podemos utilizar la siguiente instrucción de asignación:

```
la_raiz = sqrt(9.0)
```

llamada de función

La expresión `sqrt(9.0)` se conoce como una **llamada de función** (o si quiere verse elegante, puede decir **invocación de función**). El argumento en una llamada de función puede ser una constante, por ejemplo `9.0`, variables o expresiones más complicadas. Una llamada de función es una expresión que puede utilizarse como cualquier otra expresión. Podemos emplear una llamada de función en cualquier lugar que se permite usar una expresión del tipo especificado para el valor devuelto por la función. Por ejemplo, el valor devuelto por `sqrt` es de tipo *`double`*. Por lo tanto, la siguiente expresión es válida (aunque no muy generosa):

```
bono = sqrt(ventas)/10;
```

`ventas` y `bono` son variables que normalmente serían de tipo *`double`*. La llamada de función `sqrt(ventas)` es una sola cosa, como si todo estuviera encerrado entre paréntesis. Así pues, la instrucción de asignación anterior equivale a

```
bono = (sqrt(ventas))/10;
```

También podemos usar una llamada de función directamente en una instrucción `cout`, como se muestra a continuación

```
cout << "El lado de un cuadrado con un area de" << area
     << " es " << sqrt(area);
```

El cuadro 3.1 contiene un programa completo que usa la función predefinida `sqrt`. El programa calcula el tamaño de la perrera cuadrada más grande que se puede construir con la cantidad de dinero que el usuario está dispuesto a gastar. El programa pide al usuario una cantidad de dinero y luego determina cuántos pies cuadrados de espacio de piso se pueden comprar con esa cantidad. El cálculo produce un área en pies cuadrados para el piso de la perrera. La función `sqrt` produce la longitud de un lado del piso de la perrera.

Observe que el programa del cuadro 3.1 contiene otro elemento nuevo:

```
#include <cmath>
```

Llamada de función

Una llamada de función es una expresión que consiste en el nombre de la función seguido de los argumentos encerrados entre paréntesis. Si hay más de un argumento, los argumentos se separan con comas. Una llamada de función es una expresión que se puede usar como cualquier otra expresión del tipo especificado para el valor devuelto por la función

Sintaxis

Nombre_de_Funcion(*Lista_de_Argumentos*)

donde la *Lista_de_Argumentos* es una lista de argumentos separados por comas:

Argumento_1, *Argumento_2*, ... , *Ultimo_Argumento*

Ejemplos

```
lado = sqrt(area)
cout << "2.5 elevado a la potencia 3.0 es"
     << pot(2.5, 3.0);
```

CUADRO 3.1 *Una llamada de función*

```
//Calcula el tamaño de la perrera que se puede comprar
//con el presupuesto del usuario.
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    const double COSTO_POR_PIE^2 = 10.50;
    double presup, area, long_lado;

    cout << "Introduzca la cantidad presupuestada para su perrera $";
    cin >> presup;

    area = presup/COSTO_POR_PIE^2;
    long_lado = sqrt(area);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Por un precio de $" << presup << endl
         << "puede comprar una lujosa perrera cuadrada \n"
         << "que mide " << long_lado
         << " pies por lado.\n";
    return 0;
}
```

Diálogo de ejemplo

```
Introduzca la cantidad presupuestada para su perrera $25.00
Por un precio de $25.0
puede comprar una lujosa perrera cuadrada
que mide 1.54 pies por lado
```

Esa línea se parece mucho a ésta

```
#include <iostream>
```

directiva `#include` y archivo de encabezado

y de hecho las dos líneas tienen funciones similares. Como señalamos en el capítulo 2, tales líneas se llaman **directivas** `include`. El nombre incluido en los paréntesis angulares < > es el nombre de un archivo llamado **archivo de encabezado**. El archivo de encabezado de

una biblioteca proporciona al compilador cierta información básica acerca de la biblioteca, y la directiva `include` proporciona dicha información al compilador. Esto permite al enlazador (linker) encontrar el código objeto de las funciones en la biblioteca y así enlazar correctamente la biblioteca con nuestro programa. Por ejemplo, la biblioteca `iostream` contiene las definiciones de `cin` y `cout`, y el archivo de encabezado para la biblioteca iostream se llama `iostream`. La biblioteca de matemáticas contiene la definición de la función `sqrt` y varias otras funciones matemáticas, y el archivo de encabezado de esa biblioteca es `math`. Si nuestro programa usa una función predefinida de alguna biblioteca, deberá contener una directiva que nombre al archivo de encabezado como se muestra a continuación

```
#include <cmath>
```

Asegúrese de seguir la sintaxis que se ilustra en nuestros ejemplos. No olvide los símbolos < y >; son los mismos que se emplean para menor que y mayor que. No debe haber ningún espacio entre < y el nombre de archivo ni entre el nombre de archivo y el >. Además, algunos compiladores exigen que no haya espacios antes ni después del #; por ello, lo más seguro es colocar el # al principio de la línea y nunca dejar ningún espacio entre el # y la palabra `include`. Las directivas `#include` por lo regular se colocan al principio del archivo que contiene el programa.

Como vimos anteriormente, la directiva

```
#include <iostream>
```

debe ir seguida además de la directiva *`using`*:

```
using namespace std;
```

Esto se debe a que la definición de nombres como `cin` y `cout`, los cuales se dan en `iostream`, definen esos nombres como parte del nombre de espacios `std`. Esto es cierto para la mayoría de las librerías. Si tenemos una directiva `include` para una librería estándar como

```
#include <cmath>
```

entonces probablemente necesitaremos la directiva *using*:

```
using namespace std;
```

No hay necesidad de utilizar múltiples copias de esta directiva *`using`*, cuando se tienen múltiples directivas include.

la directiva `#include` podría ser insuficiente

En general, todo lo que debe hacer para utilizar una biblioteca es colocar en el archivo del programa una directiva `include` y una directiva *`using`* para esa biblioteca. Si las cosas funcionan con las directivas `include` y `using`, no tiene que preocuparse por hacer algo más. Sin embargo, para algunas bibliotecas de ciertos sistemas, es probable que necesite dar instrucciones adicionales al compilador, o ejecutar explícitamente un programa enlazador para enlazar la biblioteca. Las primeras versiones de los compiladores de C y C++ no buscaban automáticamente a todas las bibliotecas para enlazarlas. Los detalles varían de un sistema a otro, por lo que tendría que revisar su manual o consultar a un experto local para ver exactamente lo que se necesita.

preprocesador

Algunas personas le dirán que el compilador no procesa las directivas `include`, que quien lo hace es un **preprocesador**. Tienen razón, pero la diferencia es sólo cuestión de terminología y no debe preocuparnos realmente. En casi todos los compiladores el preprocesador se invoca automáticamente al compilar un programa.

abs y labs

En el cuadro 3.2 se describen algunas funciones predefinidas; en el apéndice 4 se describen más. Observe que las funciones de valor absoluto, `abs` y `labs`, se encuentran en la biblioteca con el archivo de encabezado `cstdlib`, por lo que un programa que utilice cualquiera de estas funciones debe contener la siguiente directiva:

```
#include <cstdlib>
```

Todas las demás funciones listadas se encuentran en la biblioteca con el archivo de encabezado `cmath`, por ejemplo `sqrt`.

Además, observe que existen tres funciones de valor absoluto. Si queremos producir el valor absoluto de un número de tipo *`int`*, usamos `abs`; si queremos producir el valor absoluto de un número de tipo *`long`*, debemos utilizar `labs`; si queremos producir el valor absoluto de un número de tipo *`double`*, usamos `fabs`. Para complicar las cosas aún más, `abs` y `labs` se encuentran en la biblioteca con el archivo de encabezado `cstdlib`, mientras que `fabs` está en la biblioteca con el archivo de encabezado `cmath`. `fabs` es una abreviatura de *valor absoluto de punto flotante*. Recuerde que los números con una fracción después del punto decimal, como los números de tipo *`double`*, con frecuencia se conocen como *números de punto flotante*.

fabs

pow

Otro ejemplo de funciones predefinidas es `pow`, que se encuentra en la biblioteca con archivo de encabezado `cmath`. La función `pow` puede utilizarse para la exponenciación en C++. Por ejemplo, si quiere establecer la variable `resultado` como igual a x^y, puede utilizar lo siguiente:

```
resultado = pow(x, y);
```

CUADRO 3.2 *Algunas funciones predefinidas*

Nombre	Descripción	Tipo de argumentos	Tipo de valor devuelto	Ejemplo	Valor	Encabezado de biblioteca
`sqrt`	raíz cuadrada	*`double`*	*`double`*	`sqrt(4.0)`	`2.0`	`cmath`
`pow`	potencia	*`double`*	*`double`*	`pow(2.0, 3.0)`	`8.0`	`cmath`
`abs`	valor absoluto para *`int`*	*`int`*	*`int`*	`abs(-7)` `abs(7)`	`7` `7`	`cstdlib`
`labs`	valor absoluto para *`long`*	*`long`*	*`long`*	`labs(-70000)` `labs(70000)`	`70000` `70000`	`cstdlib`
`fabs`	valor absoluto para *`double`*	*`double`*	*`double`*	`fabs(-7.5)` `fabs(7.5)`	`7.5` `7.5`	`cmath`
`ceil`	límite superior (redondear hacia arriba)	*`double`*	*`double`*	`ceil(3.2)` `ceil(3.9)`	`4.0` `4.0`	`cmath`
`floor`	límite inferior (redondear hacia abajo)	*`double`*	*`double`*	`floor(3.2)` `floor(3.9)`	`3.0` `3.0`	`cmath`

Entonces, las siguientes tres líneas de código del programa desplegarán en la pantalla el número 9.0, ya que $(3.0)^{20}$ es 9.0:

```
double resultado, x = 3.0, y= 2.0;
resultado = pow(x, y);
cout << resultado;
```

los argumentos tienen un tipo

Observe que la llamada de arriba a la función pow, devuelve 9.0 y no 9. Esta función siempre devuelve un valor de tipo *double*, no de tipo *int*. También observe que la función pow requiere dos argumentos. Una función puede tener cualquier cantidad de argumentos. Además, cualquier posición de argumento tiene un tipo específico, y el argumento utilizado en una llamada de función debe ser de ese tipo. En muchos casos, si utiliza un argumento de tipo erróneo, entonces C++ realizará automáticamente algunas conversiones de tipo. Sin embargo, los resultados pueden no ser los deseados. Cuando llame a una función, debe utilizar los argumentos del tipo especificado para esa función. Una excepción a esta advertencia es la conversión de argumentos de tipo *int* a tipo *double*. En muchas situaciones, como en las llamadas a la función pow, podemos utilizar, sin peligro, un argumento de tipo *int* cuando se especifica un argumento de tipo *double*.

restricciones de pow

Muchas implementaciones de pow tienen una restricción con respecto a los argumentos que pueden utilizarse. En estas implementaciones, si el primer argumento de pow es negativo, entonces el segundo debe ser un número entero. Debido a que probablemente tenga suficientes cosas de que preocuparse mientras aprende a programar, sería más sencillo y seguro que sólo utilizara pow cuando el primer argumento sea positivo.

Conversión de tipo

La división puede requerir el tipo *double*

Recuerde que 9/2 es una división entera, y da como resultado 4 y no 4.5. Si queremos que la división produzca una división de tipo *double* (es decir, que incluya la parte fraccionaria después del punto decimal), al menos uno de los dos números de la división debe ser de tipo *double*. Por ejemplo, 9/2.0 arroja 4.5. Si uno de los dos números se da como constante, podemos añadir simplemente un punto decimal y un cero a uno o a ambos, y la división producirá un valor que incluirá los dígitos después del punto decimal.

Pero, ¿qué sucede si los dos operandos de la división son variables, como en el siguiente caso?

```
int dulces_totales, numero_de_personas;
double dulces_por_persona;
<El programa de alguna manera establece el valor de dulces_totales en 9
  y el valor de numero_de_personas en 2.
  No importa como es que el programa hace esto.>
dulces_por_persona = dulces_totales/numero_de_personas;
```

A menos que convirtamos el valor de una de las variables dulces_totales o numero_de_personas a un valor de tipo *double*, entonces el resultado de la división será 4, y no 4.5 como debería ser. El hecho de que la variable dulces_por_persona sea de tipo *double*, no nos ayuda. El valor 4, obtenido de la división, se convertirá en un valor de tipo *double*, antes de almacenarse en la variable dulces_por_persona, pero ya será demasiado tarde. El 4 se convertirá en 4.0 y el valor final de dulces_por_persona será 4.0, y no 4.5. Si una de las cantidades de la división fuera una constante, podría agregar un punto decimal y un cero para convertirla en tipo *double*, pero en este caso ambas cantidades son variables.

Por fortuna, existe una forma de convertir del tipo *int* al tipo *double*, que podemos utilizar tanto con constantes como con variables.

En C++, podemos indicarle a la computadora que convierta un valor de tipo *int* en un valor de tipo *double*. La forma en que debe escribir "Convierte el valor 9 a uno de tipo *double*" es

```
static_cast<double>(9)
```

conversión de tipo

La notación *static_cast<double>* es un tipo de función predefinida que convierte un valor de cierto tipo, como 9, en un valor de tipo *double*, en este caso 9.0. Una expresión como *static_cast<double>*(9) se conoce como **conversión de tipo**. Podemos utilizar una variable u otra expresión en lugar de 9. También es posible utilizar otro tipo de nombres además de *double* para obtener una conversión de tipo diferente de *double*, pero pospondremos este tema para más adelante.

Por ejemplo, en el siguiente utilizamos una conversión de tipo para cambiar el tipo *int* de 9 a un tipo *double*, para que el valor de respuesta se establezca en 4.5:

```
double respuesta;
respuesta = static_cast<double>(9)/2;
```

La conversión de tipo aplicada a una constante como 9 puede hacer que su código sea más fácil de leer, ya que hace que su significado sea más claro. Sin embargo, la conversión de tipo aplicada a constantes de tipo *int* no nos proporciona mayor capacidad. Podemos utilizar 9.0 en lugar de *static_cast<double>*(9) cuando quiera convertir 9 en un valor de tipo *double*. Sin embargo, si la división sólo involucra variables, la conversión de tipo podría ser la única alternativa viable. Por medio de la conversión de tipo podemos reescribir nuestro primer ejemplo para que la variable dulces_por_persona reciba el valor correcto de 4.5 en lugar de 4.0; para hacerlo, el único cambio que necesitamos es reemplazar dulces_totales por *static_cast<double>*(dulces_totales)/numero_de_personas, como muestran las siguientes líneas:

```
int dulces_totales, numero_de_personas;
double dulces_por_persona;
<El programa de alguna manera establece el valor de dulces_totales en 9
  y el valor de numero_de_personas en 2.
  No importa cómo es que el programa hace esto.>
dulces_por_persona =
            static_cast<double>(dulces_totales)/numero_de_personas;
```

¡Advertencia!

Observe la colocación de los paréntesis en la conversión de tipo que utilizamos en el código. Es necesario realizar la conversión de tipo antes de la división para que el operador se aplique en un valor de tipo *double*. Si esperamos hasta que la división se realice, entonces los dígitos que se encuentran después del punto decimal ya estarán perdidos. Si erróneamente utilizamos lo que sigue como última línea del código anterior, entonces el valor de dulces_por_persona será 4.0 y no 4.5.

```
dulces_por_persona =
static_cast<double>(dulces_totales/numero_de_personas); // ¡ERROR!
```

Forma antigua de la conversión de tipo

***double* como una función**

El uso de *static_cast<double>*, tal como explicamos en la sección anterior, es la mejor forma para llevar a cabo una conversión de tipo. Sin embargo, en versiones anteriores de C++ se utilizaba una notación diferente para dicha conversión. Esta notación anterior sim-

Una función para convertir de `int` a `double`

La notación `static_cast<double>` puede utilizarse como una función predefinida y convertirá un valor de algún otro tipo en un valor de tipo `double`. Por ejemplo, `static_cast <double>(2)` devuelve `2.0`. A esto se le llama conversión de tipo. (Esta conversión puede hacerse con otros tipos además de `double`, pero lo haremos más adelante, por el momento, seguiremos haciendo conversiones sólo con el tipo `double`.)

Sintaxis

`static_cast<double>`(*Expresion_de_tipo_int*)

Ejemplo

```
int (pozo_total), numero_de_ganadores;
double sus_ganancias;
  . . .
sus_ganancias =
  static_cast<double>(pozo_total)/numero_de_ganadores;
```

plemente utiliza el nombre de tipo como si fuera un nombre de función, por lo que `double(9)` devuelve `9.0`. Por lo tanto, si `dulces_por_persona` es una variable de tipo `double`, y si tanto `dulces_totales` como `numero_de_personas` son variables de tipo `int`, entonces las siguientes dos instrucciones de asignación son equivalentes:

```
dulces_por_persona =
          static_cast<double>(dulces_totales)/numero_de_personas;
```

y

```
dulces_por_persona =
           double(dulces_totales)/numero_de_personas;
```

Aunque `static_cast<double>(dulces_totales)` y `double (dulces_totales)` son más o menos equivalentes, es necesario que utilice la forma `static_cast<double>`, ya que la forma `double (dulces_totales)` podría estar descontinuada en versiones más recientes de C++.

RIESGO *La división entera desecha la parte fraccionaria*

En una división entera, como 11/2, es fácil olvidar que el resultado será 5 y no 5.5. El resultado es el siguiente entero menor. Por ejemplo,

```
double d;
d= 11/2;
```

Aquí, la división se hace utilizando enteros; el resultado de la división es `5`, el cual se convierte en `double`, y después se asigna a `d`. La parte fraccionaria no se genera. Observe que el hecho de que `d` sea de tipo `double`, no cambia el resultado de la división. La variable `d` recibe el valor de `5.0` y no de `5.5`.

Ejercicios de AUTOEVALUACIÓN

1. Determine el valor de cada una de las siguientes expresiones aritméticas:

```
sqrt(16.0)          sqrt(16)            pow(2.0, 3.0)
pow(2, 3)           pow(2.0, 3)         pow(1.1, 2)
abs(3)              abs(-3)             abs(0)
fabs(-3.0)          fabs(-3.5)          fabs(3.5)
ceil(5.1)           ceil(5.8)           floor(5.1)
floor(5.8)          pow(3.0, 2)/2.0     pow(3.0, 2)/2
7/abs(-2)           (7 + sqrt(4.0))/3.0  sqrt(pow(3, 2))
```

2. Convierta cada una de las expresiones matemáticas siguientes en expresiones aritméticas de C++:

$\sqrt{x + y}$ $\quad x^{y+7}$ $\quad \sqrt{área + perímetro}$

$\dfrac{\sqrt{tiempo + corriente}}{nadie}$ $\quad \dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$ $\quad |x - y|$

3. Escriba un programa completo en C++ que calcule y arroje la raíz cuadrada de PI; PI es aproximadamente 3.14159. La `const double PI` está predefinida en `cmath`. Le invitamos a utilizar esta constante predefinida.

4. Escriba y compile programas cortos para probar lo siguiente:

   a) Determine si su compilador permitirá colocar la directiva `#include <iostream>` en cualquier parte de la línea, o si necesita que el # esté alineado de acuerdo con el margen izquierdo.

   b) Determine si su compilador permitirá un espacio entre # e `include`.

3.3 Funciones definidas por el programador

Un traje hecho a la medida, siempre luce mejor que uno del montón.
Mi tío, el sastre

En la sección anterior explicamos cómo utilizar funciones predefinidas. En ésta veremos cómo definir nuestras propias funciones.

Definiciones de función

Podemos definir nuestras propias funciones, ya sea en el mismo archivo que la parte `main` de nuestro programa, o en un archivo aparte de modo que distintos programas puedan utilizar dichas funciones. La definición es la misma en ambos casos, pero por el momento asumiremos que la definición de funciones se hará en el mismo archivo que la parte `main` del programa.

El cuadro 3.3 contiene un ejemplo de definición de función en un programa completo que demuestra una llamada a la función. El nombre de la función es llamada `costo_total`. Ésta recibe dos argumentos, el precio de un artículo y el número de artículos en una compra. La función devuelve el costo total, incluyendo el impuesto de venta para todos esos elementos al precio especificado. A la función se le llama de la misma forma en que se llama a una función predefinida. La descripción de la función, la cual debe escribir el programador, es un poco más complicada.

La descripción de la función se da en dos partes que se conocen como *declaración de función* y *definición de función*. La **declaración de función** (también conocida como el **prototipo de función**) describe cómo se le llama a la función. C++ requiere que la definición completa de la función o bien la declaración de la función aparezca en el código, antes de que la función sea invocada. En la parte superior del cuadro 3.3 se muestra en negritas la declaración de función `costo_total` y lo reproducimos a continuación:

declaración de función

```
double costo_total(int cantidad_par, double precio_par);
```

La declaración de función nos dice todo lo que necesitamos para escribir una llamada a la función. Nos dice el nombre de ésta, en este caso `costo_total`; cuántos argumentos necesita la función y de qué tipo deben ser; en este caso, la función `costo_total` toma dos argumentos, el primero de tipo `int` y el segundo de tipo `double`. Los identificadores `cantidad_par` y `precio_par` se conocen como *parámetros formales*. Un **parámetro formal** se utiliza como un tipo de espacio en blanco, o como un marcador de posición el cual representa los argumentos. Cuando escribimos una declaración de función no sabemos cuáles serán los argumentos, por lo que usamos los parámetros formales en lugar de los argumentos. Los nombres de dichos parámetros pueden ser cualesquier identificadores válidos, pero mientras tanto finalizaremos los nombres de nuestros parámetros formales con `_par`, para que nos sea más sencillo distinguirlos de otros elementos de un programa. Observe que una declaración de función finaliza con un punto y coma.

parámetro formal

La primera palabra de una declaración de función especifica el tipo del valor devuelto por la función. Entonces, en el caso de la función `costo_total`, el tipo del valor devuelto es *double*.

tipo del valor devuelto

Como puede ver, la llamada de función que aparece en el cuadro 3.3 satisface todos los requerimientos dados por su declaración de función. La llamada de función está en la siguiente línea:

```
factura = costo_total(cantidad, precio);
```

La llamada de función es la expresión del lado derecho del signo igual. El nombre de la función es `costo_total`, y tiene dos argumentos: el primero es de tipo `int` y el segundo de tipo `double`, y debido a que la variable factura es de tipo `double`, parece que la función devuelve un valor de tipo `double` (y en realidad así es). Todo ese detalle lo determina la declaración de función.

Al compilador no le importa si hay un comentario en la declaración de función, pero recomendamos siempre incluir un comentario que explique qué valor devuelve la función.

comentario de declaración de funciones

Al final del cuadro 3.3 aparece la definición de función. Una **definición de función** describe cómo es que la función calcula el valor que devuelve. Si considera a una función como un pequeño programa dentro de su programa, entonces la definición de función es como el código de este pequeño programa. De hecho, la sintaxis de la definición de una función es muy parecida a la de la parte `main` de un programa. Una definición de función consiste en un *encabezado de función*, seguido por un *cuerpo de función*. El **encabezado de función** se escribe de la misma manera que la declaración de función, excepto que el encabezado no tiene un punto y coma al final. Esto hace al encabezado un poco repetitivo, pero no hay problema.

definición de función

encabezado de función

CUADRO 3.3 *Una definición de función*

```
#include <iostream>

using namespace std;

double costo_total(int cantidad_par, double precio_par);
//Calcula el costo total, incluido i.v.a. del 5%,
//en la compra de cantidad_par artículos que cuestan precio_par cada uno.
 int main()
 {
    double precio, factura;
    int cantidad;

    cout << "Escriba el numero de articulos adquiridos: ";
    cin >> cantidad;
    cout << "Escriba el precio por articulo $";
    cin >> precio;

    factura = costo_total(cantidad, precio);

   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(2);
    cout << cantidad << " articulos a "
         << "$" << precio << " cada uno.\n"
         << "La factura total, con i.v.a., es de $" << factura
         <<endl;
    return 0;
 }
 double costo_total(int cantidad_par, double precio_par)
 {
    const double TASA_IVA = 0.05; //5% de I.V.A.
    double subtotal;

    subtotal = precio_par * cantidad_par;
    return (subtotal + subtotal*TASA_IVA);
 }
```

declaración de función

llamada de función

encabezado de función

cuerpo de la función

definición de función

Diálogo de ejemplo

```
Escriba el numero de articulos adquiridos: 2
Escriba el precio por articulo: $10.10
2 articulos a $10.10 cada uno.
La factura total, con i.v.a., es de $21.21
```

Declaración de función

Una **declaración de función** nos dice todo lo que necesitamos saber para escribir una llamada de función. Es necesario que la declaración de función aparezca en el código antes de la llamada de función cuya definición aún no ha aparecido. Las declaraciones de función normalmente se colocan antes de la parte `main` del programa.

Sintaxis

Tipo_Devuelto Nombre_de_Funcion(*Lista_de_Parametros*); *No olvide este punto y coma.*
Comentario_de_la_declaracion_de_funcion

donde la *Lista_de_Parametros* es una lista de parámetros separada por comas:

Tipo_1 Parametro_Formal_1, Tipo_2 Parametro_Formal_2, . . .
. . ., Ultimo_Tipo Ultimo_Parametro_Formal

Ejemplo

```
double peso_total(int numero, double peso_de_uno);
// Devuelve el peso total del número de elementos de
// cada uno de los peso_de_uno.
```

Aunque la declaración de función nos indica todo lo que necesitamos saber para escribir una llamada de función, no nos dice qué valor devolverá. El valor devuelto es determinado por las instrucciones del *cuerpo de la función*. El **cuerpo de la función** sigue al encabezado de la función y completa la definición de la función. El cuerpo de la función consiste en declaraciones e instrucciones ejecutables encerradas entre un par de llaves. Por lo tanto, el cuerpo de la función es parecido al cuerpo de la parte `main` de un programa. Cuando se invoca una función, los valores de los argumentos se insertan en lugar de los parámetros formales y después se ejecutan las instrucciones del cuerpo del programa. El valor devuelto por la función se determina cuando la función ejecuta una instrucción `return`. (En la siguiente sección explicaremos los detalles de esta "inserción").

cuerpo de una función

Una **instrucción** `return` consiste en la palabra clave `return` seguida por una expresión. La definición de función del cuadro 3.3 contiene la siguiente instrucción *return*:

instrucción `return`

```
return (subtotal + subtotal*TASA_IVA);
```

Cuando se ejecuta esta instrucción, el valor de la siguiente expresión se devuelve como el valor de la llamada de función:

```
(subtotal + subtotal*TASA_IVA)
```

Los paréntesis no son necesarios. El programa se ejecutaría exactamente igual si la instrucción `return` se escribiera de la siguiente manera:

```
return subtotal + subtotal*TASA_IVA;
```

Sin embargo, en expresiones más largas los paréntesis facilitan la lectura de la instrucción `return`. Por consistencia, algunos programadores utilizan estos paréntesis incluso en expresiones sencillas. En la definición de función del cuadro 3.3 no hay más instrucciones después de la instrucción `return`, pero si las hubiera, no se ejecutarían. Cuando se ejecuta una instrucción `return`, la llamada de función termina.

anatomía de una llamada de función

Veamos exactamente qué ocurre cuando se ejecuta la siguiente llamada de función en el programa que aparece en el cuadro 3.3:

```
factura = costo_total(cantidad, precio);
```

Primero, los valores de los argumentos `cantidad` y `precio` se insertan en lugar de los parámetros formales; es decir, estos valores se sustituyen por `cantidad_par` y `precio_par`. En el ejemplo de diálogo, `cantidad` recibe el valor de `2` y `precio` recibe el valor de `10.10`. Por lo tanto, `2` y `10.10` se sustituyen por `cantidad_par` y `precio_par`, respectivamente.

llamada por valor

Este proceso de sustitución se conoce como **mecanismo de llamada por valor**, y los parámetros formales con frecuencia se conocen como **parámetros formales de llamada por valor**, o simplemente como **parámetros de llamada por valor**. Hay tres puntos que debe notar con respecto a este proceso de sustitución:

1. Los valores de los argumentos se insertan en lugar de los parámetros formales. Si los argumentos son variables, se insertan los valores de estas variables y no las variables mismas.
2. El primer argumento sustituye al primer parámetro formal, el segundo argumento sustituye al segundo parámetro formal y así sucesivamente.
3. Cuando un argumento se inserta en lugar de un parámetro formal (por ejemplo cuando `2` sustituye a `cantidad_par`), el argumento sustituye a todos los ejemplares del parámetro formal que ocurren en el cuerpo de la función (por ejemplo, `2` se inserta cada vez que `cantidad_par` aparece en el cuerpo de la función).

En el cuadro 3.4 se describe con detalle el proceso de llamada de función que se muestra en el cuadro 3.3.

Forma alterna para declarar funciones

No es necesario que al declarar una función se listen los nombres de los parámetros formales. Las dos siguientes declaraciones de funciones son equivalentes:

```
double costo_total(int cantidad_par, double precio_par);
```

y

```
double costo_total(int, double);
```

Nosotros siempre utilizaremos la primera forma, ya que podemos hacer referencia a los parámetros formales en el comentario que acompaña a la declaración de la función. Sin embargo, con frecuencia verá la segunda forma en los manuales que describen funciones.[1]

Esta forma alterna aplica sólo a declaración de funciones. *Los encabezados de función siempre deben listar los nombres de los parámetros formales.*

[1] Lo único que C++ necesita para que el programa pueda enlazarse con la biblioteca o con nuestra función es el nombre de la función y la secuencia de tipos de los parámetros formales. Los nombres de dichos parámetros sólo son importantes para la definición de la función. Sin embargo, los programas deben comunicarse con los programadores además de con los compiladores. Para entender una función, a menudo resulta muy útil usar el nombre que el programador da a los datos de la función.

CUADRO 3.4 *Detalles de una llamada de función (parte 1 de 2)*

Anatomía de la llamada de función del cuadro 3.3

0 Antes de invocarse la función, las variables `cantidad` y `precio` reciben los valores `2` y `10.10` mediante instrucciones `cin` (como se aprecia en el diálogo de ejemplo del cuadro 3.3).

1 La siguiente instrucción que incluye una llamada de función comienza a ejecutarse:

```
factura = costo_total(cantidad, precio);
```

2 El valor de `cantidad` (que es `2`) se inserta en lugar de `cantidad_par`, y el valor de `precio` (que es `10.10`) se inserta en lugar de `precio_par`:

insertar valor de `cantidad`

```
double costo_total(int cantidad_par, double precio_par)
{
    const double TASA_IVA = 0.05; //5% de I.V.A.
    double subtotal;

    subtotal = precio_par * cantidad_par;
    return (subtotal + subtotal*TASA_IVA);
}
```

insertar valor de `precio`

para producir lo siguiente:

```
double costo_total(int 2, double 10.10)
{
      const double TASA_IVA = 0.05; //5% de I.V.A.
      double subtotal;

      subtotal = 10.10 * 2;
      return (subtotal + subtotal*TASA_IVA);
}
```

CUADRO 3.4 *Detalles de una llamada de función (parte 2 de 2)*

Anatomía de la llamada de función del cuadro 3.3 *(conclusión)*

3 Se ejecuta el cuerpo de la función; es decir, se ejecuta lo siguiente:

```
{
   const double TASA_IVA = 0.05; //5% de I.V.A.
   double subtotal;

   subtotal = 10.10 * 2;
   return (subtotal + subtotal*TASA_IVA);
}
```

4 Cuando se ejecuta la instrucción `return`, el valor de la expresión que sigue a la palabra `return` es el valor devuelto por la función. En este caso, cuando se ejecuta

```
return (subtotal + subtotal*TASA_IVA);
```

el valor de (`subtotal + subtotal*TASA_IVA`), que es `21.21`, es devuelto por la llamada de función

```
costo_total(cantidad, precio)
```

y así el valor de `factura` (que está a la izquierda del signo igual) se hace igual a `21.21` cuando por fin termina de ejecutarse la instrucción:

```
factura = costo_total(cantidad, precio);
```

Una función es como un pequeño programa

Para comprender las funciones, mantenga en mente los siguientes tres puntos:

- Una definición de función es como un pequeño programa, e invocar a la función es lo mismo que ejecutar este "pequeño programa".
- Una función emplea parámetros formales en lugar de `cin` para obtener sus entradas. Los argumentos de la función son las entradas, y se insertan en lugar de los parámetros formales.
- Una función (del tipo que explicamos en este capítulo) normalmente no envía ninguna salida hacia la pantalla, pero sí envía una especie de "salida" hacia el programa. La función devuelve un valor, que es como la "salida" de la función. La función emplea una instrucción `return`, en lugar de una instrucción `cout` para dicha salida.

RIESGO *Argumentos en el orden erróneo*

Cuando se llama a una función, la computadora sustituye el primer argumento por el primer parámetro formal, el segundo argumento por el segundo parámetro formal, y así sucesivamente. No se verifica si la sustitución es razonable. Si confundimos el orden de los argumentos en una llamada de función, el programa no hará lo que queremos. Para ver qué puede suceder, consideremos el programa del cuadro 3.5. El programador que escribió ese programa invirtió por descuido el orden de los argumentos en la llamada de la función `calif`. La llamada de función debió ser

```
calif_letra = calif(puntos, minimo_para_aprobar);
```

Éste es el único error en el programa. Incluso, algunos estudiantes desventurados han reprobado el curso injustamente, a causa de este descuido. La función `calif` es tan sencilla que podríamos esperar que el programador descubrirá su error al probar el programa. Sin embargo, si `calif` fuera una función más complicada, el error podría pasar fácilmente inadvertido.

Si el tipo de un argumento no coincide con el parámetro formal, entonces el compilador puede enviarle un mensaje de advertencia. Por desgracia, no todos los compiladores le darán dichos mensajes. Además, en una situación como la del cuadro 3.5 ningún compilador resentirá el orden de los argumentos, porque los tipos de los argumentos de la función coincidirán con los tipos de los parámetros formales, independientemente del orden en que se encuentren los argumentos.

Definición de función-resumen de sintaxis

La declaración de función normalmente se coloca antes de la parte `main` del programa (o como veremos más adelante, en un archivo aparte). El cuadro 3.6 le muestra un resumen de la sintaxis para la definición y declaración de una función. En realidad existe un poco más de libertad de lo que el cuadro indica. Las declaraciones y las instrucciones ejecutables de la definición de función pueden intercalarse siempre y cuando cada variable se declare antes de usarse. Las reglas respecto a intercalar declaraciones e instrucciones ejecutables en una definición de función son las mismas que las que aplican para la parte `main` de un programa. Sin embargo, a menos que tengamos una razón para hacerlo de otra manera, es mejor colocar primero las declaraciones, como se indica en el cuadro 3.6.

instrucción `return`

Debido a que una función no devuelve un valor, a menos que ejecute una instrucción *`return`*, la función debe contener una o más instrucciones *`return`* en su cuerpo. Una definición de función puede contener más de una instrucción *`return`*. Por ejemplo, el cuerpo del código puede contener una instrucción *`if-else`*, y cada bifurcación de la instrucción *`if-else`* puede contener una instrucción *`return`* diferente, como muestra el cuadro 3.5.

espaciado y cambios de línea

El compilador acepta cualquier patrón razonable de espacios y saltos de línea en una definición de función. Sin embargo, recomendamos utilizar las mismas reglas de sangrías y organización en una definición de función que en la parte `main` del programa. En particular, observe la ubicación de las llaves `{}` en nuestras definiciones de función y del cuadro 3.6. Las llaves inicial y final que delimitan el cuerpo de la función se colocan cada una en una línea aparte. Esto destaca el cuerpo de la función.

CUADRO 3.5 *Argumentos en el orden incorrecto*

```
//Determina la calificación del usuario. Las calificaciones
//son Aprobar o Reprobar.
#include <iostream>
using namespace std;

char calif(int recibido_par, int puntos_min_par);
//Devuelve 'A' si aprueba, es decir, si recibido_par es
//puntos_min_par o mayor. Si no, devuelve 'R' (reprobado).
int main()
{
    int puntos, minimo_para_aprobar;
    char calif_letra;

    cout << "Escribe tus puntos"
         << " y los puntos minimos que necesitas para aprobar:\n";
    cin >> puntos >> minimo_para_aprobar;

    calif_letra = calif(minimo_para_aprobar, puntos);

    cout << "Recibiste " << puntos << endl
         << "El mínimo para aprobar es " << minimo_para_aprobar << endl;
    if (calif_letra == 'A')
        cout << "Aprobaste. ¡Te felicito!\n";
    else
        cout << "Lo siento. Reprobaste.\n";

    cout << "Se asentará " << calif_letra
         << " en tu expediente.\n";
    return 0;
}
char calif(int recibido_par, int puntos_min_par)
{
    if (recibido_par >= puntos_min_par)
       return 'A';
    else
       return 'R'
}
```

Diálogo de ejemplo

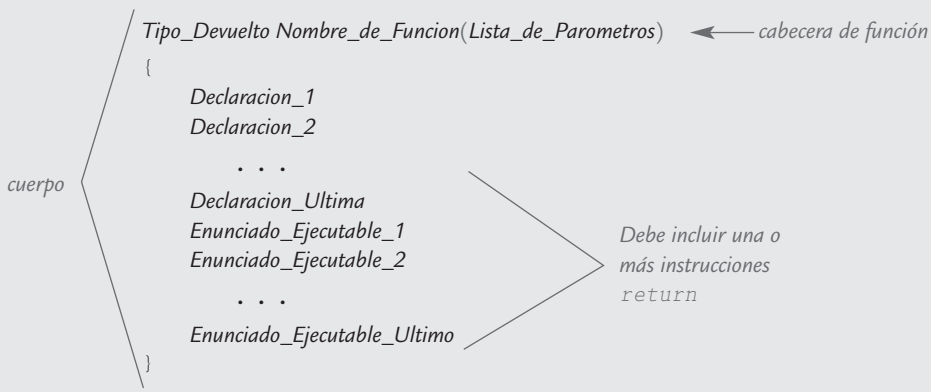
```
Escribe tus puntos y los puntos minimos que necesitas aprobar:
98 60
Recibiste 98 puntos
El minimo para aprobar es 60
Lo siento. Reprobaste.
Se asentara R en tu expediente
```

CUADRO 3.6 *Sintaxis para una función que devuelve un valor*

Declaración de función

Tipo_Devuelto Nombre_de_Funcion(*Lista_de_Parametros*);
Comentario_de_Prototipo

Definición de función



Más sobre la colocación de definiciones de funciones

Ya hemos explicado en dónde se colocan normalmente las definiciones y declaraciones de funciones. Bajo circunstancias normales, éstas son las mejores ubicaciones para declarar y definir funciones. Sin embargo, el compilador aceptará programas con definiciones y declaraciones en otras partes. Un planteamiento más preciso de las reglas es el siguiente: toda llamada de función debe estar precedida por una declaración o por la definición de esa función. Por ejemplo, si colocamos todas las definiciones de funciones antes de la parte `main` del programa, entonces no necesitaremos incluir ninguna declaración de función. Conocer esta regla general le ayudará a comprender programas en C++ que vea en otros libros, pero le recomendamos seguir el ejemplo de los programas de este libro. El estilo que utilizamos nos prepara para aprender a construir nuestras propias bibliotecas de funciones, y es el estilo que utiliza la mayoría de los programadores en C++.

Ejercicios de AUTOEVALUACIÓN

5. ¿Cuál es la salida que produce el siguiente programa?

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

```
char misterio(int primer_par, int segundo_par);
int main()
{
    cout << misterio(10, 9) << "ow\n";
    return 0;
}
char misterio(int primer_par, int segundo_par)
{
    if(primer_par >= segundo_par)
       return 'W';
    else
       return 'H';
}
```

6. Escriba una declaración y una definición para una función que recibe tres argumentos, todos de tipo `int`, y que devuelve la suma de sus tres argumentos.

7. Escriba una declaración y una definición para una función que recibe un argumento de tipo `int` y un argumento de tipo `double`, y que devuelva un valor de tipo `double` que sea el promedio de los dos argumentos.

8. Escriba una declaración y una definición para una función que recibe un argumento de tipo `double`. La función devuelve el valor de carácter `'P'`, si su argumento es positivo, y devuelve `'N'`, si su argumento es cero o negativo.

9. Describa cuidadosamente el mecanismo del parámetro de llamada por valor.

10. Liste las similitudes y diferencias entre utilizar un función predefinida (es decir, de una biblioteca) y una definida por el usuario.

3.4 Abstracción de procedimientos

La causa está oculta, pero todos conocen el resultado.

Ovidio, *Metamorfosis IV*

La analogía de la caja negra

Una persona que utiliza un programa no necesita saber los detalles sobre cómo está codificado el programa. Imagine lo pesada que sería su vida si tuviera que conocer y recordar el código del compilador que utiliza. Un programa tiene una tarea que hacer, como compilar su programa o verificar la ortografía de un trabajo escolar. Necesitamos saber *cuál* es la tarea del programa, para poder utilizarlo, pero no necesitamos (o no deberíamos necesitar) conocer la manera en que el programa hace su trabajo. Una función es como un pequeño programa y debe utilizarse de manera similar. Un programador que utiliza una función en un programa debe saber *qué hace* la función (como calcular una raíz cuadrada o convertir una temperatura de grados Fahrenheit a Celsius), pero no necesita saber *la ma-*

nera en que la función realiza la tarea. Solemos decir que esto equivale a tratar a la función como una *caja negra*.

caja negra

Llamar a algo una **caja negra** es un modo de hablar, buscando transmitir la imagen de un dispositivo físico que sabemos utilizar, pero cuyo método de operación es un misterio, ya que éste se encuentra encerrado en una caja negra de la que no podemos ver su interior (ni podemos abrirla). Si una función está bien diseñada, el programador puede utilizarla como si fuera una caja negra. Todos los programadores deben saber que si colocan los argumentos apropiados en la caja negra, de ésta saldrá un valor apropiado. Diseñar una función para que pueda utilizarse como una caja negra en ocasiones se conoce como **ocultamiento de información** pues el programador actúa como si el cuerpo de la función estuviera oculto.

ocultamiento de información

El cuadro 3.7 contiene la declaración de una función y dos definiciones distintas para una función llamada `saldo_nuevo`. Como explica el comentario de la declaración de la

CUADRO 3.7 *Definiciones que son equivalentes a una caja negra*

Declaración de función

```
double saldo_nuevo(double saldo_par, double tasa_par);
//Devuelve el saldo de una cuenta bancaria después de
//acumular interés simple. El parámetro formal saldo_par es el
//saldo anterior. El parámetro formal tasa_par es la tasa de interés.
//Por ejemplo, si tasa_par es 5.0, la tasa de interés es del 5%
//y saldo_nuevo(100, 5.0) devuelve 105.00
```

Definición 1

```
double saldo_nuevo(double saldo_par, double tasa_par)
{
    double fraccion_intereses, intereses;
    fraccion_intereses = tasa_par/100;
    intereses = fraccion_intereses*saldo_par;
    return (saldo_par + intereses);
}
```

Definición 2

```
double saldo_nuevo(double saldo_par, double tasa_par)
{
    double fraccion_intereses, saldo_actualizado;
    fraccion_intereses = tasa_par/100;
    saldo_actualizado = saldo_par*(1 + fraccion_intereses);
    return saldo_actualizado;
}
```

función, dicha función calcula el nuevo saldo de una cuenta bancaria cuando se le agrega el interés simple. Por ejemplo, si una cuenta inicia con $100, y se le agrega un interés del 4.5%, entonces el nuevo saldo es $104.50. Entonces, el siguiente código cambiará el valor de `fondo_vacacional` de `100.00` a `104.50`:

```
fondo_vacacional = 100.00;
fondo_vacacional = saldo_ nuevo (fondo_vacacional, 4.5);
```

No tiene importancia cuál de las implementaciones de `saldo_nuevo` que aparecen en el cuadro 3.7 utilice el programador. Ambas definiciones producen funciones que devuelven exactamente el mismo valor. También podemos colocar una caja negra sobre el cuerpo de la definición de función, para que el programador no sepa qué implementación se está utilizando. Lo único que el programador necesita hacer para usar la función `saldo_nuevo` es leer la declaración de función y el comentario que la acompaña.

abstracción de procedimientos

Escribir y utilizar funciones como si fueran cajas negras también se conoce como **abstracción de procedimientos**. Cuando se programa en C++ podría tener más sentido llamarlo *abstracción funcional*. Sin embargo, *procedimiento* es un término más general que *función*. Los expertos en computación utilizan el término procedimiento para todos los conjuntos de instrucciones "tipo función", y es por ello que utilizan el término *abstracción de procedimientos*. El término abstracción intenta transmitir la idea de que, cuando se utiliza una función como caja negra, estamos ocultando los detalles del código contenido en el cuerpo de la función. Podemos llamar a esta técnica *principio de la caja negra* o *principio de la abstracción de procedimientos* u *ocultamiento de información*. Los tres términos significan lo mismo. Sea cual sea la manera en que llamemos a este principio, lo importante es que debemos utilizarlo al diseñar y escribir sus definiciones de funciones.

Abstracción de procedimientos

Cuando el principio de **abstracción de procedimientos** se aplica a la definición de una función, significa que la función debe escribirse de modo que pueda utilizarse como una **caja negra**. Esto significa que el programador que emplea la función no necesita examinar el cuerpo de la definición de la función para ver cómo trabaja. La declaración de función y el comentario que la acompaña debe ser todo lo que el programador necesite para saber cómo utilizarla. Para garantizar que nuestras definiciones de función tengan esta importante propiedad, debemos cumplir con las siguientes reglas:

Cómo escribir una definición de función de caja negra (que devuelve un valor)

- El comentario de la declaración de función debe decirle al programador todas y cada una de las condiciones que necesitan los argumentos de la función, y debe describir el valor que es devuelto por la función cuando sea llamada con estos argumentos.
- Todas las variables utilizadas en el cuerpo de la función deben declararse en el cuerpo de ésta. (Los parámetros formales no necesitan declararse, ya que éstos se listan en la declaración de la función.)

TIP DE PROGRAMACIÓN

Cómo elegir los nombres de parámetros formales

El principio de abstracción de procedimientos dice que las funciones deben ser módulos autosuficientes que se diseñan aparte del resto del programa. En grandes proyectos de programación es probable que varios programadores estén asignados para escribir cada función. El programador debe elegir los nombres más significativos que pueda encontrar para los parámetros formales. Los argumentos que se insertarán en lugar de los parámetros formales bien pueden ser variables en la parte `main` del programa. A estas variables también se les debe dar nombres significativos, y en muchos casos los elige otra persona, no el programador que escribe la definición de función. Esto hace probable que alguno o todos los argumentos tengan los mismos nombres que algunos de los parámetros formales. Esto es perfectamente aceptable. Independientemente de los nombres elegidos para las variables que se utilizarán como argumentos, estos nombres no producirán confusión alguna con los nombres usados para los parámetros formales. Después de todo, las funciones sólo utilizarán los valores de los argumentos. Cuando utiliza una variable como un argumento de función, ésta sólo toma el valor de la variable e ignora su nombre.

Ahora que sabemos que contamos con completa libertad para elegir los nombres de los parámetros formales, dejaremos de añadir "`_par`" al final de los nombres de parámetros formales. Por ejemplo, en el cuadro 3.8 reescribimos la definición de la función `costo_total` del cuadro 3.3, por lo que los parámetros formales son llamados cantidad y precio, en lugar de `cantidad_par` y `precio_par`. Si sustituimos la declaración y la definición de la función `costo_total` que aparece en el cuadro 3.3 con las versiones del cuadro 3.8, entonces el programa se desempeñará exactamente de la misma manera, aunque habrán parámetros formales llamados `cantidad` y `precio`, y variables en la parte `main` del programa, que también se llaman `cantidad` y `precio`.

CASO DE ESTUDIO Compra de pizza

El "ahorro" por el tamaño grande de un elemento no siempre resulta una mejor compra que si se adquiere uno de menor tamaño. Esto es particularmente cierto cuando se compran pizzas. Los tamaños de las pizzas están dados por el diámetro de la pizza expresado en pulgadas. Sin embargo, la cantidad de pizza se determina por el área de la pizza, y ésta no es proporcional al diámetro. La mayoría de la gente no puede fácilmente estimar la diferencia en área entre una pizza de 10 pulgadas y una de 12, por lo que no fácilmente puede identificar qué tamaño es mejor comprar, es decir, qué tamaño tiene el menor precio por pulgada cuadrada. En este caso de estudio diseñaremos un programa que compara dos tamaños de pizza, para determinar cuál es la mejor compra.

Definición del problema

La especificación precisa de la entrada y salida del programa es la siguiente:

Entrada

La entrada consistirá en el diámetro en pulgadas, y el precio de cada uno de los tamaños de las pizzas.

CUADRO 3.8 *Nombres más sencillos para los parámetros formales*

Declaración de función

```
double costo_total(int cantidad, double precio);
//Calcula el costo total, incluido i.v.a. del 5%, en
//la compra de cantidad artículos que cuestan precio cada uno.
```

Definición de función

```
double costo_total(int cantidad, double precio)
{
    const double TASA_IVA = 0.05; //5% de I.V.A.
    double subtotal;

    subtotal = precio * cantidad;
    return (subtotal + subtotal*TASA_IVA);
}
```

Salida

La salida dará el costo por pulgada cuadrada para cada uno de los tamaños de pizza, e indicará cuál es la mejor compra, es decir, cuál tiene el menor costo por pulgada cuadrada. (Si tienen el mismo costo por pulgada cuadrada, consideraremos el más pequeño como la mejor compra.)

Análisis del problema

Utilizaremos el diseño descendente para dividir la tarea que nuestro programa debe realizar, en las siguientes subtareas:

Subtarea 1: Obtener los datos de entrada para las pizzas pequeña y grande.

Subtarea 2: Calcular el precio por pulgada cuadrada para la pizza pequeña.

Subtarea 3: Calcular el precio por pulgada cuadrada para la pizza grande.

Subtarea 4: Determinar cuál es la mejor compra.

Subtarea 5: Desplegar los resultados.

subtareas 2 y 3

Observe las subtareas 2 y 3, ya que tienen dos importantes propiedades:

1. Son exactamente la misma tarea. La única diferencia es que utilizan diferentes datos para realizar el cálculo. Lo único que cambia entre la subtarea 2 y la 3 es el tamaño de la pizza y su precio.
2. Los resultados de la subtarea 2 y 3 son valores individuales, el precio por pulgada cuadrada de la pizza.

cuándo definir una función

Siempre que una subtarea recibe algunos valores, digamos algunos números, y devuelva un solo valor, es natural implementar la subtarea como una función. Siempre que dos o más subtareas realicen el mismo cálculo, pueden implementarse como la misma función, la cual se invoca con diferentes argumentos cada vez que se utilice. Por lo tanto decidimos utilizar una función llamada `preciounitario` para calcular el precio por pulgada cuadrada de una pizza. La declaración y el comentario explicativo para esta función serán los siguientes:

```
double preciounitario(int diametro, double precio);
//Devuelve el precio por pulgada cuadrada de una pizza.
//El parámetro formal llamado diametro, es el diámetro de la pizza
//en pulgadas. El parámetro formal llamado precio es el precio de
//la pizza.
```

Diseño del algoritmo

subtarea 1

La subtarea 1 es directa. El programa simplemente solicitará los valores de entrada y los almacenará en cuatro variables, a las que llamaremos `diametro_pequeño`, `diametro_grande`, `precio_pequeño` y `precio_grande`.

subtareas 4 y 5

La subtarea 4 es rutina. Para determinar cuál pizza es la mejor compra, simplemente comparamos el costo por pulgada cuadrada de las dos pizzas, utilizando el operador menor que. La subtarea 5 es una salida de rutina de los resultados.

subtareas 2 y 3

Las subtareas 2 y 3 se implementan como llamadas a la función `preciounitario`. Después, diseñamos el algoritmo para esta función. La parte difícil del algoritmo es determinar el área de la pizza. Una vez que conocemos el área, podemos fácilmente determinar el precio por pulgada cuadrada, utilizando una división de la siguiente manera:

```
precio/area
```

donde `area` es una variable que contiene el área de la pizza. Esta expresión será el valor devuelto por la función `preciounitario`. Pero todavía necesitamos formular un método para calcular el área de la pizza.

Una pizza es básicamente un círculo (hecho de masa, queso, salsa, etcétera). El área de un círculo (y por lo tanto de una pizza) es πr^2, donde r es el radio del círculo y π es el número llamado "pi", que es aproximadamente igual a 3.14159. El radio es la mitad del diámetro.

El algoritmo para la función preciounitario puede bosquejarse como sigue:

Bosquejo del algoritmo para la función `preciounitario`

1. Calcular el radio de la pizza.
2. Calcular el área de la pizza, utilizando la fórmula πr^2.
3. Devolver el valor de la expresión (`precio/area`).

seudocódigo

Antes de traducir este bosquejo a código de C++, lo plantearemos con un poco más de detalle. Expresaremos en *pseudocódigo* esta versión más detallada de nuestro algoritmo. El **pseudocódigo** es una mezcla de C++ y español ordinario; nos permite precisar nuestro algoritmo, sin tener que preocuparnos por los detalles de la sintaxis de C++. Después es fácil traducir el pseudocódigo en código de C++. En nuestro pseudocódigo, `radio` y `area` serán variables para almacenar los valores indicados por sus nombres.

Pseudocódigo de la función `preciounitario`

```
radio = mitad de diametro;
area = π * radio * radio;
return (precio/area);
```

Esto completa nuestro algoritmo para `preciounitario`. Ahora estamos listos para convertir nuestras soluciones a las subtareas 1 a 5 en un programa completo de C++.

Codificación

Codificar la subtarea 1 es rutina, por lo que mejor consideraremos las subtareas 2 y 3. Nuestro programa puede implementar las subtareas 2 y 3 mediante las siguientes dos llamadas a la función `preciounitario`:

```
preciounit_chica = preciounitario(diametro_chica, precio_chica);
preciounit_grande = preciounitario(diametro_grande, precio_grande);
```

donde `preciounit_chica` y `preciounit_grande` son dos variables de tipo *`double`*. Uno de los beneficios de una definición de función es que en su programa puede tener múltiples llamadas a la función. Esto nos ahorra la tarea de repetir el mismo (o casi el mismo) código. Pero aún nos falta escribir el código para la función `preciounitario`.

Cuando traducimos nuestro pseudocódigo en código C++, obtenemos lo siguiente para el cuerpo de la función `preciounitario`:

```
{//Primer borrador del cuerpo de la función preciounitario
    const double PI = 3.14159;
    double radio, area;

    radio = diametro/2;
    area = PI * radio * radio;
    return (precio/area);
}
```

Observe que convertimos a PI en una constante con nombre, utilizando el modificador *`const`*. Además observe la siguiente línea del código:

```
radio = diametro/2;
```

Ésta es sólo una simple división entre dos, y tal vez piense que nada podría ser más de rutina. Sin embargo, como está escrita, esta línea contiene un grave error. Nosotros queremos que la división produzca el radio de la pizza, incluyendo cualquier fracción. Por ejemplo, si consideramos comprar la especial "mala suerte", que es una pizza de 13 pulgadas, entonces el radio es 6.5 pulgadas. Pero la variable `diametro` es de tipo *`int`*. La constante `2` también es de tipo *`int`*. Por lo tanto, como vimos en el capítulo 2, esta línea realizaría una división entera y calcularíamos el radio de `13/2` como `6`, en lugar de calcular el valor correcto de `6.5`, con lo que despreciaríamos media pulgada del radio de la pizza. Con toda certeza esto pasaría inadvertido, pero el resultado podría ser que millones de suscriptores de la Unión de Consumidores de Pizza desperdiciaran su dinero comprando el tamaño de pizza equivocado. No es probable que esto ocasionara una recesión importante a nivel mundial, pero el programa estaría fallando en su tarea de ayudar a los consumidores a realizar la mejor compra. En un programa más importante, el resultado de un error tan simple como éste podría ser desastroso.

¿Cómo arreglamos este error? Queremos que la división entre dos sea una división normal que incluya cualquier parte fraccionaria en la respuesta. Esa forma de división requiere que al menos uno de los argumentos del operador de división / sea de tipo *double*. Podemos utilizar la conversión de tipo para convertir la constante 2 en un valor de tipo *double*. Recuerde que *static_cast<double>(2)*, que se conoce como una conversión de tipo, convierte el valor *int* de 2 en un valor de tipo *double*. Entonces, si remplazamos 2 por *static_cast<double>(2)*, que cambiará el segundo argumento de la división de tipo *int* a tipo *double*, se producirá el resultado que queremos. La instrucción de asignación queda como:

static_cast <double>

```
radio = diametro/static_cast<double>(2);
```

El código corregido para la definición de la función `preciounitario`, junto con el resto del programa aparece en el cuadro 3.9.

La conversión de tipo *static_cast<double>*`(2)` devuelve el valor de `2.0`, por lo que pudimos haber usado la constante `2.0` en lugar de *static_cast<double>*`(2)`. De cualquier manera, la función `preciounitario` devolverá el mismo valor. Sin embargo, utilizando `static_cast<double>(2)` hacemos muy evidente que queremos la versión de la división que incluye la parte fraccionaria en la respuesta. Si hubiésemos utilizado `2.0`, entonces cuando revisáramos o copiáramos el código fácilmente podríamos equivocarnos cambiando `2.0` por `2`, y se produciría un problema.

Necesitamos hacer una observación más con respecto a la codificación de nuestro programa. Como puede ver en el cuadro 3.9, cuando codificamos las tareas 4 y 5 combinamos estas dos tareas en una sola sección de código consistente en una secuencia de instrucciones *cout*, seguida por una instrucción *if-else*. Cuando dos tareas son muy sencillas y están muy relacionadas, algunas veces tiene sentido combinarlas en una sola tarea.

Prueba del programa

Sólo porque un programa se compila y produce respuestas que parecen correctas, no significa que el programa sea correcto. Con el fin de incrementar la confianza en nuestro programa deberemos probarlo con algunos valores de entrada, cuya respuesta conozcamos por algún otro medio, como hacer las cuentas con lápiz y papel, o utilizando una calculadora manual. Por ejemplo, no tiene sentido comprar una pizza de dos pulgadas, pero puede utilizarse como un caso de prueba sencillo para este programa. Éste es un caso sencillo de prueba porque es fácil calcular la respuesta a mano. Calculemos el costo por pulgada cuadrada de una pizza de 2 pulgadas que cuesta $3.14. Debido a que el diámetro es de dos pulgadas, el radio es una pulgada. El área de la pizza con radio uno es `3.14159*12`, lo que resulta `3.14159`. Si dividimos esto entre el precio de $3.14, encontramos que el precio por pulgada cuadrada es `3.14/3.14159`, que aproximadamente es $1.00. Por supuesto, éste es un tamaño absurdo para una pizza, y un precio absurdo también para una pizza tan pequeña, pero es fácil determinar el valor que devolverá la función `preciounitario` con estos argumentos.

Después de verificar nuestro programa con este caso, podemos tener más confianza en él, pero aún no podemos estar seguros de que sea correcto. Un programa incorrecto algunas veces produce la respuesta correcta, aunque dará respuestas incorrectas con algunas otras entradas. Podríamos haber probado un programa incorrecto en uno de los casos en los que por casualidad da los resultados correctos. Por ejemplo, supongamos que no detectamos el error que descubrimos cuando codificamos la función `preciounitario`. Supongamos que en la siguiente línea erróneamente utilizamos `2` en lugar de *static_cast<double>*`(2)`:

```
radio = diametro/ static_cast<double>(2);
```

CUADRO 3.9 *Compra de pizza (parte 1 de 2)*

```
//Determina cuál de dos tamaños de pizza es la mejor compra.
#include <iostream>
using namespace std;

double preciounitario(int diametro, double precio);
//Devuelve el precio por pulgada cuadrada de una pizza.
//El parámetro formal llamado diametro es el diámetro de la pizza en pulgadas.
//El parámetro formal llamado precio es el precio de la pizza.

int main()
{
    int diametro_chica, diametro_grande;
    double precio_chica, preciounit_chica,
           precio_grande, preciounit_grande;
    cout << "Bienvenido a la Union de Consumidores de Pizza.\n";
    cout << "Escriba el diametro de una pizza chica (en pulg): ";
    cin >> diametro_chica;
    cout << "Escriba el precio de una pizza chica: $";
    cin >> precio_chica;
    cout << "Escriba el diametro de una pizza grande (en pulg): ";
    cin >> diametro_grande;
    cout << "El precio de una pizza grande: $";
    cin >> precio_grande;

    preciounit_chica = preciounitario(diametro_chica, precio_chica);
    preciounit_grande = preciounitario(diametro_grande, precio_grande);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Pizza chica:\n"
         << "Diametro = " << diametro_chica << " pulg\n"
         << "Precio = $" << precio_chica
         << " Por pulgada cuadrada = $" << preciounit_chica << endl
         << "Pizza grande:\n"
         << "Diametro = " << diametro_grande << " pulg\n"
         << "Precio = $" << precio_grande
         << " Por pulgada cuadrada = $" << preciounit_grande << endl;
```

CUADRO 3.9 *Compra de pizza (parte 2 de 2)*

```
    if (preciounit_grande < preciounit_chica)
        cout << "La grande es la mejor compra.\n";
    else
        cout << "La chica es la mejor compra.\n";
    cout<< "Buon Appetito!\n";

    return 0;
}

double preciounitario(int diametro, double precio)
{
    const double PI = 3.14159;
    double radio, area;

    radio = diametro/static_cast<double>(2);
    area = PI * radio * radio;
    return (precio/area);
}
```

Diálogo de ejemplo

```
Bienvenido a la Union de Consumidores de Pizza.
Escriba el diametro de una pizza chica (en pulg): 10
Escriba el precio de una pizza chica: $7.50
Escriba el diametro de una pizza grande (en pulg): 13
Escriba el precio de una pizza grande: $14.75
Pizza chica:
Diametro = 10 pulg
Precio = $7.50 Por pulgada cuadrada= $0.10
Pizza grande:
Diametro = 13 pulg
Precio = $14.75 Por pulgada cuadrada = $0.11
La chica es la mejor compra.
¡Buon Appetito!
```

Por lo tanto, esa línea se lee como sigue:

```
radio = diametro/2;
```

Mientras el diámetro de la pizza sea un número par, como `2`, `8`, `10` o `12`, el programa dará la misma respuesta, si dividimos entre `2` o entre `static_cast<double>(2)`. Es poco probable que se nos ocurriera probar el programa con pizzas de diámetro tanto par como impar. Sin embargo, si probamos el programa con diferentes tamaños de pizza, habrá más posibilidades de que nuestros casos de prueba contengan muestras de los tipos de datos pertinentes.

TIP DE PROGRAMACIÓN

Utilice pseudocódigo

pseudocódigo

Los algoritmos normalmente se expresan en pseudocódigo. El **pseudocódigo** es una mezcla de C++ (o cualquier otro lenguaje de programación que utilice) y el español ordinario (o cualquier otro idioma). El pseudocódigo nos permite establecer nuestro algoritmo de manera precisa, sin tener que preocuparnos por todos los detalles de sintaxis de C++. Cuando el código de C++ para un paso de nuestro algoritmo es obvio, no tiene mucho caso expresarlo en español. Cuando un paso es difícil de expresar en C++, el algoritmo será más claro si se expresa en español. Podemos ver un ejemplo de pseudocódigo en el caso de estudio anterior, en el que expresamos en pseudocódigo nuestro algoritmo para la función `preciounitario`.

Ejercicios de AUTOEVALUACIÓN

11. ¿Cuál es el objetivo del comentario que acompaña a una declaración de función?

12. ¿Cuál es el principio de la abstracción de procedimientos, aplicado a la definición de funciones?

13. ¿Qué significa cuando decimos que el programador que utiliza una función debe ser capaz de tratarla como una caja negra? (*Tip*: esta pregunta está muy relacionada con la pregunta anterior.)

14. Describa cuidadosamente el proceso de prueba de programa.

15. Considere dos posibles definiciones para la función `preciounitario`. Una es la definición que aparece en el cuadro 3.9. La otra definición es la misma, excepto que la conversión de tipo `static_cast<double>(2)` es reemplazada por la constante `2.0`, en otras palabras, la línea

    ```
    radio = diametro/ static_cast<double>(2);
    ```

    es reemplazada por la línea

    ```
    radio = diametro/2.0;
    ```

    ¿Son equivalentes a una caja negra, estas dos definiciones de función?

3.5 Variables locales

Él era un chico de la localidad,
desconocido fuera de su pueblo natal.

Dicho popular

En la última sección recomendamos utilizar las funciones como si fueran cajas negras. Con el objetivo de definir una función para que pudiera usarse como una caja negra, con frecuencia necesitamos proporcionar variables de función que no interfieran con el resto del programa. Estas variables que "pertenecen" a una función se conocen como *variables locales*. En esta sección describiremos las variables locales y le diremos cómo utilizarlas.

La analogía del programa pequeño

Examine nuevamente el programa del cuadro 3.1. Éste contiene una llamada a la función predefinida `sqrt`. Para utilizar dicha función no necesitamos saber nada acerca de los detalles de su definición. En particular, no necesitamos saber cuáles variables se declararon en la definición de `sqrt`. Las funciones que nosotros mismos definimos no son distintas. La declaración de variables en la definición de funciones que usted escribe están separadas, como aquellas definidas para funciones predefinidas. La declaración de variables dentro de una definición de función bien podría ser declaración de variables en otra parte del programa. Si declaramos una variable en una definición de función y después declaramos otra variable con el mismo nombre en la parte `main` del programa (o en el cuerpo de alguna otra definición de función) entonces estas dos variables son diferentes, aunque tengan el mismo nombre. Veamos un programa que tiene una variable en la definición de función con el mismo nombre que otra variable del programa.

El programa del cuadro 3.10 tiene dos variables llamadas `promedio_chich`; una se declara y utiliza en la definición de la función `est_total` y la otra se declara y utiliza en la parte `main` del programa. La variable `promedio_chich` en la definición de `est_total` y en la parte `main` del programa es diferente. Es lo mismo que si la función `est_total` fuera una función predefinida. Las dos variables llamadas `promedio_chich` no interfieren entre sí, más de lo que lo harían dos variables en dos programas completamente diferentes. Cuando se le asigna un valor a la variable `promedio_chich` en la llamada de la función `est_total` no se modifica el valor de la variable que lleva el mismo nombre en la parte `main` del programa. (Los detalles del programa del cuadro 3.10, fuera de esta coincidencia de nombre, se explican en la sección de ejemplo de programación que sigue a ésta.)

local respecto a una función
alcance

Se dice que las variables que se declaran dentro del cuerpo de una definición de función son **locales respecto a esa función**, o que esa función es su **alcance**. Las variables que se definen dentro de la parte `main` del programa se conocen como **locales a la parte** `main` **del programa**, o que la parte `main` del programa es su **alcance**. Hay otros tipos de variables que no son locales a ninguna función o a la parte `main` del programa, pero no utilizaremos tales variables. Todas las variables que usaremos serán locales respecto a una definición de función, o bien respecto a la parte `main` del programa. Cuando decimos que una **variable es local**, sin mencionar una función o la parte `main` del programa, queremos decir que la variable es local a alguna definición de función.

variable local

CUADRO 3.10 *Variables locales (parte 1 de 2)*

```
//Calcula el rendimiento medio de un huerto experimental de chícharos.
#include <iostream>
using namespace std;

double est_total(int min_chich, int max_chich, int num_vainas);
//Devuelve un estimado del número total de chícharos cosechados.
//El parámetro formal num_vainas es el número de vainas.
//Los parámetros formales min_chich y max_chich son los números
//mínimo y máximo de chícharos en una vaina.

int main()
{
    int num_max, num_min, num_vainas;
    double promedio_chich, rendimiento;

    cout << "Escribe el numero minimo y maximo de chicharos en una vaina: ";
    cin >> num_min >> num_max;
    cout << "Escribe el numero de vainas: ";
    cin >> num_vainas;
    cout << "Escribe el peso promedio de un chicharo (en gramos): ";
    cin >> promedio_chich;

    rendimiento = est_total(num_min, num_max, num_vainas) * promedio_chich;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(3);
    cout << "Minimo de chicharos por vaina = " << num_min << endl
         << "Maximo de chicharos por vaina = " << num_max << endl
         << "Numero de vainas = " << num_vainas << endl
         << "Peso promedio de un chicharo = "
         << promedio_chich << " gramos" << endl
         << "Rendimiento medio estimado = " << rendimiento << " gramos"
         <<endl;

    return 0;
}
```

Esta variable llamada `promedio_chich` *es local respecto a la parte* `main` *del programa.*

CUADRO 3.10 *Variables locales (parte 2 de 2)*

```
double est_total(int min_chich, int max_chich, int num_vainas)
{
    double promedio_chich;

    promedio_chich = (max_chich + min_chich)/2.0;
    return (num_vainas * promedio_chich);
}
```

Esta variable llamada `promedio_chich` *es local respecto a la función* `estimar_total`.

Diálogo de ejemplo

```
Escribe el numero minimo y maximo de chicharos en una vaina: 4 6
Escribe el numero de vainas: 10
Escribe el peso promedio de un chicharo (en gramos): 0.5
Minimo de chicharos por vaina = 4
Maximo de chicharos por vaina = 6
Numero de vainas = 10
Peso promedio de un chicharo = 0.500 gramos
Rendimiento medio estimado = 25.000 gramos
```

Variables locales

Se dice que las variables que se declaran dentro del cuerpo de una definición de función son **locales respecto a esa función**, o que esa función es su **alcance**. Se dice que las variables que se declaran dentro de la parte `main` del programa son **locales respecto a la parte** `main`, o que la parte `main` es su **alcance**. Cuando decimos que una variable es una **variable local** sin mencionar a una función o a la parte `main` del programa, queremos decir que la variable es local respecto a alguna definición de función. Si una variable es local respecto a una función, entonces podemos tener otra variable con el mismo nombre declarada en la parte `main` del programa o en alguna otra definición de función, y serán dos variables distintas, aunque tengan el mismo nombre.

EJEMPLO DE PROGRAMACIÓN
Huerto experimental de chícharos

El programa del cuadro 3.10 proporciona un estimado del total producido en un pequeño huerto que se utiliza para cultivar una gran variedad experimental de chícharos. La función `est_total` devuelve un estimado del número total de chícharos cosechados. La función `est_total` toma tres argumentos. Uno es el número de vainas de chícharos que se

cosecharon. Los otros dos argumentos se utilizan para estimar el número promedio de chícharos en una vaina. Diferentes vainas contienen diferentes cantidades de chícharos, por lo que los otros dos argumentos de la función son los números mínimo y máximo de chícharos encontrados en cualquiera de las vainas. La función `est_total` promedia estos dos números y utiliza dicho promedio como una estimación del número promedio de chícharos en una vaina.

Constantes y variables globales

Como vimos en el capítulo 2, podemos y debemos nombrar los valores constantes, utilizando el modificador *const*. Por ejemplo, en el cuadro 3.9 utilizamos la siguiente declaración para dar el nombre PI a la constante 3.14159:

```
const double PI = 3.14159;
```

En el cuadro 3.3 utilizamos el modificador *const* para nombrar a la tasa de impuesto de ventas con la siguiente declaración:

```
const double TASA_IVA = 0.05; //5% de impuesto de ventas
```

Al igual que con nuestras declaraciones de variables, colocamos estas declaraciones para dar nombre a constantes dentro del cuerpo de las funciones que las usan. Esto salió bien porque cada constante con nombre era utilizada sólo por una función. Sin embargo, puede ocurrir que más de una función utilice una constante con nombre. En ese caso podemos colocar la declaración para nombrar una constante al principio del programa, fuera del cuerpo de todas las funciones y fuera del cuerpo de la parte `main` de su programa. Se dice que la constante con nombre es una **constante global con nombre**, y puede utilizarse en cualquier definición de función que siga a la declaración de la constante.

constantes globales con nombre

El cuadro 3.11 muestra un programa con un ejemplo de una constante global con nombre. El programa pide un radio y después calcula tanto el área de un círculo como el volumen de una esfera con ese radio. El programador que escribió ese programa investigó las fórmulas para calcular dichas cantidades y encontró lo siguiente:

$$\textit{área} \times \pi \times (\textit{radio})^2$$

$$\textit{volumen} = (4/3) \times \pi \times (\textit{radio})^3$$

Ambas fórmulas incluyen la constante π, que es aproximadamente igual a 3.14159. El símbolo π es la letra griega llamada "pi". En programas anteriores utilizamos la siguiente declaración para producir una constante con nombre llamada PI, para utilizarla cuando convertimos dichas fórmulas en código de C++:

```
const double PI = 3.14159;
```

En el programa del cuadro 3.11 utilizamos la misma declaración, pero la colocamos cerca del principio del archivo para definirla como constante global con nombre que pueda utilizarse en todos los cuerpos de las funciones.

El compilador permite colocar las declaraciones de constantes globales con nombre en varios lugares, pero para facilitar la lectura debemos colocar juntas todas las directivas `include`, todas las constantes globales con nombre y todas las declaraciones de funciones juntas. Nosotros seguiremos esta práctica estándar y colocaremos todas nuestras declaraciones de constantes con nombre después de nuestras directivas `include` y antes de nuestras declaraciones de funciones.

CUADRO 3.11 *Constante global con nombre (parte 1 de 2)*

```
//Calcula el área de un círculo y el volumen de una esfera.
//Utiliza el mismo radio en ambos cálculos.
#include <iostream>
#include <cmath>
using namespace std;

const double PI = 3.14159;

double area(double radio);
//Devuelve el área de un círculo con el radio especificado.

double volumen(double radio);
//Devuelve el volumen de una esfera con el radio especificado.

int main()
{
    double radio_de_ambos, area_circulo, volumen_esfera;

    cout << "Escribe el radio tanto del circulo como de\n"
         << "la esfera (en centimetros): ";
    cin >> radio_de_ambos;

    area_circulo = area(radio_de_ambos);
    volumen_esfera = volumen(radio_de_ambos);

    cout << "Radio = " << radio_de_ambos << " centimetros\n"
         << "Area del circulo = " << area_circulo
         << " cm cuadrados\n"
         << "Volumen de la esfera = " << volumen_esfera
         << " cm cubicos\n";

    return 0;
}
```

CUADRO 3.11 *Constante global con nombre (parte 2 de 2)*

```
double area(double radio)
{
    return (PI * pow(radio, 2));
}

double volumen(double radio)
{
    return ((4.0/3.0) * PI * pow(radio, 3));
}
```

Diálogo de ejemplo

```
Escribe el radio tanto del circulo como de
la esfera (en centimetros): 2
Radio = 2 centimetros
Area del circulo = 12.5664 cm cuadrados
Volumen de la esfera = 33.5103 cm cubicos
```

Si colocamos una declaración de constante con nombre al principio del programa puede ayudar a la claridad de la constante con nombre que utiliza solamente una vez. Si la constante con nombre pudiera necesitar modificación en una futura versión del programa, sería más fácil encontrarla si se encuentra al principio de su programa. Por ejemplo, si colocamos la declaración de la constante para el impuesto de ventas al principio de un programa contable será más fácil revisar si el programa debe incrementar la tasa de interés.

variables globales

Es posible declarar variables comunes sin el modificador `const`, como **variables globales**, las cuales son accesibles a toda la definición dentro del archivo. Lo anterior se hace de la misma manera que en el caso de las constantes globales con nombre, excepto que el modificador `const` no se utiliza en las variables. Sin embargo, muy pocas veces es necesario utilizar tales variables. Es más, las variables globales pueden hacer que un programa sea más fácil de comprender y mantener, así que no utilizaremos variables globales. Una vez que tenga mayor experiencia en el diseño de programas, podría elegir variables globales de manera ocasional.

Los parámetros formales por valor son variables locales

Los parámetros formales son más que solamente espacios en blanco rellenados con los valores para la función. Los parámetros formales son en realidad variables locales a la definición de la función, así que se pueden utilizar como las variables locales que se declaran dentro de la definición de la función. Anteriormente explicamos el mecanismo de llamado por valor que manipula los argumentos dentro de la función. Ahora podemos definir con más detalle este mecanismo para "insertar argumentos". Cuando se invoca a una función,

los parámetros formales para la función (las cuales son variables locales) se inicializan con los valores de los argumentos. Éste es el significado preciso de la frase "insertar en lugar de los parámetros formales" los cuales ya hemos utilizado. Por lo general, un parámetro formal se utiliza solamente como una especie de espacio, o marcador de posición, que se llena con el valor de su argumento correspondiente, sin embargo, ocasionalmente un parámetro formal se utiliza como una variable cuyo valor se modifica. En esta sección explicaremos un ejemplo de un parámetro formal utilizado como variable local.

En el cuadro 3.12 se muestra un programa de facturación para el bufete de abogados Dewey, Cheatham, and Howe. Observe que, a diferencia de otros bufetes de abogados, Dewey, Cheatham, and Howe no cobra por tiempo menor a un cuarto de hora. Esto es por lo que se le denomina "el bufete compasivo". Si trabajan por una hora y catorce minutos, entonces solamente cobran cuatro cuartos de hora, no cinco cuartos de hora como otras firmas de abogados, así que solamente pagaría $600 por la consulta.

Observe el parámetro formal `minutos_trabajados` en la definición de la función `honorarios`, se utiliza como una variable y contiene su propio valor modificado por la siguiente línea, lo cual ocurre dentro de la definición de la función:

```
minutos_trabajados = horas_trabajadas*60 + minutos_trabajados;
```

No agregue una declaración para un parámetro formal.

Los parámetros formales son variables locales como las variables que declaramos dentro del cuerpo de la función. Sin embargo, no debemos agregar una declaración de variable para los parámetros formales. Listar el parámetro formal `minutos_trabajados` en la declaración de la función también sirve como la declaración de la variable. Lo siguiente es la *forma errónea* de comenzar la definición de una función para `honorarios` al declarar `minutos_trabajados` por duplicado:

```
double honorarios(int horas_trabajadas, int minutos_trabajados)
{
    int cuartos_de_hora;
    int minutos_trabajados;      ¡No haga esto!
        . . .
```

De nuevo los espacios de nombre

Hasta aquí, todos nuestros programas comienzan con el siguiente par de líneas:

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

Sin embargo, el inicio del archivo no siempre es la mejor ubicación para la línea

```
using namespace std;
```

Más adelante utilizaremos más espacios de nombres además de `std`. De hecho, podríamos utilizar espacios de nombre diferentes para definiciones de funciones diferentes. Si colocamos la directiva

```
using namespace std;
```

dentro de la llave `{` que comienza con el cuerpo de la definición de la función, entonces la directiva `using` solamente aplica a la definición de la función. Esto le permitirá utilizar dos espacios de nombres diferentes dentro de dos funciones diferentes, incluso si las dos definiciones de funciones se encuentran en el mismo archivo e incluso si los dos espacios

CUADRO 3.12 *Uso de un parámetro formal como variable local (parte 1 de 2)*

```
//Programa de facturación de bufete de abogados.
#include <iostream>
using namespace std;
const double TARIFA = 150.00; //Dólares por cuarto de hora.
double honorarios(int horas_trabajadas, int minutos_trabajados);
//Devuelve el cargo por horas_trabajadas horas y
//minutos_trabajados minutos de servicios legales.
int main()
{
    int horas, minutos;
    double factura;
    cout << "Bienvenido a las oficinas de\n"
         << "Dewey, Cheatham & Howe.\n"
         << "El bufete compasivo.\n"
         << "Escriba las horas y minutos"
         << " de su consulta:\n";
    cin >> horas >> minutos;
    factura = honorarios(horas, minutos);
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Por " << horas << " horas y " << minutos
         << " minutos, su factura es de $" << factura << endl;
    return 0;
}
double honorarios(int horas_trabajadas,
                  int minutos_trabajados)
{
     int cuartos_de_hora;
     minutos_trabajados = horas_trabajadas*60 + minutos_trabajados;
     cuartos_de_hora = minutos_trabajados/15;
     return (cuartos_de_hora*TARIFA);
}
```

La llamada a `honorarios` *no cambia el valor de* `minutos`.

`minutos_trabajados` *es una variable local que se inicializa con el valor de* `minutos`.

CUADRO 3.12 *Uso de un parámetro formal como variable local (parte 2 de 2)*

Diálogo de ejemplo

```
Bienvenido a las oficinas de
Dewey, Cheatham & Howe.
El bufete compasivo.
Escriba las horas y minutos de su consulta:
2 45
Por 2 horas y 45 minutos, su factura es de $1650.00
```

de nombres contienen algunos nombres con significados diferentes en los dos espacios de nombres.

Colocar una directiva `using` dentro de la definición de una función es análoga a colocar una declaración de variable dentro de la definición de la función. Si colocamos una definición de variable dentro de una definición de funciones, la variable es local a la función; esto es, el significado de la declaración de la variable se confina a la definición de la función. Si colocamos una directiva `using` dentro de la definición de la función, la directiva `using` es local a la definición de la función; en otras palabras, el significado de la directiva `using` se confina a la definición de la función.

Pasará algún tiempo antes de que utilicemos un espacio de nombre diferente a `std` dentro de una directiva `using`, pero sería una buena práctica comenzar a colocar estas directivas `using` en donde deben ir.

En el cuadro 3.13 rescribimos el programa del cuadro 3.11 con las directivas `using` escritas en donde deben ir. El programa en el cuadro 3.13 se comportará exactamente de la misma manera que el programa del cuadro 3.11. En este caso en particular, la diferencia es solamente de estilo, pero cuando comience a utilizar con más espacios de nombres, la diferencia afectará el rendimiento de sus programas.

Ejercicios de AUTOEVALUACIÓN

16. Si utiliza una variable en la definición de una función, ¿en dónde debe declarar la variable?, ¿en la definición de la función?, ¿en la parte **main** del programa?, ¿en cualquier lugar es correcto?

17. Suponga que una función llamada **Funcion1** contiene una variable llamada **sam** declarada dentro de la definición de la **Funcion1**, y la función llamada **Funcion2** también como una variable llamada **sam** declarada dentro de la definición de la función **Funcion2**. ¿Compilará el programa (asumiendo que todo lo demás es correcto)? Si el programa se compila, ¿se ejecutará (asumiendo que todo lo demás es correcto)? Si se ejecuta, generará un mensaje de error durante la ejecución (asumiendo que todo lo demás es correcto)? Si se ejecuta, y no produce mensaje de error durante la ejecución, ¿producirá las salidas correctas (asumiendo que todo lo demás es correcto)?

CUADRO 3.13 *Uso de un parámetro formal como variable local (parte 1 de 2)*

```
//Calcula el área de un círculo y el volumen de una esfera.
//Usa el mismo radio en ambos cálculos.
#include <iostream>
#include <cmath>
using namespace std;
const double PI = 3.14159;
double area(double radio);
//Devuelve el área de un círculo con el radio especificado.
double volumen(double radio);
//Devuelve el volumen de una esfera con el radio especificado.
int main()
{
    using namespace std;
    double radio_de_ambos, area_circulo, volumen_esfera;
    cout << "Escribe el radio tanto del circulo como de\n"
    << "la esfera (en pulgadas): ";
    cin >> radio_de_ambos;
    area_circulo = area(radio_de_ambos);
    volumen_esfera = volumen(radio_de_ambos);
    cout << "Radio = " << radio_de_ambos << " pulgadas\n"
         << "Area del circulo = " << area_circulo
         << " pulgadas cuadradas\n"
         << "Volumen de la esfera = " << volumen_esfera
         << " pulgadas cubicas\n";
    return 0;
}
```

CUADRO 3.13 *Uso de espacios de nombres (parte 2 de 2)*

```
double area(double radio)
{
    using namespace std;
    return (PI * pow(radio, 2));
}

double volumen(double radio)
{
    using namespace std;
    return ((4.0/3.0) * PI * pow(radio, 3));
}
```

El diálogo de ejemplo de este programa podría ser el mismo que el del programa para el cuadro 3.11.

18. Se supone que la siguiente función debe tomar como argumentos una longitud expresada en pies y pulgadas y devolver el número total de la longitud en pulgadas. Por ejemplo, `pulgadas_totales(1, 2)` debe devolver `14`, debido a que `1` pie y `2` pulgadas es lo mismo que `14` pulgadas. ¿Se ejecutará de manera correcta la función? ¿Y si no, por qué?

```
double pulgadas_totales(int pies, int pulgadas)
{
        pulgadas = 12*pies + pulgadas;
        return pulgadas;
}
```

19. Escriba la declaración de la función y la definición de la función llamada `lee_filtro` que no contienen parámetros y que devuelve un valor de tipo `double`. La función `lee_filtro` solicita al usuario un valor de tipo `double` y devuelve el valor dentro de variables locales. La función devuelve el valor de `lee` proporcionada si el valor es mayor o igual que cero y devuelve cero si el valor es negativo.

EJEMPLO DE PROGRAMACIÓN
La función factorial

El cuadro 3.14 contiene la declaración y la definición de la función para una función matemática empleada generalmente conocida como función *factorial*. Por lo general, en los textos de matemáticas la función factorial se escribe *n!* y se define como el producto de todos los enteros desde 1 a *n*. En la notación matemática tradicional, podemos definir *n!* de la siguiente manera:

$n! = 1 \times 2 \times 3 \times \ldots \times n$

CUADRO 3.14 *Función factorial*

Declaración de función

```
int factorial(int n);
//Devuelve el factorial de n.
//El argumento n debe ser positivo.
```

Definición de la función

```
int factorial(int n)
{
    int producto = 1;
    while (n> 0)
    {
        producto = n * producto;
        n--;
    }
    return producto;
}
```

El parámetro formal `n` *se utiliza como variable local.*

En la definición de la función realizamos la multiplicación mediante un ciclo `while`. Observe que la multiplicación se realiza en el orden inverso al que usted podría esperar. El programa multiplica por `n`, luego `n-1`, luego `n-2`, y así sucesivamente.

un parámetro formal utilizado como variable local

La definición de la función para `factorial` utiliza dos variables locales: `producto`, la cual declaramos como el inicio del cuerpo de la función, y el parámetro formal `n`. Ya que un parámetro es una variable local, podemos cambiar su valor. En este caso modificamos el valor del parámetro formal `n` con el operador de decremento `n--`. (Explicamos el operador de decremento en el capítulo 2.)

Cada vez que se ejecuta el cuerpo del ciclo, el valor de la variable `producto` se multiplica por el valor de `n`, y posteriormente el valor de `n` se decrementa en uno usando `n--`. Si la función `factorial` se invoca con `3` como argumento, entonces la primera vez que se ejecuta el cuerpo del ciclo el valor de `producto` será 3, la siguiente vez que se ejecuta el cuerpo del ciclo el valor del producto será `3*2`, la siguiente vez el valor de `producto` será 3*2*1 y luego el ciclo `while` terminará. De esta manera, lo siguiente establecerá la variable `x` igual a `6` que es `3*2*1`:

```
x = factorial(3);
```

Observe que cuando se declara la variable local `producto` se inicializa al valor `1`. (En el capítulo `2` explicamos esta manera de inicializar una variable al declararla.) Es fácil ver que `1` es el valor inicial correcto para la variable `producto`. Para ello observe que después de que se ejecuta el cuerpo del ciclo `while` por primera vez, queremos que el valor de

`producto` sea igual al valor (original) del parámetro formal `n`; si inicializamos `producto` a `1`, sucederá precisamente eso.

3.6 Sobrecarga de nombres de función

"... —y eso demuestra que hay trescientos sesenta y cuatro días en los que podrías recibir regalos de no cumpleaños—"
"Ciertamente", dijo Alicia.
"Y sólo uno para los regalos de cumpleaños, ¿no? ¡Eso sí que es gloria!"
"No sé qué quieres decir con 'gloria'", dijo Alicia.
Humpty Dumpty esbozó una sonrisa despreciativa. "Claro que no— hasta que te lo diga. Quiero decir '¡Eso sí que es un bonito argumento convincente!'"
"Pero 'gloria' no significa 'un bonito argumento convincente'", objetó Alicia.
"Cuando yo uso una palabra", dijo Humpty Dumpty usando un tono burlón, "significa exactamente lo que quiero que signifique— ni más ni menos."
"La cuestión es", dijo Alicia, "si puedes o no hacer que las palabras signifiquen tantas cosas distintas."
"La cuestión es", sentenció Humpty Dumpty, "quién va a ser el amo— eso es todo."

Lewis Carroll, *Alicia a través del espejo*

C++ nos permite dar dos o más definiciones distintas del mismo nombre de función, lo que significa que podemos reutilizar nombres que son muy atractivos por tener un significado intuitivo en muy diversas situaciones. Por ejemplo, podríamos tener tres funciones llamadas `max`: una que calcula el más grande de dos números, otra que calcula el más grande de tres números y otra más que calcula el más grande de cuatro números. Cuando damos dos (o más) definiciones para el mismo nombre de función, estamos **sobrecargando** el nombre de función. La sobrecarga requiere un cuidado adicional al definir las funciones y no debe usarse si no mejora mucho la legibilidad de nuestros programas. Sin embargo, en los casos en que es apropiada, la sobrecarga puede ser muy eficaz.

sobrecarga

Introducción a la sobrecarga

Supongamos que estamos escribiendo un programa que requiere el cálculo de la media de dos números. Podríamos usar la siguiente definición de función:

```
double media (double n1, double n2)
{
      return ((n1 + n2)/2.0)
}
```

Supongamos ahora que nuestro programa también requiere una función para calcular la media de tres números. Podríamos definir una nueva función llamada `media3` como sigue:

```
double media3(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3)/3.0);
}
```

Esto funciona, y en muchos lenguajes de programación es la única opción. Por fortuna, C++ permite una solución más elegante. En C++ podemos utilizar el mismo nombre de función `media` para ambas funciones. Es decir, podemos usar la siguiente definición de función en lugar de la definición de la función `media3`:

```
double media(double n1, double n2, double n3)
{
return ((n1 + n2 + n3)/3.0);
}
```

de modo que el nombre de función `media` tenga dos definiciones. Éste es un ejemplo de sobrecarga. En este caso sobrecargamos el nombre de función `media`. En el cuadro 3.15 hemos incrustado estas dos definiciones de la función `media` en un programa completo. Observe que cada definición de función tiene su propia declaración de función.

La sobrecarga es una excelente idea, pues hace a los programas más fáciles de entender y evita que tengamos que quebrarnos la cabeza inventando un nombre nuevo para una función sólo porque ya usamos el nombre más natural en alguna otra definición de función. Pero, ¿cómo sabe el compilador cuál definición de función debe usar cuando encuentra una llamada a una función que tiene dos o más definiciones? El compilador no puede leer la mente del programador. Para saber cuál definición de función debe usar, el compilador verifica el número y los tipos de los argumentos de la llamada de función. En el programa del cuadro 3.15, una de las funciones llamada `media` tiene dos argumentos y la otra tiene tres argumentos. Para saber cuál definición debe usar, el compilador simplemente cuenta el número de argumentos en la llamada de función. Si hay dos argumentos, usa la primera definición; si hay tres argumentos, usa la segunda.

determinación de cuál definición se utiliza

Siempre que damos dos o más definiciones para el mismo nombre de función, éstas deben tener diferentes especificaciones de argumentos; es decir, cualesquier dos definiciones de función que tengan el mismo nombre deberán usar diferentes números de parámetros formales o usar parámetros formales de tipos distintos (o ambas cosas). Cabe señalar que cuando sobrecargamos un nombre de función los prototipos de las dos definiciones distintas deben diferir en sus parámetros formales. *No podemos sobrecargar un nombre de función dando dos definiciones que sólo difieren en el tipo del valor devuelto*.

Sobrecarga del nombre de una función

Si tenemos dos o más definiciones para el mismo nombre de función, estamos **sobrecargando** ese nombre. Cuando sobrecargamos un nombre de función, las definiciones de función deben tener diferentes números de parámetros formales o algunos parámetros formales deben ser de tipos distintos. Al llegar a una llamada de función, el compilador usa la definición de función cuyo número de parámetros formales y cuyos tipos de parámetros formales coincidan con los argumentos de la llamada.

CUADRO 3.15 *Sobrecarga del nombre de una función*

```
//Ilustra la sobrecarga del nombre de función media.
#include <iostream>

double media(double n1, double n2);
//Devuelve la media de los dos números n1 y n2.

double media(double n1, double n2, double n3);
//Devuelve la media de los tres números n1, n2 y n3.

int main()
{
    using namespace std;
    cout << "La media de 2.0, 2.5 y 3.0 es "
         << media(2.0, 2.5, 3.0) << endl;

cout << "La media de 4.5 y 5.5 es "
     << media(4.5, 5.5) << endl;

    return 0;
}

double media(double n1, double n2)
{
    return ((n1 + n2)/2.0);
}

double media(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3)/3.0);
}
```

dos argumentos

tres argumentos

Salida

```
La media de 2.0, 2.5 y 3.0 es 2.50000
La media de 4.5 y 5.5 es 5.00000
```

La sobrecarga no es realmente algo nuevo para el usted. En el capítulo 2 vimos una especie de sobrecarga que se realiza mediante el operador de división /. Si ambos operandos son de tipo *int*, como en *13/2*, el valor devuelto es el resultado de una división entera, en este caso 6. Por otra parte, si uno o ambos operandos son de tipo *double*, el valor devuelto es el resultado de la división ordinaria; por ejemplo, 13/2.0 devuelve el valor 6.5. Hay dos definiciones para el operador de división /, y las dos definiciones se distinguen no por tener diferente cantidad de operandos, sino por requerir operandos de distintos tipos. La diferencia entre sobrecargar / y sobrecargar nombres de función es que el compilador ya efectuó la sobrecarga de /, mientras que nosotros programamos la sobrecarga del nombre de función. En un capítulo posterior veremos cómo sobregargar operadores como +, –, etcétera.

EJEMPLO DE PROGRAMACIÓN

Programa para comprar pizzas revisado

La Unión de Consumidores de Pizza ha tenido mucho éxito con el programa que le escribimos en el cuadro 3.9. De hecho, ahora todo mundo adquiere la pizza que es la mejor compra. Una pizzería de mala muerte solía hacer dinero engañando a los consumidores para que compraran la pizza más costosa, pero nuestro programa ha puesto fin a sus perversas prácticas. Sin embargo, los dueños desean continuar con su deleznable comportamiento y se les ha ocurrido una nueva forma de engañar a los consumidores. Ahora ofrecen pizzas redondas y pizzas rectangulares. Saben que el programa que escribimos no puede manejar pizzas con forma rectangular, y así esperan confundir otra vez a los consumidores. Necesitamos actualizar nuestro programa para poder frustrar su malévolo plan. Queremos modificar el programa de modo que pueda comparar una pizza redonda y una rectangular.

Los cambios que necesitamos hacer a nuestro programa evaluador de pizzas son obvios: necesitamos modificar un poco las entradas y las salidas de modo que manejen dos formas de pizza distintas. También necesitamos añadir una nueva función que pueda calcular el costo por pulgada cuadrada de una pizza rectangular. Podríamos usar la siguiente definición de función en nuestro programa para calcular el precio unitario de una pizza rectangular:

```
double preciounitario_rectangular
       (int largo, int ancho, double precio)
{
      double area = largo * ancho;
      return (precio/area);
}
```

Sin embargo, el nombre de esta función es un tanto largo; de hecho, es tan largo que tuvimos que poner el encabezado de la función en dos líneas. Eso está permitido, pero sería más bonito usar el mismo nombre, preciounitario, tanto para la función que calcula el precio unitario de una pizza redonda como de la que calcula el precio unitario de una pizza rectangular. Puesto que C++ permite sobrecargar nombres de función, podemos hacerlo. Tener dos definiciones de la función preciounitario no representará un problema para el compilador, porque las dos funciones tienen diferente número de argumentos. El cuadro 3.16 muestra el programa que obtuvimos al modificar nuestro programa evaluador de pizzas de modo que nos permitiera comparar pizzas redondas y rectangulares.

CUADRO 3.16 *Sobrecarga del nombre de una función (parte 1 de 3)*

```
//Determina si una pizza redonda o una rectangular es la mejor compra.
#include <iostream>

double preciounitario(int diametro, double precio);
//Devuelve el precio por pulgada cuadrada de una pizza redonda.
//El parámetro formal diametro es el diámetro de la pizza en pulgadas.
//El parámetro formal precio es el precio de la pizza.

double preciounitario(int largo, int ancho, double precio);
//Devuelve el precio por pulgada cuadrada de una pizza rectangular
//cuyas dimensiones son largo por ancho pulgadas.
//El parámetro formal precio es el precio de la pizza.

int main()
{
    using namespace std;
    int diametro, largo, ancho;
    double precio_redonda, preciounit_redonda,
           precio_rectangular, preciounit_rectangular;

    cout << "Bienvenido a la Union de Consumidores de Pizza.\n";
    cout << "Escriba el diametro en pulgadas"
         << " de una pizza redonda: ";
    cin >> diametro;
    cout << "Escriba el precio de una pizza redonda: $";
    cin >> precio_redonda;
    cout << "Escriba el largo y el ancho en pulgadas\n"
         << "de una pizza rectangular: ";
    cin >> largo >> ancho;
    cout << "Escriba el precio de una pizza rectangular: $";

    cin >> precio_rectangular;

    preciounit_rectangular = preciounitario(largo, ancho, precio_rectangular);
    preciounit_redonda = preciounitario(diametro, precio_redonda);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
```

CUADRO 3.16 *Sobrecarga de un nombre de función (parte 2 de 3)*

```
    cout << endl
         << "Pizza redonda: Diametro = "
         << diametro << " pulg\n"
         << "Precio = $" << precio_redonda
         << " Por pulgada cuadrada = $" << preciounit_redonda
         << endl
         << "Pizza rectangular: largo = "
         << largo << " pulg\n"
         << "Pizza rectangular: ancho = "
         << ancho << " pulg\n"
         << "Precio = $" << precio_rectangular
         << " Por pulgada cuadrada = $" << preciounit_rectangular
         << endl;
    if (preciounit_redonda < preciounit_rectangular)
        cout << "La redonda es la mejor compra.\n";
    else
        cout << "La rectangular es la mejor compra.\n";
    cout << "Buon Appetito! \n";
    return 0;
}

double preciounitario(int diametro, double precio)
{
    const double PI = 3.14159;
    double radio, area;

    radio = diametro/static_cast<double>(2);
    area = PI * radio * radio;
    return (precio/area);
}

double preciounitario(int largo, int ancho, double precio)
{
double area = largo * ancho;
return (precio/area);
}
```

CUADRO 3.16 *Sobrecarga de un nombre de función (parte 3 de 3)*

Diálogo de ejemplo

```
Bienvenido a la Union de Consumidores de Pizza.
Escriba el diametro en pulgadas de una pizza redonda: 10
Escriba el precio de una pizza redonda: $8.50
Escriba el largo y el ancho en pulgadas
de una pizza rectangular: 6 4
Escriba el precio de una pizza rectangular: $7.55

Pizza redonda: Diametro = 10 pulg
Precio = $8.50 Por pulgada cuadrada= $0.11
Pizza rectangular: largo = 6 pulg
Pizza rectangular: ancho = 4 pulg
Precio = $7.55 Por pulgada cuadrada = $0.31
La redonda es la mejor compra.
```

Conversión automática de tipos

Suponga que un programa contiene la siguiente definición de función, y que *no* ha sobrecargado el nombre de función `mpg` (así que ésta es la única definición de una función llamada `mpg`).

```
double mpg(double millas, double galones)
//Devuelve millas por galón.
{
    return (millas/galones);
}
```

Si invocamos la función *`mpg`* con argumentos de tipo *`int`*, C++ convertirá automáticamente esos argumentos a valores de tipo *`double`*. Por tanto, lo que sigue desplegará en la pantalla `22.5 millas por galon`:

```
cout << mpg(45, 2) << " millas por galon";
```

C++ convierte `45` a `45.0` y `2` a `2.0`, y luego efectúa la división `45.0/2.0` para obtener el valor devuelto, que es `22.5`.

interacción de la sobrecarga y la conversión de tipo

Si una función requiere un argumento de tipo *`double`* y le proporcionamos un argumento de tipo *`int`*, C++ convertirá automáticamente el argumento *`int`* en un valor de tipo *`double`*. Esto es tan útil y natural que casi ni pensamos en ello. Sin embargo, la sobrecarga puede interferir con esta conversión automática de tipos. Veamos un ejemplo.

Supongamos que (por imprudencia) sobrecargamos el nombre de función `mpg` de modo que nuestro programa también contiene la siguiente definición de `mpg` (además de la anterior):

```
int mpg(int goles, int fallos)
//Devuelve el mejor promedio de goleo
//que se calcula como (goles – fallos).
{
     return (goles - fallos);
}
```

En un programa que contenga ambas definiciones de la función `mpg`, lo que sigue producirá `43 millas por galon` (ya que `45 - 2` es `43`):

```
cout << mpg(45, 2) << " millas por galon";
```

Cuando C++ ve la llamada de función `mpg(45, 2)`, que tiene dos argumentos de tipo *`int`*, busca *primero* una definición de la función `mpg` que tenga dos parámetros formales de tipo *`int`*. Si C++ encuentra tal definición, la usará. C++ no convierte un argumento *`int`* en un valor de tipo *`double`* a menos que ésa sea la única forma de encontrar un definición de función que coincida.

El ejemplo de `mpg` ilustra otro aspecto importante de la sobrecarga. No debemos usar el mismo nombre de función para dos funciones que no tienen relación entre sí. Un uso tan descuidado de nombres de función seguramente producirá confusiones.

Ejercicios de AUTOEVALUACIÓN

20. Suponga que tiene dos definiciones de funciones con las siguientes declaraciones de funciones:

    ```
    double puntaje (double tiempo, double distancia);
    int puntaje(double puntos);
    ```

    ¿Cuál definición de función se utilizaría en la siguiente llamada y por qué? (`x` es de tipo *`double`*).

    ```
    puntaje_final = puntaje(x);
    ```

21. Suponga que tiene dos definiciones de una función con las siguientes declaraciones:

    ```
    double respuesta(double dato1, double dato2);
    double respuesta(double hora, int cuenta);
    ```

    ¿Cuál definición de función se utilizaría en la siguiente llamada a función y por qué sería utilizada? (`x` y `y` son de tipo *`double`*).

    ```
    x = respuesta(y, 6.0);
    ```

22. Suponga que tiene dos definiciones de una función con la declaración que aparece en el ejercicio de autoevaluación 21. ¿Cuál definición de función se utilizaría en la siguiente llamada a función y por qué?

    ```
    x = respuesta(5, 6);
    ```

23. Suponga que tiene dos definiciones de una función con las declaraciones que aparecen en el ejercicio de autoevaluación 21. ¿Cuál definición de función se utilizaría en la siguiente llamada a función y por qué?

```
x = respuesta(5, 6.0);
```

24. Esta pregunta tiene que ver con el ejemplo de programación "Programa de Pizzas revisado". Suponga que el vendedor de pizzas diabólico que siempre intenta engañar a los consumidores introduce una pizza cuadrada. ¿Podría sobrecargar la función `preciounitario` para que pueda calcular el precio por centímetro cuadrado de pizza así como el precio por centímetro cuadrado de una pizza redonda? ¿Por qué sí, y por qué no?

25. Revise el programa en el cuadro 3.16. La función `main` contiene la directiva *`using`*:

```
using namespace std;
```

¿Por qué el método `preciounitario` contiene la directiva *`using`*?

Resumen del capítulo

- Un buen plan de ataque para el diseño del algoritmo para un programa es dividirlo en algunas cuantas tareas más pequeñas, luego descomponer cada subtarea en subtareas más pequeñas, y así sucesivamente hasta que las subtareas sean lo suficientemente sencillas para implementarse en código de C++. Este método se llama diseño **descendente**.

- Una función que devuelve un valor es como un pequeño programa. Los argumentos de la función sirven como entrada para este "pequeño programa" y el valor devuelto es como la salida del "pequeño programa".

- Cuando una subtarea para un programa toma algunos valores como entrada y produce un sólo valor como único resultado, entonces la subtarea se puede implementar como una función.

- Una función se debe definir de tal forma que se pueda utilizar como una caja negra. El programador que utiliza la función no debe requerir conocer detalle alguno acerca de cómo se codificó la función. Todo lo que el programador necesita es la declaración de la función y el comentario anexo que describe el valor devuelto. Algunas veces a esta regla se le llama **principio de abstracción de procedimiento**.

- Se dice que una variable que se declara dentro de una función es **local a la función**.

- Los constantes globales con nombre se declaran con el uso del modificador *`const`*. Por lo general las declaraciones de constantes globales con nombre se colocan al inicio de un programa después de las directivas `include` y antes de las declaraciones de funciones.

- Los parámetros formales de llamada por valor (los cuales son los únicos parámetros formales que explicamos en este capítulo) son variables locales a la función. En ocasiones, es útil utilizar un parámetro formal como una variable local.

- Si tenemos dos o más definiciones de funciones para el mismo nombre de función, a esto se le llama **sobrecarga** del nombre de una función. Cuando sobrecarga el nombre de una función, las definiciones de las funciones deben tener diferentes de números de parámetros formales o algunos parámetros de tipos diferentes.

Respuestas a los ejercicios de autoevaluación

1.
```
4.0     4.0     8.0
8.0     8.0     1.21
3       3       0
3.0     3.5     3.5
6.0     6.0     5.0
5.0     4.5     4.5
3       3.0     3.0
```

2.
```
sqrt(x + y), pow(x, y + 7), sqrt(area + margen),

sqrt(hora + marca)/nadie (−b + sqrt(b*b − 4*a*c))/(2*), abs(x − y) o
                                                        labs(x − y) o
                                                        fabs(x − y)
```

3.
```
//Calcula la raíz cuadrada de 3.14159.
#include <iostream>
#include <cmath> //proporciona sqrt y PI.
using namespace std;
int main( )
{
        cout << "La raiz cuadrada de " >> PI
             << sqrt(PI) << endl;
        return 0;
}
```

4. a)
```
//Para determinar si el compilador tolerará
//espacios entre # e include en #include:
  #include <iostream>
int main( )
{
   cout << "hola mundo" << endl;
   return 0;
}
```

   b)
```
//Para determinar si el compilador permite espacios
//entre # e include en #include:
# include<iostream>
using namespace std;
//El resto del programa puede ser idéntico al de arriba
```

5.
```
Wow
```

6. La declaración de la función es:

```
int suma(int n1, int n2, int n3);
//Devuelve la suma de n1, n2 y n3.
```

La definición de la función es:

```
int suma(int n1, int n2, int n3)
{
   return (n1 + n2 + n3);
}
```

7. La declaración de la función es:

```
double prom(int n1, double n2);
//Devuelve el promedio de n1 y n2.
```

La definición de la función es:

```
double prom(int n1, double n2);
{
    return ((n1 + n2)/2.0);
}
```

8. La declaración de función es:

```
char prueba_positivo(double numero)
//Devuelve 'P' si numero es positivo
//Devuelve 'N' si numero es negativo o cero.
```

La definición de función es

```
char prueba_positivo(double numero)
{
    if (numero > 0
        return 'P'
    else
        return 'N'
}
```

9. Suponga que la función se define con parámetros, digamos `param1` y `param2`. Además se llaman con argumentos correspondientes `arg1` y `arg2`. Los valores de los argumentos se "insertan" en lugar de los parámetros formales correspondientes, `arg1` en lugar de `param1` y `arg2` en lugar de `param2`. Luego se usan los parámetros formales en la función.

10. La funciones predefinidas (de biblioteca) por lo regular requieren que incluyamos un archivo de encabezado con `#include`. En el caso de las funciones definidas por el programador, éste coloca el código de la función en el archivo que contiene la parte principal o en otro archivo que se compilará y enlazará con el programa principal.

11. El comentario explica qué valor devuelve la función y proporciona cualquier otra información que se necesite conocer para usar la función.

12. El principio de abstracción de procedimientos dice que una función debe escribirse de modo que se pueda usar como una caja negra. Esto implica que el programador que usa la función no necesita examinar el cuerpo de la definición de la función para ver cómo opera la función. La declaración de la función y el comentario que lo acompaña deben ser lo único que el programador necesita saber para poder usar la función.

13. Cuando decimos que el programador que usa la función debe poder tratar la función como una caja negra, queremos decir que el programador no debe tener necesidad de examinar el cuerpo de la definición de la función para ver la manera en que ésta funciona. La declaración de la función y el comentario que lo acompaña deben ser lo único que el programador necesita saber para poder usar la función.

14. Para aumentar nuestra confianza en nuestro programa, debemos probarlo con valores de entrada para los que conozcamos las respuestas correctas. Quizá podamos calcular las respuestas por otros medios, como lápiz y papel, o una calculadora. Los casos limitantes (p. ej., la pizza de dos pulgadas del ejemplo del texto) u otros casos sencillos son buenos puntos de partida.

15. Sí, la función devolvería el mismo valor en ambos casos, por lo que las dos definiciones son equivalentes como cajas negras.

16. Si usamos una variable en una definición de función, debemos declararla en el cuerpo de la definición de la función.

17. Todo saldrá bien. El programa se compilará (suponiendo que todo lo demás está correcto), se ejecutará (suponiendo que todo lo demás está correcto), no generará mensajes de error al ejecutarse (suponiendo que todo lo demás está correcto) y producirá las salidas correctas (suponiendo que todo lo demás está correcto).

18. La función operará perfectamente. Ésa es toda la respuesta, pero hay cierta información adicional: el parámetro formal `pulgadas` es un parámetro de llamada por valor y por tanto, como vimos en el texto, es una variable local. Así pues, el valor del argumento no cambiará.

19. La declaración de función es:

```
double leer_filtro();
//Lee un número del teclado. Devuelve un número
//leído si éste es >= 0; de lo contrario devuelve cero.
```

La declaración de función es:

```
//utiliza iostream
double leer_filtro()
{
     using namespace std;
     double valor_leido
     cout << "Introduzca un numero:\n";
     cin << valor_leido;

     if (valor_leido >= 0)
       return valor_leido;
     else
        retun 0.0;
}
```

20. La llamada de función sólo tiene un argumento, así que usará la definición de función que sólo tiene un parámetro formal.

21. La llamada de función tiene dos argumentos de tipo `double`, así que usará la función correspondiente a la declaración que tiene dos argumentos de tipo `double` (es decir, la primera declaración de función).

22. El segundo argumento es de tipo `int` y C++ convertirá automáticamente al primer argumento al tipo `double` en caso de ser necesario; por lo tanto, utilizará la función correspondiente a la declaración de función que tiene dos argumentos de tipo `double` (es decir, la primera declaración de función).

23. El segundo argumento es de tipo `double` y C++ convertirá automáticamente al primer argumento al tipo al tipo `double` en caso de que sea necesario; por lo tanto, utilizará la función correspondiente a la declaración de función que tiene dos argumentos de tipo `double` (es decir, la primera declaración de función).

24. No puede hacerse (al menos no de forma elegante). Las formas naturales de representar una pizza redonda y una cuadrada son iguales. Las dos se representan con un número, que es el radio de una pizza redonda o la longitud del lado de una pizza cuadrada. En ambos casos la función `preciounitario` necesitaría un parámetro formal de tipo `double` para el precio y uno de tipo `int` para el tamaño (sea el radio o el lado). Así, los dos prototipos tendrían el mismo número de parámetros formales de los mismos tipos. (Específicamente, ambos tendrían un parámetro formal de tipo `double` y uno de tipo `int`.) El compilador no podría decidir cuál definición usar. No obstante, podríamos frustrar la estrategia de la pizzería inmoral definiendo dos funciones, pero tendrían que tener diferente nombre.

25. La definición de `preciounitario` no tiene ninguna entrada o salida y por lo tanto no utilizará la biblioteca iostream. Por lo general necesitamos la direcitva `using` ya que `cin` y `cout` están definidos en `iostream` y esas definiciones colocan a `cin` y a `cout` en el espacio de nombres `std`.

Proyectos de programación

1. Un litro equivale a 0.264179 galones. Escriba un programa que lea el número de litros de gasolina consumidos por el automóvil del usuario y el número de millas recorridas por el automóvil, y despliegue el número de millas por galón que da el automóvil. El programa debe permitir al usuario repetir el cálculo cuantas veces desee. Defina una función que calcule el número de millas por galón. El programa debe usar una constante definida globalmente para el número de litros que hay en un galón.

2. Modifique el programa del proyecto de programación 1 de manera que reciba los datos de dos carros y despliegue el número de millas por galón que da cada automóvil. Su programa debe además informar qué carro tiene mejor rendimiento (el mayor número de millas por galón).

3. El precio de las acciones normalmente se da al octavo de dólar más cercano; por ejemplo, 29 7/8 u 89 1/2. Escriba un programa que calcule el valor de las acciones de una compañía que el usuario posee. El programa pide el número de acciones que se tienen, la porción de dólares enteros del precio y la porción fraccionaria. Ésta última debe introducirse como dos valores `int`, uno para el numerador y otro para el denominador. Luego, el programa desplegará el valor de las acciones del usuario. El programa deberá permitir al usuario repetir este cálculo cuantas veces desee. El programa incluirá una definición de función con tres argumentos `int` que consisten en la porción entera del precio y los dos enteros que forman la parte fraccionaria. La función devuelve el precio de una acción como un solo número de tipo `double`.

4. Escriba un programa que calcule la tasa de inflación para el año anterior. El programa pide el precio de un artículo (digamos un bizcocho o un diamante de un quilate) tanto hace un año como hoy, y estima la tasa de inflación como la diferencia en los precios dividida entre el precio hace un año. El programa deberá permitir al usuario repetir este cálculo cuantas veces desee. Defina una función que calcule la tasa de inflación, la cual debe ser un valor de tipo `double` que represente un porcentaje, digamos 5.3 para 5.3%.

5. Mejore su programa del ejercicio anterior haciendo que también despliegue el precio estimado del artículo un año y dos años después del momento en que se hace el cálculo. El aumento en el costo en un año se estima como la tasa de inflación multiplicada por el precio al iniciar el año. Defina una segunda función que determine el costo estimado de un artículo en un año, dado el precio actual del artículo y la tasa de inflación como argumentos.

6. Escriba la definición de una función que calcule los intereses del saldo de una cuenta de tarjeta de crédito. La función recibe argumentos para el saldo inicial, la tasa de interés mensual y el número de meses por los que se deberá pagar intereses. El valor devuelto son los intereses a pagar. No olvide componer el interés, es decir, cobrar intereses sobre los intereses no pagados. Los intereses no pagados se suman al saldo no pagado, y los intereses para el siguiente mes se calculan usando este saldo mayor. Use un ciclo `while` similar (pero no necesariamente idéntico) al que se muestra en el cuadro 2.14. Incruste la función en un programa que lea valores para la tasa de interés, el saldo inicial en la cuenta y el número de meses, y luego exhiba los intereses a pagar. Incorpore la definición de la función en un programa que permita al usuario calcular los intereses a pagar sobre el saldo de una cuenta de crédito. El programa deberá permitir al usuario repetir este cálculo hasta que quiera terminar

7. La fuerza de atracción gravitacional entre dos cuerpos con masas m_1 y m_2 separados por una distancia d está dada por

$$F = \frac{G m_1 m_2}{d^2}$$

donde G es la constante de gravitación universal:

G=6.673 $\times$ 10^{-8} cm3/(g $\times$ seg^2)

Escriba una definición de función que reciba argumentos para las masas de dos cuerpos y la distancia entre ellos, y que devuelva la fuerza gravitacional entre ellos. Puesto que se usará la fórmula anterior, la fuerza gravitacional estará en dinas. Una dina es igual

g x cm/seg^2

Use una constante definida globalmente para la constante de gravitación universal. Incorpore su definición de función en un programa completo que calcule la fuerza de atracción gravitacional entre dos objetos si se le proporcionan entradas apropiadas. El programa deberá permitir al usuario repetir este cálculo cuantas veces desee.

8. Escriba un programa que calcule el costo anual después de impuestos de una casa nueva durante el primer año después de comprada. El costo se calcula como el costo anual de la hipoteca menos los impuestos ahorrados. Las entradas deben ser el precio de la casa y el pago inicial (enganche). El costo anual de la hipoteca puede estimarse como el 3% del saldo del préstamo inicial, que se usa para restituir el préstamo, más 8% del saldo del préstamo inicial por concepto de intereses. El saldo del préstamo inicial es el precio

menos el enganche. Suponga una tasa de impuesto marginal del 35% y que los pagos de intereses son deducibles de impuestos. Así, el ahorro de impuestos es el 35% de los intereses pagados. El programa debe usar al menos dos definiciones de función. El programa deberá permitir al usuario repetir este cálculo cuantas veces desee.

9. Escriba un programa que pida la estatura, el peso y la edad del usuario y luego calcule las tallas de ropa según las fórmulas:

- Tamaño de sombrero = peso en libras dividido entre estatura en pulgadas y el resultado multiplicado por 2.9.
- Tamaño de saco (pecho en pulgadas) = altura multiplicada por peso y dividida entre 288. Se hace un ajuste sumando 1/8 de pulgada por cada 10 años después de los 30 años de edad. (Cabe señalar que el ajuste sólo se hace una vez transcurridos 10 años completos. Así pues, no se hace ajuste para las edades de 30 a 39, pero se suma 1/8 de pulgada por los 40 años.)
- Cintura en pulgadas = peso dividido entre 5.7. Se hace un ajuste sumando 1/10 de pulgada por cada 2 años después de los 28 años de edad. (Cabe señalar que el ajuste sólo se hace una vez transcurridos 2 años completos. Así pues, no se hace ajuste para la edad de 29, pero se suma 1/10 de pulgada por los 30 años.)

Use funciones para cada cálculo. El programa deberá permitir al usuario repetir este cálculo cuantas veces desee.

10. Modifique el programa del proyecto de programación 9 de manera que también calcule las tallas del saco y de la cintura del usuario después de 10 años.

11. El hecho que contemos con la "bendición" de varias funciones de valor absoluto es un accidente de la historia. Ya había bibliotecas para C cuando llegó C++; era fácil usarlas, por lo que no se reescribieron usando sobrecarga de nombres de función. Usted debe encontrar todas las funciones de valor absoluto que pueda, y reescribirlas sobrecargando el nombre de función `abs`. Como mínimo, deberán estar representados los tipos *`int`*, *`long`*, *`float`* y *`double`*.

12. Escriba una función sobrecargada `maximo` que tome dos o tres parámetros de tipo `double` y devuelva el más grande de ellos.

4 Funciones para todas las subtareas

4

Funciones para todas las subtareas

Todo es posible.

MÁXIMA COMÚN

Introducción

La estrategia de diseño descendente que vimos en el capítulo 3 es una forma eficaz de diseñar un algoritmo para un programa. Dividimos la tarea del programa en subtareas y luego implementamos los algoritmos de esas subtareas como funciones. Hasta aquí, hemos visto cómo definir funciones que toman los valores de algunos argumentos y devuelven un solo valor como resultado de la llamada de función. Una subtarea que calcula un solo valor es una clase muy importante de subtarea, pero no es la única clase. En este capítulo completaremos nuestra descripción de las funciones C++ y presentaremos técnicas para diseñar funciones que realicen otro tipo de subtareas.

Prerrequisitos

Deberá leer los capítulos 2 y 3 antes de leer este capítulo.

4.1 Funciones *void*

En C++ las subtareas se implementan como funciones. Las funciones que vimos en el capítulo 3 siempre devuelven un solo valor, pero hay otras formas de subtareas. Una subtarea podría producir varios valores o ninguno. En C++, una función debe devolver un solo valor o no devolver ninguno. Como veremos más adelante, las subtareas que producen varios valores distintos por lo regular (y tal vez paradójicamente) se implementan como funciones que no devuelven ningún valor. Sin embargo, por el momento, evitemos esa complicación y concentrémonos en las subtareas que intuitivamente no producen ningún valor y veamos cómo se implementan. Una función que no devuelve ningún valor es una **función** *void*. Por ejemplo, una subtarea común en un programa es enviar a la salida los resultados de algún cálculo. Esta subtarea produce salidas en la pantalla, pero no produce valores que el resto del programa vaya a usar. Este tipo de subtarea se implementaría como una función *void*.

funciones *void* no devuelven ningún valor

Definiciones de funciones *void*

En C++, una función *void* se define de forma muy similar a como se definen las funciones que devuelven un valor. Por ejemplo, la que sigue es una función *void* que despliega el resultado de un cálculo que convierte una temperatura expresada en grados Fahrenheit en una expresada en grados Celsius. El cálculo en sí se realizará en otro lugar del programa. Esta función *void* sólo implementa la subtarea de desplegar los resultados del cálculo. Por ahora, no nos preocuparemos de cómo se efectúa el cálculo.

```
void mostrar_resultados(double grados_f, double grados_c)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
    cout << grados_f
         << " grados Fahrenheit equivale a\n"
         << grados_c << " grados Celsius.\n";
    return;
}
```

definición de función

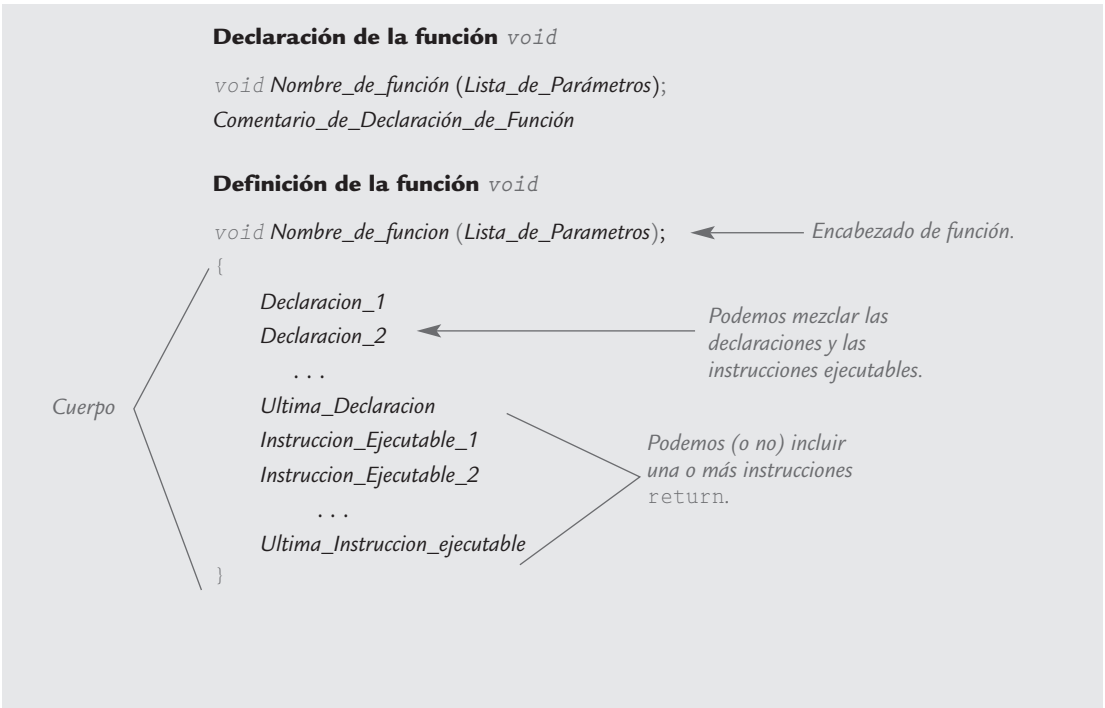
Como ilustra la **definición de función** anterior, sólo hay dos diferencias entre una definición de una función *void* y las definiciones de funciones que vimos en el capítulo 3. Una diferencia es que usamos la palabra clave *void* donde normalmente especificaríamos el tipo del valor devuelto. Esto le dice al compilador que la función no devolverá ningún valor. Usamos la palabra *void* como forma de decir "esta función no devuelve ningún valor". La segunda diferencia es que la instrucción *return* no contiene una expresión de un valor a devolver, ya que no se devuelve ningún valor. La sintaxis se resume en el cuadro 4.1.

llamada de función

Una llamada de función *void* es una instrucción ejecutable. Por ejemplo, la función `mostrar_resultados` anterior podría llamarse así:

```
mostrar_resultados(32.5, 0.3);
```

CUADRO 4.1 *Sintaxis de una definición de función* `void`



Si la instrucción anterior se ejecutara en un programa, haría que apareciera lo siguiente en la pantalla:

```
32.5 grados Fahrenheit equivale a
0.3 grados Celsius.
```

Observe que la llamada de función termina con un punto y coma, lo que le dice al compilador que se trata de una instrucción ejecutable.

Cuando se invoca una función `void`, los argumentos se sustituyen por los parámetros formales y se ejecutan las instrucciones del cuerpo de la función. Por ejemplo, una llamada a la función `void mostrar_resultados`, que presentamos antes, hace que se escriba algo en la pantalla. Una forma de visualizar una función `void` es imaginar que el cuerpo de la definición de función se copia en el programa en lugar de la llamada de función. Cuando se invoca la función, los argumentos se sustituyen por los parámetros formales, y de ahí en adelante es como si el cuerpo de la función fuera líneas del programa.

funciones sin argumentos

Es perfectamente válido, y a veces útil, tener una función sin argumentos. En tal caso simplemente no se enumeran parámetros formales en la declaración de la fun-

ción, y no se usan argumentos cuando se invoca la función. Por ejemplo, la función `void inicializar_pantalla`, que definimos a continuación, se limita a enviar un comando de nueva línea a la pantalla:

```
void inicializar_pantalla()
{
    using namespace std;
    cout << endl;
    return;
}
```

Si nuestro programa incluye la siguiente llamada a esta función como primera instrucción ejecutable, la salida del programa que se ejecutó previamente quedará separada de la salida de este programa:

```
inicializar_pantalla();
```

Observe que aun cuando una función no tiene parámetros es preciso incluir los paréntesis en la declaración de la función y en cualquier llamada a la función. El siguiente ejemplo de programación muestra estos dos ejemplos de funciones `void` en un programa completo.

EJEMPLO DE PROGRAMACIÓN
Conversión de temperaturas

El programa del cuadro 4.2 toma como entrada una temperatura en grados Fahrenheit y produce la temperatura equivalente en grados Celsius. Una temperatura Fahrenheit *F* se puede convertir en una Celsius *C* equivalente así:

$$C=(5/9)(F-32)$$

La función `celsius` que se muestra en el cuadro 4.2 usa esta fórmula para efectuar la conversión de temperatura.

Instrucciones `return` en funciones `void`

funciones `void` e instrucciones `return`

Tanto las funciones `void` como las que devuelven un valor pueden tener instrucciones `return`. En el caso de una función que devuelve un valor, la instrucción `return` especifica el valor devuelto. En el caso de una función `void`, la instrucción `return` simplemente termina la llamada de función. Como vimos en el capítulo anterior, toda función que devuelve un valor debe terminar con la ejecución de una instrucción `return`. Sin embargo, una función `void` no tiene que contener una instrucción `return`. Si una función `void` no contiene una instrucción `return`, terminará después de ejecutar el código del cuerpo de la función. Es como si hubiera una instrucción `return` implícita antes de la llave final `}` del cuerpo de la función. Por ejemplo, las funciones `inicializar_pantalla` y `mostrar_resultados` del cuadro 4.2 funcionarían exactamente igual si omitiéramos las instrucciones `return` en sus definiciones.

El hecho de que haya una instrucción `return` implícita antes de la llave final del cuerpo de una función no implica que nunca se necesite una instrucción `return` en una función `void`. Por ejemplo, la definición de función del cuadro 4.3 se podría usar como parte de un programa administrativo de un restaurante. Esa función despliega instrucciones para dividir una cantidad dada de helado entre los comensales de una mesa. Si no hay comensales (o sea, si `numero` es igual a `0`), la instrucción `return` dentro de la instrucción

CUADRO 4.2 *Funciones* `void` *(parte 1 de 2)*

```
//Programa para convertir una temperatura Fahrenheit a Celsius.
#include <iostream>

void inicializar_pantalla();
//Separa la salida actual de la
//salida del programa que se ejecutó antes.

double celsius(double fahrenheit);
//Convierte una temperatura Fahrenheit
//en una Celsius.

void mostrar_resultados(double grados_f, double grados_c);
//Muestra las salidas. Supone que grados_c
//Celsius equivale a grados_f Fahrenheit.

int main()
{
    using namespace std;
    double temperatura_f, temperatura_c;

    inicializar_pantalla();
    cout <<"Convertir una temperatura Fahrenheit"
         <<"en una Celsius.\n"
         <<"Introduzca una temperatura en grados Fahrenheit: ";
    cin >> temperatura_f;

    temperatura_c = Celsius(temperatura_f);

    mostrar_resultados(temperatura_f, temperatura_c);
    return 0;
}

//Esta definición utiliza iostream:
void inicializar_pantalla()
{
    using namespace std;
    cout << endl;
    return;
}
```

Esta instrucción `return` *es opcional.*

CUADRO 4.2 *Funciones* `void` *(parte 2 de 2)*

```
double celsius(double fahrenheit)
{
    return ((5.0/9.0)*(fahrenheit - 32));
}

//Esta definición utiliza iostream:
void mostrar_resultados(double grados_f, double grados_c)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
    cout << grados_f
         << " grados Fahrenheit equivalen a\n"
         << grados_c << " grados Celsius.\n";
    return;
}
```

Esta instrucción `return` *es opcional.*

Diálogo de ejemplo

```
Convertir una temperatura Fahrenheit en una Celsius.
Introduzca una temperatura en grados Fahrenheit: 32.5
32.5 grados Fahrenheit equivalen a
0.3 grados Celsius.
```

if termina la llamada de función y evita una división entre cero. Si `numero` no es `0`, la llamada de función termina cuando se ejecuta la última instrucción `cout` al final del cuerpo de la función.

La parte `main` de un programa es una función

Seguramente ya se dio cuenta de que la parte `main` de un programa es en realidad la definición de una función llamada `main`. Cuando el programa se ejecuta, se invoca automáticamente la función `main`, y ésta a su vez puede invocar otras funciones. Aunque podría parecer que la instrucción *return* de la parte `main` del programa debería ser opcional, oficialmente no lo es. Técnicamente, la parte `main` del programa es una función que devuelve un valor de tipo *int*, por lo que requiere una instrucción `return`. No obstante, la función `main` se usa como si fuera una función *void*. Tratar la parte `main` del programa como una función que devuelve un entero podría parecer absurdo,

CUADRO 4.3 *Uso de* `return` *en una función* `void`

Declaración de la función

```
void reparto_de_helado(int numero, double peso_total);
//Despliega instrucciones para repartir peso_total onzas de
//helado entre número clientes. Si número es 0, no se
//hace nada.
```

Definición de la función

```
//La definición utiliza iostream:
void reparto_de_helado(int numero, double peso_total)
{
    using namespace std;
    double porción;
    if (numero == 0)
        return;
    porcion = peso_total/numero;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Cada uno recibe "
         << porcion << " onzas de helado." << endl;
}
```

Si `numero` *es* 0 *la ejecución de la función termina aquí.*

pero es la tradición. Tal vez lo mejor sea seguir pensando en la parte `main` del programa simplemente como "la parte principal del programa" y no preocuparse por este pequeño detalle.[1]

Ejercicios de AUTOEVALUACIÓN

1. ¿Qué salidas produce el siguiente programa?

```
#include <iostream>
void amigable();
void timido(int num_oyentes);
```

[1] El estándar de C++ dice que se puede omitir el `return 0` en la parte *main*, pero muchos compiladores todavía lo requieren.

```
    int main()
    {
      using namespace std;
       amigable();
       timido(6);
       cout << "Una vez mas:\n";
       timido(2);
       amigable();
       cout << "Fin del programa.\n";
       return 0;
}
void amigable()
{
   using namespace std;
   cout << "Hola\n";
}
void timido(int num_oyentes)
{
   using namespace std;
   if (num_oyentes < 5)
        return;

   cout << "Adios\n";
}
```

2. ¿Es necesaria la instrucción *return* en una función *void*?

3. Suponga que omite la instrucción *return* en la definición de la función inicializar_pantalla del cuadro 4.2. ¿Qué efecto tendrá eso sobre el programa? ¿Se compilará el programa? ¿Se ejecutará? ¿Se comportará de forma diferente? ¿Qué pasará con la instrucción *return* de la definición de la función mostrar_resultados en ese mismo programa? ¿Qué efecto tendría sobre el programa omitir la instrucción *return* en la definición de mostrar_resultados? ¿Y la instrucción *return* de la definición de la función celsius en ese mismo programa? ¿Qué efecto tendría sobre el programa omitir la instrucción *return* en la definición de celsius?

4. Escriba una definición de una función *void* que tiene tres argumentos de tipo *int* y que despliega en la pantalla el producto de esos tres argumentos. Coloque la definición en un programa completo que lea tres números y luego llame a esa función.

5. ¿Su compilador permite *void* main() e *int* main()? ¿Qué advertencias se dan si se usa *int* main() y no se incluye una instrucción *return* 0;? Para averiguarlo, escriba varios programas de prueba pequeños y quizá pregunte también a su profesor o a un gurú local.

6. ¿Una llamada a una función *void* se utiliza como instrucción o como una expresión?

4.2 Parámetros de llamada por referencia

Cuando llamamos a una función, sus argumentos son sustituidos por los parámetros formales de la definición de la función o, para decirlo de forma menos formal, los argumentos "son insertados" en lugar de los parámetros formales. Existen diferentes mecanismos para este procedimiento de sustitución. El mecanismo que usamos en el capítulo 3, y hasta ahora en este capítulo, se denomina mecanismo de *llamada por valor*. El segundo mecanismo importante para sustituir argumentos se llama mecanismo de *llamada por referencia*.

Una primera mirada a la llamada por referencia

El mecanismo de llamada por valor que hemos usado hasta ahora no basta para ciertas subtareas. Por ejemplo, una subtarea común es obtener uno o más valores de entrada del usuario. Examine otra vez el programa del cuadro 4.2. Sus tareas se dividen en cuatro subtareas: inicializar la pantalla, obtener la temperatura Fahrenheit, calcular la temperatura Celsius correspondiente y exhibir los resultados. Tres de estas cuatro subtareas se implementan como funciones: `inicializar_pantalla`, `celsius` y `mostrar_resultados`. Sin embargo, la subtarea de obtener la entrada se implementa con las cuatro líneas de código siguientes (no con una llamada de función):

```
cout << "Convertiré una temperatura Fahrenheit"
     << " a Celsius.\n"
     << "Introduzca una temperatura en Fahrenheit: ";
cin >> temperatura_f;
```

La subtarea de obtener la entrada debería efectuarse con una llamada de función. Sin embargo, para hacerlo con una llamada de función necesitamos un parámetro de llamada por referencia.

Una función para obtener entradas debe hacer que los valores de una o más variables sean iguales a los valores tecleados, por lo que la llamada de función deberá tener una o más variables como argumentos y deberá modificar los valores de dichas variables. Con los parámetros formales de llamada por valor que hemos usado hasta ahora, un argumento correspondiente de una llamada de función puede ser una variable, pero la función sólo toma el valor de la variable y no modifica la variable en absoluto. Con un parámetro formal de llamada por valor sólo es sustituido *el valor* del argumento por el parámetro formal. En el caso de una función de entrada, queremos que *la variable* (no el valor de la variable) sea sustituida por el parámetro formal. El mecanismo de llamada por referencia funciona precisamente así. Con un parámetro formal de **llamada por referencia**, (también llamado simplemente **parámetro de referencia**) el argumento correspondiente de una llamada de función debe ser una variable y esta variable argumento es sustituida por el parámetro formal. Es como si la variable argumento se copiara literalmente en el cuerpo de la definición de función en lugar del parámetro formal. Una vez sustituido el argumento, el código del cuerpo de la función se ejecuta y puede modificar el valor de la variable argumento.

parámetro de referencia

Los parámetros de llamada por referencia se deben marcar de alguna manera para que el compilador pueda distinguirlos de los parámetros de llamada por valor. La forma de indicar un **parámetro de llamada por referencia** es anexar el **signo &** al final del nombre de tipo en la lista de parámetros formales, tanto en la declaración de la función como en el encabezado de su definición. Por ejemplo, la siguiente definición de función tiene un parámetro formal, `f_variable`, que es un parámetro de llamada por referencia:

&

```
void obtener_entrada(double& f_variable)
{
```

```
    using namespace std;
    cout << "Convertir una temperatura Fahrenheit"
         << " a Celsius.\n"
         << "Introduzca una temperatura en Fahrenheit: ";
    cin >> f_variable;
}
```

En un programa que contiene esta definición de función, la siguiente llamada hará que la variable `temperatura_f` sea igual a un valor leído del teclado:

```
obtener_entrada(temperatura_f);
```

Con esta definición de función, es fácil reescribir el programa del cuadro 4.2 de modo que la subtarea de leer la entrada se lleve a cabo con esta definición de función. Sin embargo, en lugar de reescribir un programa viejo, veamos un programa totalmente nuevo.

El cuadro 4.4 demuestra los parámetros de llamada por referencia. El programa no hace mucho; simplemente lee dos números y luego los escribe, pero en el orden inverso. Los parámetros de las funciones `obtener_numeros` e `intercambiar_valores` son parámetros de llamada por referencia. La entrada corre por cuenta de la llamada

```
obtener_numeros(primer_num, segundo_num);
```

Esta llamada de función asigna valores a las variables `primer_num` y `segundo_num`. Luego, la siguiente llamada de función intercambia los valores que están en las variables `primer_num` y `segundo_num`:

```
intercambiar_valores(primer_num, segundo_num);
```

En las subsecciones que siguen describiremos el mecanismo de llamada por referencia con mayor detalle y también explicaremos las funciones específicas que usamos en el cuadro 4.4.

Detalles de la llamada por referencia

En casi todas las situaciones el mecanismo de llamada por referencia funciona como si el nombre de la variable dada como argumento de la función sustituyera literalmente al parámetro formal de llamada por referencia. Sin embargo, el proceso es un poco más complicado. En algunas situaciones esas complicaciones son importantes, así que necesitamos examinar con mayor detalle este proceso de sustitución de llamada por referencia.

Recuerde que las variables de programa se implementan como posiciones de memoria. El compilador asigna una posición de memoria a cada variable. Por ejemplo, cuando el programa del cuadro 4.4 se compila, podría asignarse la posición 1010 a la variable `primer_num`, y la posición 1012 a `segundo_num`. Para fines prácticos, estas posiciones de memoria son las variables.

Por ejemplo, consideremos la siguiente declaración de función del cuadro 4.4:

```
void obtener_numeros(int& entrada1, int& entrada2);
```

Los parámetros formales de llamada por referencia `entrada1` y `entrada2` representan los argumentos reales que se usan en una llamada de función.

CUADRO 4.4 *Parámetros de llamada por referencia (parte 1 de 2)*

```
//Programa para demostrar parámetros de llamada por referencia.
#include <iostream>

void obtener_numeros(int& entrada1, int& entrada2);
//Lee dos enteros del teclado.

void intercambiar_valores(int& variable1, int& variable2);
//Intercambia los valores de variable1 y variable2.

void mostrar_resultados(int salida1, int salida2);
//Muestra los valores de variable1 y variable2, en ese orden.

int main ()
{
    int primer_num, segundo_num;

    obtener_numeros(primer_num, segundo_num);
    intercambiar_valores(primer_num, segundo_num);
    mostrar_resultados(primer_num, segundo_num);
    return 0;
}
//Usa iostream:
void obtener_numeros(int& entrada1, int& entrada2)
{
    using namespace std;
    cout << "Introduzca dos enteros: ";
    cin >> entrada1
        >> entrada2;
}
void intercambiar_valores(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
//Utiliza iostream:
void mostrar_resultados(int salida1, int salida2)
{
    using namespace std;
    cout << "En orden inverso los numeros son: "
         << salida1 << " " << salida2 << endl;
}
```

CUADRO 4.4 *Parámetros de llamada por referencia (parte 2 de 2)*

Diálogo de ejemplo

```
Teclea dos enteros: 5 10
En orden inverso los numeros son: 10 5
```

Llamada por referencia

Para hacer que un parámetro formal sea de **llamada por referencia**, anexamos el **signo &** a su nombre de tipo. El argumento correspondiente en una llamada a la función deberá ser entonces una variable, no una constante ni alguna otra expresión. Cuando la función se invoque, la variable argumento corespondiente (no su valor) sustituirá al parámetro formal. Cualquier cambio que se efectúe al parámetro formal en el cuerpo de la función se efectuará a la variable argumento cuando la función se invoque. Los detalles exactos de los mecanismos de sustitución se dan en el texto de este capítulo.

Ejemplo (de parámetros de llamada por referencia en una declaración de función):

```
void obtener_datos(int& primer_dato, double& segundo_dato);
```

Consideremos ahora una llamada de función como la que sigue, del mismo cuadro:

```
obtener_numeros(primer_num, segundo_num);
```

Cuando se ejecuta la llamada, la función no recibe los nombres de argumento `primer_num` y `segundo_num`. En vez de ello, recibe una lista de las posiciones de memoria asociadas a cada nombre. En este ejemplo, la lista consiste en las posiciones:

```
1010
1012
```

que son las posiciones asociadas a las variables argumento `primer_num` y `segundo_num` *en ese orden*. Son estas mismas posiciones de memoria las que se asocian a los parámetros formales. La primera posición de memoria se asocia al primer parámetro formal, la segunda posición de memoria se asocia al segundo parámetro formal, etcétera. De forma gráfica, la correspondencia en este caso es

```
primer_num  ——► 1010 ——► entrada1
segundo_num ——► 1012 ——► entrada2
```

Cuando se ejecutan las instrucciones de la función, lo que el cuerpo de la función diga que debe hacerse a un parámetro formal en realidad se hace a la variable que está en la posición de memoria asociada a ese parámetro formal. En este caso, las instrucciones del cuerpo de la función `obtener_numeros` dicen que se debe almacenar un valor en el parámetro

formal `entrada1` usando una instrucción `cin`, así que ese valor se almacena en la variable que está en la posición de memoria 1010 (y da la casualidad que ésa es la variable `primer_num`). Así mismo, las instrucciones del cuerpo de la función `obtener_numeros` dicen que a continuación se debe almacenar un valor en el parámetro formal `entrada2` usando una instrucción `cin`, por lo que ese valor se almacena en la variable que está en la posición de memoria 1012 (que es la variable `segundo_num`). Así pues, lo que sea que la función ordene a la computadora hacer a `entrada1` y `entrada2`, se hará realmente a las variables `primer_num` y `segundo_num`. En el cuadro 4.5 se describen estos detalles del funcionamiento del mecanismo de llamada por referencia en esta llamada a `obtener_numeros`.

CUADRO 4.5 *Comportamiento de argumentos de llamada por referencia (parte 1 de 2)*

Anatomía de una llamada de función del cuadro 4.4 usando argumentos de llamada por referencia

0. Suponga que el compilador asignó las siguientes direcciones de memoria a las variables `primer_num` y `segundo_num`:

```
primer_num  ----------> 1010
segundo_num ----------> 1012
```

(No sabemos qué direcciones se asignaron y los resultados no dependen de las direcciones reales, pero esto hará al proceso más concreto y tal vez más fácil de entender.)

1. En el programa del cuadro 4.4 comienza a ejecutarse la siguiente llamada de función:

```
obtener_numeros(primer_num, segundo_num);
```

2. Se le dice a la función que use la posición de memoria de la variable `primer_num` en lugar del parámetro formal `entrada1`, y la posición de memoria de `segundo_num` en lugar del parámetro formal `entrada2`. El efecto es el mismo que se obtendría si la definición de función se reescribiera como sigue (no es código C++ válido, pero nosotros lo entendemos claramente):

```
void obtener_numeros(int& <la variable en la posición de memoria 1010>,
                     int& <la variable en la posición de memoria 1012>)
{
      using namespace std;
      cout << "Introduzca dos enteros: ";
      cin >> <la variable en la posición de memoria 1010>
          >> <la variable en la posición de memoria 1012>;
}
```

CUADRO 4.5 *Comportamiento de argumentos de llamada por referencia (parte 2 de 2)*

Anatomía de una llamada de función del cuadro 4.4 (*conclusión*)

Puesto que las variables que están en las posiciones 1010 y 1012 son `primer_num` y `segundo_num`, el efecto es el mismo que si la definición de función se reescribiera así:

```
void obtener_numeros(int& primer_num, int& segundo_num)
{
    using namespace std;
    cout << "Introduzca dos enteros: ";
    cin >> primer_num
        >> segundo_num;
}
```

3. Se ejecuta el cuerpo de la función. El efecto es el mismo que si se ejecutara lo siguiente:

```
{
    using namespace std;
    cout << "Introduzca dos enteros: ";
    cin >> primer_num
        >> segundo_num;
}
```

4. Cuando se ejecuta la instrucción `cin`, se asignan a las variables `primer_num` y `segundo_num` los valores introducidos. (Si el diálogo es como el que se muestra en el cuadro 4.4, el valor de `primer_num` se vuelve `5` y el valor de `segundo_num` se vuelve `10`.)

5. Cuando la llamada de función termina, las variables `primer_num` y `segundo_num` conservan los números que les dio la instrucción `cin` del cuerpo de la función. (Si el diálogo es como el que se muestra en el cuadro 4.4, el valor de `primer_num` es `5` y el valor de `segundo_num` es `10` al terminar la llamada de función.)

Podría parecer que hay un nivel adicional de detalle, o al menos un nivel adicional de explicación. Si `primer_num` es la variable que está en la posición de memoria 1010, ¿por qué insistimos en decir "la variable que está en la posición de memoria 1010" en lugar de decir simplemente "`primer_num`"? Se requiere este nivel adicional de detalle si los argumentos y parámetros formales tienen alguna coincidencia de nombres que pueda causar confusión. Por ejemplo, los parámetros formales de la función `obtener_numeros` se llaman `entrada1` y `entrada2`. Suponga que quiere modificar el programa del cuadro 4.4 de modo que use la

función `obtener_numeros` con argumentos que también se llaman `entrada1` y `entrada2`, y suponga que quiere hacer algo que no sea nada obvio; por ejemplo, que el primer número que se teclea se guarde en una variable llamada `entrada2` y el segundo se guarde en la variable llamada `entrada1`, tal vez porque el segundo número se va a procesar primero o porque es el más importante. Supongamos ahora que a las variables `entrada1` y `entrada2`, que se declaran en la parte `main` del programa se les ha asignado las posiciones de memoria 1014 y 1016. La llamada de función podría ser la siguiente:

```
int entrada1, entrada2;
obtener_numeros(entrada2, entrada1);
```

Observe el orden de los argumentos.

En este caso, si decimos "`entrada1`", no sabemos si nos referimos a la variable llamada `entrada1` que se declaró en la parte `main` del programa o al parámetro formal `entrada1`. Pero si a la variable `entrada1` declarada en la parte `main` del programa se le asigna la posición de memoria 1014, la frase "la variable que está en la posición de memoria 1014" no es ambigua. Repasemos los detalles del mecanismo de sustitución en este caso.

En la llamada anterior, el argumento que corresponde al parámetro formal `entrada1` es la variable `entrada2`, y el argumento que corresponde al parámetro formal `entrada2` es la variable `entrada1`. Esto podría confundirnos, pero no representa ningún problema para la computadora, porque ella nunca "sustituye `entrada2` por `entrada1`" ni "sustituye `entrada1` por `entrada2`". La computadora simplemente maneja posiciones de memoria. La computadora sustituye el parámetro formal `entrada1` por "la variable que está en la posición de memoria 1016", y el parámetro formal `entrada2` por "la variable que está en la posición de memoria 1014".

EJEMPLO DE PROGRAMACIÓN

La función `intercambiar_valores`

La función `intercambiar_valores` que definimos en el cuadro 4.4 intercambia los valores almacenados en dos variables. La descripción de la función está dada por la siguiente declaración y su comentario acompañante:

```
void intercambiar_valores(int& variable1, int& variable2);
//Intercambia los valores de variable1 y variable2.
```

Para ver cómo debe operar la función, supondremos que la variable `primer_num` tiene el valor `5` y la variable `segundo_num` tiene el valor `10`, y consideramos la siguiente llamada de función:

```
intercambiar_valores(primer_num, segundo_num);
```

Después de esta llamada de función, el valor de `primer_num` será `10` y el valor de `segundo_num` será `5`.

Como se muestra en el cuadro 4.4, la definición de la función `intercambiar_valores` usa una variable local llamada `temp`. Esta variable local es necesaria. Podríamos pensar que la definición de la función se puede simplificar a lo que sigue:

```
void intercambiar_valores(int& variable1, int& variable2)
{
   variable1 = variable2;
   variable2 = variable1;
}
```

¡*Esto no funciona*!

Para ver que esta definición no puede funcionar, consideremos lo que sucedería con esa definición y la llamada

```
intercambiar_valores(primer_num, segundo_num);
```

Las variables `primer_num` y `segundo_num` sustituyen a los parámetros formales `variable1` y `variable2`, así que, con esta definición de función incorrecta, la llamada de función equivale a lo siguiente:

```
primer_num = segundo_num;
segundo_num = primer_num;
```

Este código no produce el resultado deseado. El valor de `primer_num` se hace igual al de `segundo_num`, como debía ser, pero luego el valor de `segundo_num` se hace igual al nuevo valor de `primer_num`, que ahora es el valor original de `segundo_num`. Así pues, el valor de `segundo_num` no habrá cambiado. (Si esto no le queda claro, siga los pasos con valores específicos para las variables `primer_num` y `segundo_num`.) Lo que la función necesita hacer es guardar el valor original de `primer_num` para que ese valor no se pierda. Para esto se usa la variable local `temp` en la definición correcta de la función. Esa definición es la que aparece en el cuadro 4.4. Cuando se usa esa versión correcta y la función se invoca con los argumentos `primer_num` y `segundo_num`, la llamada de función equivale al siguiente código, que funciona correctamente:

```
temp = primer_num;
primer_num = segundo_num;
segundo_num = temp;
```

Parámetros y argumentos

Los diferentes términos relacionados con parámetros y argumentos pueden causar confusión. Sin embargo, si tenemos presentes unas cuantas cuestiones sencillas, podremos manejar fácilmente estos términos:

1. Los **parámetros formales** de una función aparecen en la declaración de la función y se usan en el cuerpo de la definición de la función. Un parámetro formal (de cualquier clase) es una especie de hueco o "marcador de posición" que se llena con algo cuando la función se invoca.
2. Un **argumento** es algo que se usa para "llenar el hueco" de un parámetro formal. Cuando se escribe una llamada de función, los argumentos se enumeran entre paréntesis después del nombre de la función. Cuando se ejecuta la llamada de función, los argumentos se "insertan" en los lugares donde aparecen los parámetros formales.
3. Los términos *llamada por valor* y *llamada por referencia* se refieren al mecanismo que se usa en el proceso de "inserción". Con el método de **llamada por valor**, sólo se usa el valor del argumento. En este mecanismo de llamada por valor el parámetro formal es una variable local que se inicializa con el valor del argumento correspondiente. Con el mecanismo de **llamada por referencia**, el argumento es una variable y se usa toda la variable. Esta variable argumento sustituye al parámetro formal, de modo que cualquier cambio que sufra el parámetro formal lo sufrirá realmente la variable argumento.

Listas de parámetros mixtas

Lo que determina si un parámetro formal es de llamada por valor o de llamada por referencia es la ausencia o presencia de un signo & después de su especificación de tipo. Si el & está presente, el parámetro formal es de llamada por referencia; si no, es de llamada por valor.

mezcla de llamada por referencia y llamada por valor

Es perfectamente válido mezclar parámetros formales de llamada por valor y llamada por referencia en la misma función. Por ejemplo, el primero y el último de los parámetros formales de la siguiente declaración de función son de llamada por referencia, y el de en medio es de llamada por valor:

```
void cosa_buena(int& par1, int par2, double& par3);
```

Los parámetros de llamada por referencia no están limitados a las funciones *void*; también podemos usarlos en funciones que devuelven un valor. Así pues, una función que tiene un parámetro de llamada por referencia puede cambiar el valor de una variable dada como argumento y también devolver un valor.

TIP DE PROGRAMACIÓN

Qué clase de parámetros debemos usar

El cuadro 4.6 ilustra las diferencias entre la manera en que el compilador trata los parámetros formales de llamada por valor y llamada por referencia. Dentro del cuerpo de la función, se asigna un valor a los parámetros `par1_val` y `par2_ref`, pero como son de diferente especie el efecto es distinto en los dos casos.

`par1_val` es un parámetro de llamada por valor, así que es una variable local. Cuando se emite la siguiente llamada:

```
hacer_cosas(n1, n2);
```

la variable local `par1_val` se inicializa con el valor de `n1`. Es decir, la variable local se inicializa con `1` y la función hace caso omiso de la variable `n1`. Como puede verse en el diálogo de ejemplo, en el cuerpo de la función se cambia a `111` el valor del parámetro formal `par1_val` (que es una variable local), y este valor se despliega en la pantalla. Sin embargo, el valor del argumento `n1` no cambia. Como se aprecia en el diálogo de ejemplo, `n1` conserva el valor `1`.

En cambio, `par2_ref` es un parámetro de llamada por referencia. Cuando se invoca la función, la variable argumento `n2` (no sólo su valor) sustituye al parámetro formal `par2_ref`, de modo que cuando se ejecuta el siguiente código:

```
par2_ref = 222;
```

es como si se ejecutara lo siguiente:

```
n2 = 222;
```

Así, el valor de la variable `n2` cambia cuando se ejecuta el cuerpo de la función. Como muestra el diálogo, la llamada de función cambió el valor de `n2` de `2` a `222`.

CUADRO 4.6 *Comparación de mecanismo de argumentos*

```
//Ilustra la diferencia entre un parámetro de llamada
//por valor y un parámetro de llamada por referencia.
#include <iostream>

void hacer_cosas(int par1_val, int& par2_ref);
//par1_val es un parámetro formal de llamada por valor y
//par2_ref es un parámetro formal de llamada por referencia.

int main()
{
    using namespace std;
    int n1, n2;

    n1 = 1;
    n2 = 2;
    hacer_cosas(n1, n2);
    cout << "n1 despues de la llamada = " << n1 << endl;
    cout << "n2 despues de la llamada = " << n2 << endl;
    return 0;
}
void hacer_cosas(int par1_val, int& par2_ref)
{
    using namespace std;
    par1_val = 111;
    cout << "par1_val en la llamada = "
         << par1_val << endl;
    par2_ref = 222;
    cout << "par2_ref en la llamada = "
         << par2_ref << endl;
}
```

Diálogo de ejemplo

```
par1_val en la llamada = 111
par2_ref en la llamada = 222
n1 despues de la llamada = 1
n2 despues de la llamada = 222
```

Si tenemos presente la lección del cuadro 4.6, es fácil decidir cuál mecanismo de parámetros debemos usar. Si queremos que una función cambie el valor de una variable, el parámetro formal correspondiente debe ser de llamada por referencia y debe marcarse con el signo `&`. En todos los demás casos, podemos usar un parámetro formal de llamada por valor.

RIESGO *Variables locales inadvertidas*

Si queremos que una función modifique el valor de una variable, el parámetro formal correspondiente debe ser de llamada por referencia y por tanto su tipo debe ir seguido del signo `&`. Si por descuido omitimos el signo `&`, la función tendrá un parámetro de llamada por valor donde queríamos que tuviera un parámetro de llamada por referencia, y cuando el programa se ejecute descubriremos que la llamada de función no modifica el valor del argumento correspondiente. La razón es que un parámetro formal de llamada por valor es una variable local, y si su valor cambia dentro de la función dicho cambio no surtirá efecto alguno fuera del cuerpo de la función, como sucede con cualquier variable local. Éste es un error de lógica que puede ser muy difícil de detectar porque *no parece* error.

Por ejemplo, el programa del cuadro 4.7 es idéntico al del cuadro 4.4, excepto que por error se omitieron los signos `&` en la función `intercambiar_valores`. El resultado es que los parámetros formales `variable1` y `variable2` son variables locales. Las *variables* argumento `primer_num` y `segundo_num` nunca sustituyen a `variable1` y `variable2`; en vez de ello, `variable1` y `variable2` se inicializan con *los valores de* `primer_num` y `segundo_num`. Luego se intercambian los valores de `variable1` y `variable2`, pero los valores de `primer_num` y `segundo_num` no cambian. La omisión de dos signos `&` ha hecho que el programa sea totalmente erróneo, pero se ve casi idéntico al programa correcto y se compila y ejecuta sin mensajes de error.

Ejercicios de AUTOEVALUACIÓN

7. ¿Qué salidas produce el siguiente programa?

```
#include <iostream>
void hacer_calculo(int& x, int y, int& z);
int main()
{
     using namespace std;
     int a, b, c;
     a = 10;
     b = 20;
     c = 30;
     hacer_calculo(a, b, c);
     cout << a << " " << b << " " << c;
     return 0;
}
```

CUADRO 4.7 *Variable local inadvertida*

```
//Programa para demostrar parámetros de llamada por referencia.
#include <iostream>
void obtener_numeros(int& entrada1, int& entrada2);
//Lee dos enteros del teclado.
void intercambiar_valores(int variable1, int variable2);
//Intercambia los valores de variable1 y variable2.
void mostrar_resultados(int salida1, int salida2);
//Muestra los valores de variable1 y variable2, en ese orden.
int main()
{
     int primer_num, segundo_num;

     obtener_numeros(primer_num, segundo_num);
     intercambiar_valores(primer_num, segundo_num);
     mostrar_resultados(primer_num, segundo_num);
     return 0;
}
void intercambiar_valores(int variable1, int variable2)
{
     int temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

olvidamos los & aquí

olvidamos los & aquí

variables locales inadvertidas

<Las definiciones de obtener_numeros y
mostrar_resultados son las mismas que en el cuadro 4.4.>

Diálogo de ejemplo

```
Teclea dos enteros: 5 10
En orden inverso los numeros son: 5 10
```

```
void hacer_calculo(int& x, int y, int& z)
{
    using namespace std;
    cout << x << " " << y << " " << z << endl;
    x = 1;
    y = 2;
    z = 3;
    cout << x << " " << y << " " << z << endl;
}
```

8. ¿Qué salidas tendría el programa del cuadro 4.4 si omitimos los signos `&` del primer parámetro en la declaración y en el encabezado de la función `intercambiar_valores`? El `&` no se omite del segundo parámetro.

9. ¿Qué salidas tendría el programa del cuadro 4.6 si cambiamos la declaración de la función `hacer_cosas` a lo que se muestra a continuación y modificamos igualmente el encabezado de la función, de modo que el parámetro formal `par2_ref` sea un parámetro de llamada por valor?

   ```
   void hacer_cosas(int par1_val, int par2_ref);
   ```

10. Escriba una definición de función *`void`* para una función llamada `cero_ambos` que tiene dos parámetros de llamada por referencia que son variables de tipo `int`, y que asigna el valor `0` a ambas variables.

11. Escriba una definición de una función *`void`* llamada `sumar_impuesto` que tiene dos parámetros formales: `tasa_impuesto`, que es el impuesto al valor agregado expresado como un porcentaje, y `costo`, que es el costo de un artículo antes de impuestos. La función modifica el valor de `costo` de modo que incluya el I.V.A.

12. ¿Podría una función que devuelve un valor tener un parametro de llamada por referencia? ¿Podría una función tener tanto un parametro de llamada por valor como un parámetro de llamada por referencia?

4.3 Uso de abstracción de procedimientos

Mi memoria es tan mala,
que a menudo olvido mi propio nombre.

Miguel de Cervantes Saavedra, *Don Quijote*

Recuerde que el principio de abstracción de procedimientos dice que las funciones deben diseñarse de modo que puedan usarse como cajas negras. Para que un programador pueda usar una función de forma eficaz, lo único que tendría que conocer es la declaración de la función y el comentario acompañante que le dice qué hace la función. No debe ser necesario que el programador conozca los detalles del cuerpo de la función. En esta sección trataremos con mayor detalle varios temas relacionados con este principio.

Funciones que llaman a funciones

Un cuerpo de función puede contener una llamada a otra función. La situación en este tipo de llamadas es exactamente la misma que si la llamada de función se hubiera efectuado en la función `main` del programa; la única restricción es que la declaración de la función

debe aparecer antes de que se utilice la función. Si organizamos los programas como lo hemos venido haciendo, esto sucederá automáticamente, ya que todas las declaraciones de función aparecen antes de la función `main` y todas las definiciones de función aparecen después de la función `main`. Aunque podemos incluir una *llamada* de función dentro de la definición de otra función, no podemos colocar la *definición* de una función dentro del cuerpo de otra definición de función.

El cuadro 4.8 muestra una versión mejorada del programa del cuadro 4.4. Ese programa siempre invertía los valores de las variables `primer_num` y `segundo_num`. El programa del cuadro 4.8 invierte dichas variables sólo en algunas ocasiones. Este programa utiliza la función `ordenar` para reacomodar los valores de estas variables de modo que

```
primer_num <= segundo_num
```

Si esta condición ya se cumple, nada se hace a las variables `primer_num` y `segundo_num`. En cambio, si `primer_num` es mayor que `segundo_num`, se llama a la función `intercambiar_valores` para que intercambie los valores de estas dos variables. Esta prueba para ver si los números están ordenados, y el intercambio, se efectúan dentro del cuerpo de la función `ordenar`. Por tanto, la función `intercambiar_valores` se invoca dentro del cuerpo de la función `ordenar`. Esto no presenta problemas especiales. Si usamos el principio de la abstracción de procedimientos, la función `intercambiar_valores` sólo es una cosa que realiza una acción (intercambiar los valores de dos variables); esta acción es la misma donde ocurra.

Precondiciones y postcondiciones

precondición

postcondición

Una buena forma de escribir un comentario de declaración de función es desglosarlo en dos tipos de información llamados *precondición* y *postcondición*. La **precondición** dice qué se supone que es verdad cuando la función se llama. La función no puede invocarse, y no puede esperarse que opere correctamente, si la precondición no se cumple. La **postcondición** describe el efecto de la llamada de función. Es decir, la postcondición dice qué será verdad después de que la función se ejecute en una situación en la que se cumpla la precondición. En el caso de una función que devuelve un valor, la postcondición describirá el valor devuelto por la función. En el caso de una función que modifica el valor de algunas variables argumento, la postcondición describirá todos los cambios que sufren los valores de los argumentos.

Por ejemplo, el comentario de la declaración de función `intercambiar_valores` que aparece en el cuadro 4.8 puede tener el siguiente formato:

```
void intercambiar_valores(int& variable1, int& variable2);
//Precondición: variable 1 y variable2 han recibido
//valores.
//Postcondición: Los valores de variable1 y variable2
//se han intercambiado.
```

Podemos dar el mismo formato al comentario de la función `celsius` del cuadro 4.2, así:

```
double celsius(double fahrenheit);
//Precondición: fahrenheit es una temperatura expresada
//en grados Fahrenheit.
//Postcondición: Devuelve la temperatura equivalente
//expresada en grados Celsius.
```

CUADRO 4.8 *Función que llama a otra función (parte 1 de 2)*

```
//Programa para demostrar una función que llama a otra.
#include <iostream>

void obtener_entradas(int& entrada1, int& entrada2);
//Lee dos enteros del teclado.

void intercambiar_valores(int& variable1, int& variable2);
//Intercambia los valores de variable1 y variable2.

void ordenar(int& n1, int& n2);
//Ordena los números que están en las variables n1 y n2
//de modo que después de la llamada de función n1 <= n2.

void dar_resultados(int salida1, int salida2);
//Muestra los valores que están en salida1 y salida2.
//Supone que salida1 <= salida2.

int main()
{
    int primer_num, segundo_num;
    obtener_entradas(primer_num, segundo_num);
    ordenar(primer_num, segundo_num);
    dar_resultados(primer_num, segundo_num);
    return 0;
}

//Usa iostream:
void obtener_entradas(int& entrada1, int& entrada2)
{
    using namespace std;
    cout << "Introduzca dos enteros: ";
    cin >> entrada1 >> entrada2;
}
```

CUADRO 4.8 *Función que llama a otra función (parte 2 de 2)*

```
void intercambiar_valores(int& variable1, int& variable2)
{
     int temp;

     temp = variable1;
     variable1 = variable2;
     variable2 = temp;
}
```

```
void ordenar(int& n1, int& n2)
{
     if(n1 > n2)
        intercambiar_valores(n1, n2);
}
```

Estas definiciones de función pueden estar en cualquier orden.

```
//Usa iostream:
void dar_resultados(int salida1, int salida2)
{
     using namespace std;
     cout << "En orden creciente los numeros son: "
          << salida1 << " " << salida2 << endl;
}
```

Diálogo de ejemplo

```
Introduzca dos enteros: 10 5
En orden creciente los numeros son: 5 10
```

Cuando la única postcondición es una descripción del valor devuelto, los programadores suelen omitir la palabra *postcondición*. Otra forma aceptable del comentario de la declaración de función anterior sería:

```
//Precondición: fahrenheit es una temperatura expresada
//en grados Fahrenheit.
//Devuelve la temperatura equivalente expresada en
//grados Celsius.
```

Otro ejemplo de precondiciones y postcondiciones está dado por la siguiente declaración de función:

```
void acumular_intereses(double& saldo, double tasa);
//Precondición: saldo es un saldo de cuenta de ahorros
//no negativo.
//tasa es la tasa de interés expresada como porcentaje,
//por ejemplo 5 para indicar 5%.
//Postcondición: Se ha incrementado el valor de saldo
//en tasa porciento.
```

No es necesario conocer la definición de la función `acumular_intereses` para usar esta función. Por ello, sólo dimos la declaración de función y el comentario acompañante.

Las precondiciones y postcondiciones son algo más que una forma de resumir las acciones de una función: deben ser el primer paso del diseño y escritura de una función. Cuando diseñamos un programa, debemos especificar lo que cada función hace antes de comenzar a diseñar la forma en que las funciones lo harán. En particular, los comentarios de la declaración de función y la declaración de función deben diseñarse y escribirse antes de comenzar a diseñar el cuerpo de la función. Si después descubrimos que nuestra especificación no puede implementarse de alguna forma razonable, quizá tengamos que volver atrás y pensar mejor lo que la función debe hacer; pero si especificamos claramente lo que creemos que la función debe hacer minimizaremos los errores de diseño y el tiempo desperdiciado en escribir código que no realiza la tarea que se requiere.

Algunos programadores prefieren no utilizar las palabra precondición y postcondición en sus comentarios de función. Sin embargo, utilice o no las palabras, los comentarios de función siempre deberán contener información de la precondición y postcondición.

CASO DE ESTUDIO Precios en el supermercado

Este caso de estudio resuelve una tarea de programación muy sencilla. Podría parecer que contiene más detalles de los necesarios para una tarea tan simple; sin embargo, si examinamos los elementos del diseño en un contexto sencillo, podremos concentrarnos en aprenderlos sin distraernos con cuestiones secundarias. Una vez que aprendamos las técnicas que se ilustran en este sencillo caso de estudio, podremos aplicarlas a tareas de programación mucho más complicadas.

Definición del problema

La cadena de supermercados Compra Ágil nos ha encargado escribir un programa que determine el precio al detalle de un artículo a partir de las entradas apropiadas. Su política de precios es que cualquier artículo que se espera se venda en una semana o menos debe llevar un sobreprecio del 5% respecto al precio al mayoreo, y cualquier artículo que se espera permanezca en anaqueles durante más de una semana lleva un sobreprecio del 10%. Tenga presente que el sobreprecio bajo de 5% se usa para siete días o menos, y que con ocho días o más el sobreprecio cambia al 10%. Es importante indicar con precisión cuándo el programa debe cambiar de una forma de cálculo a la otra.

Como siempre, debemos asegurarnos de tener un planteamiento claro de las entradas requeridas y las salidas producidas por el programa.

Entradas

Las entradas consistirán en el precio al mayoreo de un artículo y el número de días que se espera tarde en venderse el artículo.

Salidas

Las salidas darán el precio al detalle del artículo.

Análisis del problema

Al igual que muchas tareas de programación sencillas, ésta se divide en tres subtareas principales:

1. Introducir los datos.
2. Calcular el precio al detalle del artículo.
3. Exhibir los resultados.

Estas tres subtareas se implementarán con tres funciones. Esas funciones se describen con sus declaraciones de función y comentarios acompañantes, que se dan a continuación. Observe que sólo son parámetros de llamada por referencia las cosas que las funciones modifican. Los demás parámetros formales son de llamada por valor.

```
void obtener_entradas(double& costo, int& recambio);
//Precondición: El usuario está listo para introducir
//los valores correctos
//Postcondición: costo contiene el costo al mayoreo de un artículo,
//recambio contiene el número de días que se espera que el
//artículo tarde en venderse.

double precio(double costo, int recambio);
//Precondición: costo es el costo al mayoreo de un artículo.
//recambio es el número de días que creemos tardará en venderse.
//Devuelve el precio al detalle del artículo.

void dar_salidas(double costo, int recambio, double precio);
//Precondición: costo es el costo al mayoreo de un artículo,
//recambio es el número de días que creemos tardará en venderse;
//precio es el precio al detalle del artículo.
//Postcondición: Se han escrito en la pantalla los valores de
//costo, recambio y precio.
```

Ahora que tenemos los encabezados de función, resulta trivial escribir la parte `main` de nuestro programa:

```
int main()
{
     double costo_mayoreo, costo_detalle;
     int tiempo_en_anaquel;

     obtener_entradas(costo_mayoreo, tiempo_en_anaquel);
     precio_detalle = precio(costo_mayoreo, tiempo_en_anaquel);
    dar_salidas(costo_mayoreo, tiempo_en_anaquel, precio_detalle);
     return 0;
}
```

Aunque todavía no hemos escrito los cuerpos de las funciones y no tenemos idea de cómo operan éstas, podemos escribir el código anterior que usa las funciones. Éste es el significado del principio de la abstracción de procedimientos. Las funciones se tratan como cajas negras.

Diseño de algoritmos

Las implementaciones de las funciones `obtener_entradas` y `dar_entradas` son obvias: consisten en unas cuantas instrucciones `cin` y `cout`. El algoritmo de la función `precio` está dado por el siguiente pseudocódigo:

```
if recambio ≤ 7 días entonces
   return (costo + 5% de costo);
else
   return (costo + 10% de costo);
```

Codificación

Hay tres constantes que se usan en este programa: un sobreprecio bajo del 5%, un sobreprecio alto del 10% y una estancia en anaqueles esperada de 7 días como umbral arriba del cual se usa el sobreprecio alto. Puesto que dichas constantes podrían tener que modificarse para actualizar el programa si la compañía decide cambiar su política de precios, declaramos constantes globales con nombre al principio de nuestro programa para cada una de estas tres cifras. Las declaraciones con el modificador `const` son las siguientes:

```
const double SOBREPRECIO_BAJO = 0.05; //5%
const double SOBREPRECIO_ALTO = 0.10; //10%
const int UMBRAL = 7; //Usar SOBREPRECIO_ALTO si no se espera
                      //que se venda en 7 días o menos.
```

El cuerpo de la función `precio` es una traducción directa de nuestro algoritmo, de pseudocódigo a código C++:

```
{
    if (recambio <= UMBRAL)
       return (costo + (SOBREPRECIO_BAJO * costo) );
    else
       return (costo + SOBREPRECIO_ALTO * costo) );
}
```

El programa completo se muestra en el cuadro 4.9.

Prueba del programa

probar todo tipo de entradas

Una técnica importante de prueba de programas consiste en probar toda especie de entradas. No existe una definición precisa de lo que queremos decir con una "especie" de entrada, pero en la práctica suele ser fácil decidir qué clase de datos de entrada maneja un programa. En el caso de nuestro programa de supermercado, hay dos tipos de entradas principales: las que usan el sobreprecio bajo de 5% y las que usan el sobreprecio alto de 10%. Así pues, deberemos probar por lo menos un caso en el que se espere que el artículo permanecerá en anaqueles menos de siete días y al menos un caso en el que se espere que permanezca ahí más de siete días.

CUADRO 4.9 *Precios de supermercado (parte 1 de 3)*

```
//Determina el precio al detalle de un artículo según las
//políticas de precios de la cadena de supermercados Compra Ágil.
#include <iostream>

const double SOBREPRECIO_BAJO = 0.05; //5%
const double SOBREPRECIO_ALTO = 0.10; //10%
const int UMBRAL = 7; //Usar SOBREPRECIO_ALTO si no se espera
                      //que se venda en 7 días o menos.
void introduccion();
//Postcondición: Se describe el programa en la pantalla.

void obtener_entradas(double& costo, int& recambio);
//Precondición: El usuario está listo para introducir correctamente los valores correctos
//Postcondición: costo contiene el costo al mayoreo
//de un artículo, recambio contiene el número de días
//que se espera que el artículo tarde en venderse.

double precio(double costo, int recambio);
//Precondición: costo es el costo al mayoreo de un artículo.
//recambio es el número de días que creemos tardará en venderse.
//Devuelve el precio al detalle del artículo.

void dar_salidas(double costo, int recambio, double precio);
//Precondición: costo es el costo al mayoreo de un artículo, recambio es el número
//de días que creemos tardará en venderse; precio es el precio al detalle del artículo.
//Postcondición: Se han escrito en la pantalla los valores de costo,
//recambio y precio.

int main()
{
    double costo_mayoreo, costo_detalle;
    int tiempo_en_anaquel;

    introduccion();
    obtener_entradas(costo_mayoreo, tiempo_en_anaquel);
    costo_detalle = precio(costo_mayoreo, tiempo_en_anaquel);
    dar_salidas(costo_mayoreo, tiempo_en_anaquel, costo_detalle);
    return 0;
}
```

CUADRO 4.9 *Precios de supermercado (parte 2 de 3)*

```
//Usa iostream:
void introduccion()
{
    using namespace std;
    cout << "Este programa determina el precio al detalle de\n"
         << "un articulo en un supermercado de la cadena Compra Agil.\n";
}
//Usa iostream:
void obtener_entradas(double& costo, int& recambio)
{
    using namespace std;
    cout << "Introduzca el costo al mayoreo del articulo: $";
    cin >> costo;
    cout << "Introduzca el numero de dias que se cree tardara en venderse: ";
    cin >> recambio;
}
//Usa iostream:
void dar_salidas(double costo, int recambio, double precio)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout<< "Costo al mayoreo = $" << costo << endl
        << "Tiempo que se cree tardara en venderse = "
        << recambio << " dias" << endl
        << "Precio al detalle = $" << precio << endl;
}

//Usa constantes definidas SOBREPRECIO_BAJO, SOBREPRECIO_ALTO y UMBRAL:
double precio(double costo, int recambio)
{
    if (recambio <= UMBRAL)
        return (costo + (SOBREPRECIO_BAJO * costo) );
    else
        return (costo + (SOBREPRECIO_ALTO * costo) );
}
```

CUADRO 4.9 *Precios de supermercado (parte 3 de 3)*

Diálogo de ejemplo

```
Este programa determina el precio al detalle de
un articulo en un supermercado de la cadena Compra Agil.
Introduzca el costo al mayoreo del articulo: $1.21
Introduzca el numero de dias que se cree tardara en venderse: 5
Costo al mayoreo = $1.21
Tiempo que se cree tardara en venderse = 5 dias
Precio al detalle = $1.27
```

probar valores de límite

Otra estrategia de prueba consiste en probar *valores de límite*. Lamentablemente, éste es otro concepto poco preciso. Un valor de entrada (de prueba) es un **valor de límite** si es un valor en el que el programa cambia de comportamiento. Por ejemplo, en nuestro programa de supermercado el comportamiento del programa cambia en una permanencia esperada en anaqueles de siete días. Por tanto, 7 es un valor de límite; el programa se comporta de distinta forma con un número de días de 7 o menos, que con un número de días mayor que 7. Por consiguiente, debemos probar el programa con al menos un caso en el que se espere que el artículo permanezca en anaqueles exactamente siete días. Normalmente, también deberíamos probar entradas que están un paso más allá del valor de límite, pues bien podríamos tener un error de 1 respecto a dónde está el límite. Por ello, deberemos probar nuestro programa con entradas para un artículo que esperamos que permanezca en anaqueles seis días, uno que esperamos que permanezca siete días y uno que esperamos que permanezca ocho días. (Esto es además de las entradas de prueba que describimos en el párrafo anterior, que deberían estar muy por abajo y muy por arriba de siete días.)

Ejercicios de AUTOEVALUACIÓN

13. ¿Puede una definición de función aparecer dentro del cuerpo de otra definición de función?

14. ¿Puede una definición de función contener una llamada a otra función?

15. Reescriba el comentario de declaración de función para la función `ordenar` que se muestra en el cuadro 4.8 de modo que se exprese en términos de precondiciones y postcondiciones.

16. Dé una precondición y una postcondición para la función predefinida `sqrt`, que devuelve la raíz cuadrada de su argumento.

4.4 Prueba y depuración de funciones

"Y ahí estaba el infeliz—el miserable monstruo que yo había creado."

Mary Wollstonecraft Shelley, *Frankenstein*

Stubs y controladores

Cada función se debe diseñar, codificar y probar como unidad individual, aparte del resto del programa. Ésta es la esencia de la estrategia de diseño descendente. Si tratamos cada función como unidad aparte, transformamos una tarea grande en una serie de tareas más pequeñas y manejables. Pero, ¿cómo probamos una función fuera del programa para el cual se escribió? Escribimos un programa especial para efectuar la prueba. Por ejemplo, el cuadro 4.10 muestra un programa para probar la función `obtener_entradas` que se usó en el programa del cuadro 4.9. Los programas de este tipo se llaman **programas controladores** o *drivers*. Estos programas controladores son herramientas temporales, y pueden ser muy escuetos. No necesitan tener rutinas de entrada elegantes, ni realizar todos los cálculos que el programa final realizará. Lo único que tienen que hacer es obtener valores razonables para los argumentos de la función de la manera más sencilla posible (casi siempre del usuario), ejecutar la función y mostrar el resultado. Un ciclo como el del programa del cuadro 4.10 nos permitirá volver a probar la función con diferentes argumentos sin tener que volver a ejecutar el programa.

controladores

Si probamos cada función por separado, encontraremos casi todos los errores del programa. Además, sabremos cuáles funciones contienen esos errores. Si sólo probáramos el programa completo, probablemente averiguaríamos si hay un error o no, pero tal vez no tendríamos idea de dónde está el error. Peor aún, podríamos creer que sabemos dónde está el error, pero equivocarnos.

Una vez que hemos probado plenamente una función, podemos usarla en el programa controlador de alguna otra función. Cada función debe probarse en un programa en el que es la única función no probada. Sin embargo, sí podemos usar una función bien probada al probar alguna otra función. Si encontramos un error, sabemos que está en la función no probada. Por ejemplo, después de probar plenamente la función `obtener_entradas` con el programa controlador del cuadro 4.10, podemos usarla como rutina de entrada de programas controladores que prueban el resto de las funciones.

A veces es imposible o poco práctico probar una función sin usar alguna otra función que todavía no se ha escrito o que todavía no se ha probado. En un caso así, podemos usar una versión simplificada de la función faltante o no probada. Estas funciones simplificadas se llaman **stubs** (módulos instrumentales). Estos stubs no necesariamente realizan el cálculo correcto, pero proporcionan valores que bastan para las pruebas, y son lo bastante sencillas como para no tener dudas acerca de su desempeño. Por ejemplo, el programa del cuadro 4.11 se diseñó para probar la función `dar_salidas` del cuadro 4.9, así como la organización básica del programa. Este programa usa la función `obtener_entradas`, que ya probamos cabalmente con el programa controlador del cuadro 4.10. Este programa también incluye la función `inicializar_pantalla`, que suponemos ya se probó con su propio programa controlador, aunque no lo mostramos porque es demasiado sencillo. Puesto que todavía no probamos la función `precio`, hemos usado un stub en su lugar. Observe que podríamos usar este programa aun antes de escribir siquiera la función `precio`. De esta forma podemos probar la organización básica del programa antes de escribir los detalles de todas las definiciones de funciones.

stubs

`precio`

CUADRO 4.10 *Programa controlador (parte 1 de 2)*

```
//Programa controlador para la función obtener_entradas.
#include <iostream>

void obtener_entradas(double& costo, int& recambio);
//Precondición: El usuario está listo para introducir los
//valores correctamente
//Postcondición: Costo contiene el costo al mayoreo
//de un artículo, recambio contiene el número de días
//que se espera que el artículo tarde en venderse.

int main()
{
     using namespace std;
     double costo_mayoreo;
     int tiempo_en_anaquel;
     char respuesta;

     cout.setf(ios::fixed);
     cout.setf(ios::showpoint);
     cout.precision(2);
     do
     {
         obtener_entradas(costo_mayoreo, tiempo_en_anaquel);

         cout << "El costo al mayoreo ahora es $"
              << costo_mayoreo << endl;
         cout << "Los dias hasta venta son ahora "
              << tiempo_en_anaquel << endl;

         cout << "Probar otra vez?"
              << " (Escriba s es si o n es no): ";
         cin >> respuesta;
         cout << endl;
     } while (respuesta == 's' || respuesta == 'S');

     return 0;
}
```

Cuadro 4.10 *Programa controlador (parte 2 de 2)*

```
//Usa iostream:
void obtener_entradas(double& costo, int& recambio)
{
     using namespace std;
     cout << "Introduzca el costo al mayoreo del articulo: $";
     cin >> costo;
     cout << "Introduzca el numero de dias que se cree tardara en venderse: ";
     cin >> recambio;
}
```

Diálogo de ejemplo

```
Introduzca el costo al mayoreo del articulo: $123.45
Introduzca el numero de dias que se cree tardara en venderse: 67
El costo al mayoreo ahora es $123.45
Los dias hasta venta son ahora 67
¿Probar otra vez? (Escriba s si si o n si no): s
Introduzca el costo al mayoreo del articulo: $9.05
Introduzca el numero de dias que se cree tardara en venderse: 3
El costo al mayoreo ahora es $9.05
Los dias hasta venta son ahora 3
¿Probar otra vez? (Escriba s si si o n si no): n
```

El uso de un "bosquejo" de programa con stubs nos permite probar y luego "detallar" el bosquejo básico del programa, en lugar de escribir un programa totalmente nuevo para probar cada función. Por esta razón, un bosquejo de programa con stubs suele ser el método más eficiente de realizar pruebas. Una estrategia común es emplear programas controladores para probar algunas funciones básicas, como las de entrada y salida, y luego usar un programa con stubs para probar el resto de las funciones. Los stubs se sustituyen por funciones uno por uno: un stub se sustituye por una función completa y se prueba; una vez probada plenamente esa función, otro stub se sustituye por una definición de función completa, y así hasta tener el programa final.

Regla fundamental para probar funciones

Cada función debe probarse en un programa en el que todas las demás funciones ya se hayan probado y depurado cabalmente.

CUADRO 4.11 *Programa con un stub (parte 1 de 2)*

```
//Determina el precio al detalle de un artículo según las
//políticas de precios de la cadena de supermercados Compra Ágil.
#include <iostream>

void introduccion();
//Postcondición: Se describe el programa en la pantalla.

void obtener_entradas(double& costo, int& recambio);
//Precondición: El usuario está listo para introducir los valores correctamente
//Postcondición: Costo contiene el costo al mayoreo
//de un artículo, recambio contiene el número de días
//que se espera que el artículo tarde en venderse.

double precio(double costo, int recambio);
//Precondición: Costo es el costo al mayoreo de un artículo.
//recambio es el número de días que creemos tardará en venderse.
//Devuelve el precio al detalle del artículo.

void dar_salidas(double costo, int recambio, double precio);
//Precondición: Costo es el costo al mayoreo de un artículo, recambio es el número
//de días que creemos tardará en venderse; precio es el precio al detalle del artículo.
//Postcondición: Se han escrito en la pantalla los valores de costo,
//recambio y precio.

int main()
{
    double costo_mayoreo, precio_detalle;
    int tiempo_en_anaquel;

    introduccion();
    obtener_entradas(costo_mayoreo, tiempo_en_anaquel);
    precio_detalle = precio(costo_mayoreo, tiempo_en_anaquel);
    dar_salidas(costo_mayoreo, tiempo_en_anaquel, precio_detalle);
    return 0;
}
//Usa iostream:
void introduccion()
{
    using namespace std;
    cout << "Este programa determina el precio al detalle de\n"
         << "un articulo en un supermercado de la cadena Compra Agil.\n";
}
```

función cabalmente probada

CUADRO 4.11 *Programa con un stub (parte 2 de 2)*

función cabalmente probada

```
//Usa iostream:
void obtener_entradas(double& costo, int& recambio)
{
    using namespace std;
    cout << "Introduzca el costo al mayoreo del articulo: $";
    cin >> costo;
    cout << "Introduzca el numero de dias que se cree tardara en venderse: ";
    cin >> recambio;
}
```

función que se está probando

```
//Usa iostream:
void dar_salidas(double costo, int recambio, double precio)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Costo al mayoreo = $" << costo << endl
         << "Tiempo que se cree tardara en venderse = "
         << recambio << " dias" << endl
         << "Precio al detalle = $" << precio << endl;
}
```

stub

```
//Éste es sólo un stub:
double precio(double costo, int recambio)
{
    return 9.99; //No es correcto, pero basta para pruebas.
}
```

Diálogo de ejemplo

```
Este programa determina el precio al detalle de
un articulo en un supermercado de la cadena Compra Agil.
Introduzca el costo al mayoreo del articulo: $1.21
Introduzca el numero de dias que se cree tardara en venderse: 5
Costo al mayoreo = $1.21
Tiempo que se cree tardara en venderse = 5 dias
Precio al detalle = $9.99
```

Ejercicios de AUTOEVALUACIÓN

17. ¿Cuál es la regla fundamental para probar funciones? ¿Por qué es esa una buena forma de probar funciones?
18. ¿Qué es un programa controlador?
19. Escriba un programa controlador para la función `introduccion` que se muestra en el cuadro 4.11.
20. Escriba un programa controlador para la función `sumar_impuesto` del ejercicio número 11 de los ejercicios de autoevaluación.
21. ¿Qué es un stub?
22. Escriba un stub para la función cuya declaración de función se da a continuación. No escriba todo un programa, sólo el stub que se insertaría en un programa. Sugerencia: Es muy corto.

```
double prob_lluvia(double presion, double humedad, double temp);
//Precondición: presion es la presión barométrica en mm de mercurio,
//humedad es la humedad relativa como porcentaje, y temp
//es la temperatura en grados Fahrenheit.
//Devuelve la probabilidad de que llueva, que es un número entre 0 y 1.
//0 indica que no lloverá, 1 indica una certeza del 100% de que lloverá.
```

Resumen del capítulo

- Todas las subtareas de un programa se pueden implementar como funciones, sea como funciones que devuelven un valor o como funciones *void*.
- Un **parámetro formal** es una especie de "marcador de posición" que se llena con un **argumento** de función cuando se invoca la función. Hay dos métodos para efectuar esta sustitución: llamada por valor y llamada por referencia.
- En el mecanismo de sustitución de **llamada por valor**, el valor del argumento sustituye a su parámetro formal correspondiente. En el mecanismo de sustitución de **llamada por referencia**, el argumento debe ser una variable y toda la variable sustituye al parámetro corespondiente.
- La forma de indicar un parámetro de llamada por referencia en una definición de función es anexar el signo `&` al tipo del parámetro formal.
- Un argumento que corresponde a un parámetro de llamada por valor no puede ser modificado por una llamada a la función. Un argumento que corresponde a un parámetro de llamada por referencia sí puede ser modificado por una llamada a la función. Si queremos que una función modifique el valor de una variable, debemos usar un parámetro de llamada por referencia.
- Una buena forma de escribir un comentario de declaración de función es usar una precondición y una postcondición. La **precondición** indica lo que se supone que se cumple inmediatamente antes de invocarse la función. La **postcondición** describe el efecto de la llamada de función; es decir, la postcondición indica lo que será verdad después de que la función se ejecute en una situación en la que se cumpla la precondición.
- Toda función debe probarse en un programa en el que todas las demás funciones de ese programa ya se hayan probado y depurado completamente.

- Un **programa controlador** es un programa que lo único que hace es probar una función.
- Una versión simplificada de una función se llama **stub**. Se usa un stub en lugar de una definición de función que todavía no se ha probado (o tal vez ni siquiera se ha escrito) a fin de poder probar el resto del programa.

Respuestas a los ejercicios de autoevaluación

1.
```
Hola
Adios
Una vez mas:
Hola
Fin del programa.
```

2. No, una definición de función *void* no necesita contener una instrucción *return*. Una definición de función *void* puede contener una instrucción *return,* pero no es necesario.

3. Omitir la instrucción *return* en la definición de la función inicializar_pantalla del cuadro 4.2 no tendría ningún efecto sobre el comportamiento del programa. El programa se compilará, ejecutará y comportará exactamente de la misma manera. Así mismo, la omisión de la instrucción return en la definición de la función mostrar_resultados no afectará el comportamiento del programa. En cambio, si se omite la instrucción *return* en la definición de la función celsius habrá un error grave que impedirá la ejecución del programa. La diferencia es que las funciones inicializar_pantalla y mostrar_resultados son funciones *void*, pero celsius no es una función *void*.

4.
```
#include <iostream>
void mostrar_producto(int n1, int n2, int n3);
int main()
{
    using namespace std;
    int num1, num2, num3;
    cout << "Introduzca tres enteros: ";
    cin >> num1 >> num2 >> num3;
    mostrar_producto(num1, num2, num3);
    return 0;
}

void mostrar_producto(int n1, int n2, int n3)
{
    using namespace std;
    cout << "El producto de los tres numeros "
         << n1 << ", " << n2 << ", y "
         << n3 << " es " << (n1*n2*n3) << endl;
}
```

5. Estas respuestas dependen del sistema.

6. Una llamada a una función *void* seguida de un signo de punto y coma es una instrucción válida. Una llamada así no puede incrustarse en una expresión.

7.
```
10 20 30
1 2 3
1 20 3
```

8.
```
Introduzca dos enteros: 5 10
En orden inverso los numeros son: 5 5
```
← *diferente*

9.
```
par1_val en la llamada = 111
par2_ref en la llamada = 222
n1 despues de la llamada = 1
n2 despues de la llamada = 2
```
← *diferente*

10.
```
void cero_ambos(int& n1, int& n2)
{
    n1 = 0;
    n2 = 0;
}
```

11.
```
void sumar_impuesto(double tasa_impuesto, double& costo)
{
     costo = costo + ( tasa_impuesto/100.0 )*costo;
}
```

La división entre 100 es para convertir un porcentaje en una fracción. Por ejemplo, 10% es 10/100.0 o una décima parte del costo.

12. Sí, una función que regresa un valor puede tener un parámetro de llamada por referencia. Sí, una función puede tener una combinación de parámetros, de llamada por valor y de llamada por referencia.

13. No, una definición de función no puede aparecer dentro del cuerpo de otra definición de función.

14. Sí, una definición de función puede contener una llamada a otra función

15.
```
void ordenar(int& n1, int& n2);
//Precondición: Las variables n1 y n2 tienen valores.
//Postcondición: Los valores de n1 y n2 se han ordenado
//de modo que n1 <= n2.
```

16.
```
double sqrt(double n);
//Precondición: n >= 0.
//Devuelve la raíz cuadrada de n.
```

Podemos reescribir la segunda línea del comentario como sigue, pero la primera versión es la forma que suele usarse para una función que devuelve un valor.

```
//Postcondición: Devuelve la raíz cuadrada de n.
```

17. La regla fundamental para probar funciones es que cada función debe probarse en un programa en el que todas las demás funciones de ese programa ya se hayan probado y depurado cabalmente. Ésta es una buena forma de probar una función porque, si seguimos esta regla, cuando encontremos un error sabremos cuál función contiene el error.

18. Un programa controlador es un programa escrito con el único propósito de probar una función.

19.
```
#include <iostream>

void introduccion();
//Postcondición: Se describe el programa
//en la pantalla.
int main()
{
     using namespace std;
     introduccion();
     cout << "Fin de la prueba.\n";
     return 0;
}
//Usa iostream:
void introduccion()
{
     using namespace std;
     cout << "Este programa determina el precio al detalle de\n"
          << "un articulo en un supermercado de la cadena Compra Agil.\n";
}
```

20.
```
//Programa controlador para la función sumar_impuesto.
#include <iostream>

void sumar_impuesto(double tasa_impuesto, double& costo);
//Precondición: tasa_impuesto es el impuesto al valor agregado expresado como un
//porcentaje, y costo es el costo de un artículo antes de impuestos.
//Postcondición: costo ha cambiado al costo del
//artículo después de añadirle impuestos.

int main()
{
     using namespace std;
     double costo, tasa_impuesto;
     char respuesta;

     cout.setf(ios::fixed);
     cout.setf(ios::showpoint);
     cout.precision(2);
     do
     {
          cout << "Ingrese el costo y la tasa de impuesto:\n";
          cin >> costo >> tasa_impuesto;
          sumar_impuesto(tasa_impuesto, costo);

          cout << "Despues de la llamada a sumar_impuesto\n"
               << "tasa_impuesto es " << tasa_impuesto << endl
               << "costo es " << costo << endl;

        cout << "Probar otra vez?"
```

```
                << " (Escriba s si si o n si no): ";
            cin >> respuesta;
            cout << endl;
        } while (respuesta == 's' || respuesta == 'S');

        return 0;

}

void sumar_impuesto(double tasa_impuesto, double& costo)
{
    costo = costo + ( tasa_impuesto/100.0 )*costo;
}
```

21. Un stub es una versión simplificada de una función que se usa en lugar de la función para poder probar otras funciones.

22.
```
//ÉSTE ES UN STUB.
double prob_lluvia(double presion, double humedad, double temp)
{
    return 0.25; //No es correcto, pero basta para pruebas.
}
```

Proyectos de programación

1. Escriba un programa que efectúe conversiones de notación de 24 horas a notación de 12 horas. Por ejemplo, deberá convertir 14:25 a 2:25 PM. Las entradas son dos enteros. Debe haber por lo menos tres funciones, una para la entrada, otra para la conversión y otra para la salida. Registre la información AM/PM como un valor de tipo *char*, `'A'` para AM y `'P'` para PM. Así, la función para realizar las conversiones tendrá un parámetro formal de llamada por referencia de tipo *char* para registrar si es AM o PM. (La función tendrá también otros parámetros.) Incluya un ciclo que permita al usuario repetir este cálculo con otros valores de entrada una y otra vez hasta que el usuario indique que desea finalizar el programa.

2. Escriba un programa que solicite la hora actual y el tiempo de espera como dos enteros para el número de horas y minutos. El programa desplegará cuál será el tiempo de espera después de un lapso. Utilice notación de 24 horas. Incluya un ciclo que permita al usuario repetir este cálculo con otros valores de entrada una y otra vez hasta que el usuario indique que desea finalizar el programa.

3. Modifique su programa para el proyecto de programación 2 de manera que utilice la notación de 12 horas, por ejemplo 3:45 P.M.

4. Escriba una función que calcule la media y la desviación estándar de cuatro puntajes. La desviación estándar se define como la raíz cuadrada de la media de los cuatro valores $(s_i - a)^2$, donde a es la media de los cuatro puntajes s_1, s_2, s_3 y s_4. La función tendrá seis parámetros e invocará otras dos funciones. Incruste la función en un programa controlador que permita probar la función una y otra vez hasta que el usuario indique que desea terminar.

5. Escriba un programa que indique qué monedas hay que entregar para dar cualquier cantidad de cambio entre 1 y 99 centavos de dólar. Por ejemplo, si la cantidad es 86 centavos, la salida sería parecida a esto:

```
86 centavos puede entregarse como
3 cuarto(s) 1 diez(dieces) y 1 centavo(s)
```

Use denominaciones de monedas de 25 centavos (cuartos), 10 centavos (dieces) y 1 centavo. No use monedas de 5 centavos (níqueles) ni de medio dólar. Su programa usará la siguiente función (entre otras):

```
void calcular_moneda(int valor_moneda, int& numero, int& restante);
//Precondición: 0 < valor_moneda < 100; 0 <= restante < 100.
//Postcondición: Se ha asignado a numero el número máximo de monedas con
//denominación valor_moneda centavos que caben en restante centavos.
//Se ha restado a restante el valor de las monedas, es decir,
//se le ha restado numero*valor_moneda.
```

Por ejemplo, supongamos que el valor de la variable `restante` es `86`. Entonces, después de la siguiente llamada, el valor de `numero` será `3` y el valor de `restante` será `11` (porque si quitamos 3 cuartos a 86 centavos nos quedan 11 centavos):

```
calcular_monedas(25, numero, restante);
```

Incluya un ciclo que permita al usuario repetir este cálculo con otros valores de entrada hasta que el usuario diga que quiere terminar el programa. Sugerencia: Use división entera y el operador `%` para implementar esta función.

6. Escriba un programa que lea una longitud en pies y pulgadas y exhiba la longitud equivalente en metros y centímetros. Use al menos tres funciones: una para la entrada, una para calcular y una para la salida. Incluya un ciclo que permita al usuario repetir este cálculo con otros valores de entrada hasta que el usuario diga que quiere terminar el programa. Un pie equivale a 0.3048 metros, hay 100 centímetros en 1 metro y 12 pulgadas en un pie.

7. Escriba un programa como el del ejercicio anterior pero que convierta metros y centímetros a pies y pulgadas. Use funciones para las subtareas.

8. (Deberá hacer los dos proyectos de programación anteriores antes de intentar éste.) Escriba un programa que combine las funciones de los dos proyectos de programación anteriores. El programa pregunta al usuario si quiere convertir de pies y pulgadas a metros y centímetros o de metros y centímetros a pies y pulgadas, y luego efectúa la conversión deseada. Haga que el usuario responda tecleando el entero `1` si desea un tipo de conversión, o `2` si desea el otro. El programa lee la respuesta del usuario y luego ejecuta una instrucción `if-else`. Cada bifurcación de la instrucción `if-else` será una llamada de función. Las dos funciones que se invocan en la instrucción `if-else` tendrán definiciones muy similares a los programas de los dos proyectos de programación anteriores; por tanto, serán definiciones de función relativamente complicadas que invocan otras funciones dentro de su cuerpo. Incluya un ciclo que permita al usuario repetir este cálculo con otros valores de entrada hasta que el usuario diga que quiere terminar el programa.

9. Escriba un programa que lea un peso en libras y onzas y exhiba el peso equivalente en kilogramos y gramos. Use al menos tres funciones: una para la entrada, una para calcular y una para la salida. Incluya un ciclo que permita al usuario repetir este cálculo con otros valores de entrada hasta que el usuario diga que quiere terminar el programa. Un kilogramo equivale a 2.2046 libras, hay 1000 gramos en un kilogramo y 16 onzas en una libra.

10. Escriba un programa como el del ejercicio anterior pero que convierta kilogramos y gramos a libras y onzas. Use funciones para las subtareas.

11. (Deberá hacer los dos proyectos de programación anteriores antes de intentar éste.) Escriba un programa que combine las funciones de los dos proyectos de programación anteriores. El programa pregunta al usuario si quiere convertir de libras y onzas a kilogramos y gramos o de kilogramos y gramos a libras y onzas, y luego efectúa la conversión deseada. Haga que el usuario responda tecleando el entero `1` si desea un tipo de conversión, o `2` si desea el otro. El programa lee la respuesta del usuario y luego ejecuta una instrucción `if-else`. Cada bifurcación de la instrucción `if-else` será una llamada de función. Las dos funciones que se invocan en la instrucción `if-else` tendrán definiciones muy similares a los programas de los dos proyectos de programación anteriores; por tanto, serán definiciones de función relativamente complicadas que invocan otras funciones dentro de su cuerpo. Incluya un ciclo que permita al usuario repetir este cálculo con otros valores de entrada hasta que el usuario diga que quiere terminar el programa.

12. (Deberá hacer los proyectos de programación 8 y 11 antes de intentar éste.) Escriba un programa que combine las funciones de los proyectos de programación 8 y 11. El programa pregunta al usuario si quiere convertir longitudes o pesos. Si el usuario escoge longitudes, entonces el programa pregunta al usuario si quiere convertir de pies y pulgadas a metros y centímetros o de metros y centímetros a pies y pulgadas. Si el usuario escoge pesos, se hace una pregunta similar acerca de libras, onzas, kilogramos y gramos. Luego, el programa efectúa la conversión deseada. Haga que el usuario responda tecleando el entero `1` si desea un tipo de conversión, o `2` si desea el otro. El programa lee la respuesta del usuario y luego ejecuta una instrucción `if-else`. Cada bifurcación de la instrucción `if-else` será una llamada de función. Las dos funciones que se invocan en la instrucción `if-else` tendrán definiciones muy similares a los programas de los proyectos de programación 8 y 11; por tanto, serán definiciones de función relativamente complicadas que invocan otras funciones dentro de su cuerpo; no obstante, será muy fácil escribirlas adaptando los programas que escribió para los proyectos de programación 8 y 11. Tenga presente que su programa incluirá instrucciones `if-else` incrustadas dentro de las instrucciones `if-else`, pero sólo de forma indirecta. Las dos bifurcaciones de la instrucción `if-else` exterior serán llamadas de función. A su vez, estas dos llamadas incluirán una instrucción `if-else`, pero no es necesario pensar en ello; sólo son llamadas de función y los detalles están en una caja negra que se crea cuando se definen estas funciones. Si trata de crear una ramificación de cuatro vías, probablemente no va por el camino correcto. Sólo es necesario pensar en ramificaciones de dos vías (aunque el programa total sí se bifurca finalmente en cuatro casos.) Incluya un ciclo que permita al usuario repetir este cálculo con otros valores de entrada hasta que el usuario diga que quiere terminar el programa.

13. El área de un triángulo arbitrario se puede calcular usando la fórmula

$$\text{área} = \sqrt{s(s-a)(s-b)(s-c)}$$

donde a, b y c son las longitudes de los lados y s es el semiperímetro

$$s = (a + b + c)/2$$

Escriba una función `void` que use cinco parámetros: tres de llamada por valor que proporcionan las longitudes de los lados y dos de llamada por referencia para el área y el perímetro (*no el semiperímetro*) que la función calcula. Haga que su función sea robusta. Tenga presente que no todas las combinaciones de a, b y c producen un triángulo. Su función deberá producir resultados correctos con datos válidos y resultados razonables con combinaciones no válidas de datos de entrada.

14. Cuando hace frío, los meteorólogos informan un índice llamado *factor de enfriamiento por el viento* que toma en cuenta la velocidad del viento y la temperatura. Este índice es una medida del efecto de enfriamiento del viento a una temperatura dada del aire. El enfriamiento por viento se puede aproximar con la fórmula:

$$W = 13.12 + 0.6215*t - 11.37*v^{0.16} + 0.3965*t*v^{0.016}$$

donde

v = velocidad del viento en m/s
t = temperatura en grados Celsius: *t* <= 10
W = índice de enfriamiento del viento (en grados Celsius)

Escriba una función que devuelva el índice de enfriamiento por el viento. Su código deberá asegurar que no se viole la restricción de la temperatura. Busque algunos informes del tiempo en números atrasados de algún diario en la biblioteca de su universidad y compare el índice de enfriamiento que calcule con el resultado que se informa en el diario.

5 Flujos de E/S como introducción a los objetos y clases

5

Flujos de E/S como introducción a los objetos y clases

Los peces, por ejemplo, tienen su corriente y su estanque;
pero, ¿hay algo más allá?

RUPERT BROOKE, "*Heaven*" *(1913)*

Como una hoja que se lleva la corriente, que puede ir a dar a un lago o al mar, así las salidas de nuestro programa son llevadas por un flujo sin saber si el flujo va a la pantalla o a un archivo.

ESCRITO EN LA PARED DE UN SANITARIO DE UN DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN (*1995*)

Introducción

E/S se refiere a las entradas y salidas de un programa. Las entradas pueden tomarse del teclado o de un archivo. Así mismo, las salidas pueden enviarse a la pantalla o a un archivo. En este capítulo explicaremos la forma de escribir programas que tomen entradas de un archivo y envíen salidas a otro archivo.

Las entradas se envían a nuestros programas por medio de una construcción de C++ llamada *flujo* (*stream*), y las salidas de nuestros programas se envían al dispositivo de salida por medio de un flujo. Los flujos son nuestros primeros ejemplos de *objetos*. Un objeto es un tipo especial de variable que tiene sus propias funciones de propósito especial, las cuales en cierto sentido están ligadas a la variable. La capacidad para manejar objetos es una de las características que distingue a C++ de lenguajes de programación anteriores. En este capítulo veremos qué son los flujos y cómo podemos usarlos para efectuar la E/S de programas. Al mismo tiempo que explicamos los flujos, presentaremos las ideas básicas acerca de la naturaleza de los objetos y la forma de usarlos en los programas.

Prerrequisitos

Este capítulo utilizará material de los capítulos 2 al 4.

5.1 Flujos y E/S de archivos básica

¡Santo cielo! Durante más de cuarenta años he estado hablando en prosa sin saberlo.

Molière, *El burgués gentilhombre*

archivos

Ya estamos usando archivos para guardar nuestros programas. También podemos usar archivos para guardar las entradas de un programa o para recibir las salidas de un programa. Los archivos que se usan para E/S de programas son similares a los que se usan para guardar programas. Los flujos, que veremos a continuación, nos permiten escribir programas que manejan entradas de archivos y entradas del teclado de manera unificada, y que manejan salidas a archivos y salidas a la pantalla de forma unificada también.

flujo
flujo de entrada
flujo de salida

Un **flujo** es una "corriente" de caracteres (u otro tipo de datos). Si el flujo entra en el programa, se trata de un **flujo de entrada**. Si el flujo sale del programa, se trata de un **flujo de salida**. Si el flujo de entrada proviene del teclado, el programa recibe entradas del teclado. Si el flujo de entrada proviene de un archivo, el programa recibirá sus entradas de ese archivo. De forma similar, un flujo de salida puede ir a la pantalla o a un archivo.

`cin` y `cout` son flujos

Aunque no se haya dado cuenta, ya ha estado empleando flujos en sus programas. El `cin` que ya usamos es un flujo de entrada conectado al teclado, y `cout` es un flujo de salida conectado a la pantalla. El programa dispone automáticamente de estos dos flujos si tiene una directiva `include` que nombre al archivo de encabezado `iostream`. Podemos definir otros flujos que provengan de o vayan a archivos; una vez definidos, podemos usarlos en nuestros programas del mismo modo que usamos los flujos `cin` y `cout`.

Por ejemplo, supongamos que nuestro programa define un flujo llamado `flujo_in` que proviene de algún archivo. (Explicaremos cómo definirlo un poco más adelante.) Entonces podremos llenar una variable *`int`* llamada `el_numero` con un número de este archivo usando lo que sigue en nuestro programa:

```
int el_numero;
flujo_in >> el_numero;
```

De forma similar, si nuestro programa define un flujo de salida llamado `flujo_out` que va a otro archivo, podemos enviar el valor de nuestra variable a ese archivo. Lo que sigue envía la cadena `"el_numero es "` seguida del contenido de la variable `el_numero` al archivo de salida conectado al flujo `flujo_out`.

```
flujo_out << "el_numero es " << el_numero << endl;
```

Una vez que los flujos se conectan a los archivos deseados, el programa puede efectuar E/S con archivos de la misma manera que efectúa E/S usando el teclado y la pantalla.

Por qué usar archivos para E/S

almacenamiento permanente

La entrada del teclado y la salida a la pantalla que hemos usado hasta ahora manejan datos temporales. Cuando el programa termina, los datos que tecleamos y los que se desplegaron en la pantalla desaparecen. Los archivos nos permiten almacenar datos de forma permanente. El contenido de un archivo perdura hasta que una persona o un programa modifica ese archivo. Si nuestro programa envía sus salidas a un archivo, el archivo de salida permanecerá después de que el programa haya terminado de ejecutarse. Los archivos de entrada

pueden ser utilizados una y otra vez por muchos programas sin necesidad de teclear los datos por separado para cada programa.

Los archivos de entrada y de salida que nuestro programa usa son el mismo tipo de archivos que leemos y escribimos con un editor, como el que usamos para escribir nuestros programas. Esto implica que podemos crear un archivo de entrada para nuestro programa o leer un archivo de salida producido por nuestro programa cada vez que nos convenga hacerlo, en lugar de tener que leer y escribir todo mientras el programa se está ejecutando.

Los archivos también son una forma adecuada de manejar grandes cantidades de datos. Si nuestro programa obtiene sus entradas de un archivo de entrada grande, recibe muchos datos sin que el usuario tenga que teclear mucho.

E/S de archivos

lectura y escritura

Cuando un programa obtiene entradas de un archivo decimos que está **leyendo** del archivo, y cuando envía salidas a un archivo decimos que está **escribiendo** en el archivo. Hay otras formas de leer entradas de un archivo, pero el método que usaremos lee el archivo de principio a fin (o hasta donde el programa logre llegar antes de terminar su ejecución). Con este método, el programa no puede retroceder y leer una parte del archivo una segunda vez. Esto es exactamente lo que sucede cuando un programa toma entradas del teclado, por lo que no deberá parecernos nuevo ni extraño. (Como veremos, el programa puede volver a leer un archivo si comienza otra vez desde el principio, pero esto es "volver a comenzar", no "retroceder".) Así mismo, con el método que presentamos aquí, el programa escribe salidas en un archivo comenzando en el principio del archivo y procediendo hacia adelante. El programa no puede retroceder y modificar alguna salida que ya escribió en el archivo. Esto es exactamente lo que sucede cuando el programa envía salidas a la pantalla; podemos enviar más salidas a la pantalla, pero no podemos retroceder y cambiar lo que se envió antes. La forma en que obtenemos entradas de un archivo para introducirlas al programa, o enviamos salidas del programa a un archivo, es conectando el programa al archivo por medio de un flujo.

un flujo es una variable

En C++ un flujo es un tipo especial de variable conocido como *objeto*. Trataremos los objetos en la próxima sección, pero primero describiremos la forma en que nuestro programa puede usar objetos de flujo para realizar E/S sencilla con archivos. Si queremos usar un flujo para obtener entradas de un archivo (o enviar salidas a un archivo) debemos declarar el flujo, y debemos conectar el flujo al archivo.

Podemos pensar en el archivo al que está conectado un flujo como el valor del flujo. Podemos desconectar un flujo de un archivo y conectarlo a otro, o sea que podemos cambiar el valor de estas variables de flujo. Sin embargo, para realizar tales cambios es preciso usar funciones especiales que sólo aplican a flujos. *No podemos* usar una variable de flujo en una instrucción de asignación igual que usamos una variable de tipo `int` o `char`. Aunque los flujos son variables, estos son tipos de variables no usuales.

declaración de flujos

Los flujos `cin` y `cout` ya están declarados, pero si queremos conectar un flujo a un archivo tendremos que declararlo igual que declararíamos cualquier otra variable. El tipo para las variables de flujo de archivo de entrada se llama `ifstream` (de "*input file stream*", flujo de archivo de entrada). El tipo para las variables de flujo de archivo de salida se llama `ofstream` (de "*output file stream*", flujo de archivo de salida). Así, podemos declarar `flujo_in` como flujo de entrada para un archivo, y `flujo_out` como flujo de salida para otro archivo como sigue:

`ifstream` y `ofstream`

```
ifstream flujo_in;
ofstream flujo_out;
```

fstream

Los tipos `ifstream` y `ofstream` están definidos en la biblioteca que lleva el archivo de encabezado `fstream`, por lo que cualquier programa que declare variables de flujo de esta manera deberá contener la siguiente directiva (normalmente cerca del principio del archivo):

```
#include <fstream>
```

Generalmente, al utilizar los tipos `ifstream` y `ofstream` nuestro programa también deberá contener, al principio del archivo o al principio del cuerpo de la declaración (que utilice los tipos `ifstream` o `ofstream`), lo siguiente:

```
using namespace std;
```

conexión de un flujo a un archivo `open`

Las variables de flujo, como `flujo_in` y `flujo_out` que declaramos antes, deben estar **conectadas** cada una a un archivo. Esto se denomina **abrir el archivo** y se efectúa con una función llamada `open`. Por ejemplo, supongamos que queremos que el flujo de entrada `flujo_in` esté conectado al archivo de nombre `archin.dat`. Entonces, nuestro programa deberá contener lo que sigue para poder leer entradas de ese archivo:

```
flujo_in.open("archin.dat");
```

Ésta podría parecer una sintaxis extraña para una llamada de función. Tendremos más que decir acerca de esta peculiar sintaxis en la sección que sigue. Por ahora, tomemos únicamente nota de un par de detalles acerca de la forma de escribir esta llamada a `open`. Primero se coloca el nombre de la variable de flujo y un punto antes de la función llamada `open`, y el nombre del archivo se da como argumento de `open`. Observe también que el nombre del archivo se da entre comillas. El nombre de archivo que se proporciona como argumento es el mismo nombre que usaríamos si quisiéramos escribir en el archivo usando un editor. Si el archivo de entrada está en el mismo directorio que el programa, probablemente bastará con dar el nombre del archivo en la forma que hemos descrito. En algunas situaciones podría ser necesario especificar también el directorio que contiene el archivo. Los detalles de la especificación de directorios varían de un sistema a otro. Si necesita especificar un directorio, pida a su profesor o a algún experto local que le explique los detalles.

nombre de archivo

Una vez que hemos declarado una variable de flujo de entrada y la hemos conectado a un archivo con la función `open`, nuestro programa puede tomar entradas del archivo empleando el operador de extracción, `>>`. Por ejemplo, lo que sigue lee dos números de entrada del archivo conectado a `flujo_in`, y los coloca en las variables `un_numero` y `otro_numero`:

```
int un_numero, otro_numero;
flujo_in >> un_numero >> otro_numero;
```

Los flujos de salida se abren (es decir, se conectan a archivos) de la misma manera que los flujos de entrada. Por ejemplo, lo que sigue declara el flujo de salida `flujo_out` y lo conecta al nombre de archivo `archout.dat`:

```
ofstream flujo_out;
flujo_out.open("archout.dat");
```

Cuando se usa con un flujo del tipo `ofstream`, la función miembro `open` crea el archivo de salida si es que no existe ya. Si el archivo de salida ya existe, la función miembro desecha el contenido del archivo, de modo que el archivo de salida está vacío después de la llamada a `open`.

Una vez que conectamos un archivo al flujo `flujo_out` con la llamada a `open`, el programa puede enviar salidas a ese archivo empleando el operador de inserción `<<`. Por ejemplo, lo que sigue escribe dos cadenas y el contenido de las variables `un_numero` y `otro_numero` en el archivo conectado al flujo `flujo_out` (que en este ejemplo es `archout.dat`):

```
flujo_out << "un_numero = " << un_numero
          << " otro_numero = " << otro_numero;
```

Cabe señalar que cuando un programa está manejando un archivo es como si éste tuviera dos nombres. Uno es el nombre normal del archivo, el que el sistema operativo usa. Este nombre se denomina **nombre de archivo externo**. En nuestro ejemplo de código los nombres de archivo externos fueron `archin.dat` y `archout.dat`. El nombre de archivo externo es en cierto modo el "nombre real" del archivo. Las convenciones para escribir estos nombres de archivo externos varían de un sistema a otro; su profesor o algún otro experto local tendrá que enseñarle esas convenciones. Los nombres `archin.dat` y `archout.dat` que usamos en nuestros ejemplos podrían parecerse o no a los nombres de archivo que se usan en el sistema del lector. Dé nombre a sus archivos siguiendo las convenciones empleadas en su sistema. Aunque el nombre de archivo externo es el verdadero nombre del archivo, por lo regular sólo se le usa una vez dentro de un programa. El nombre de archivo externo se da como argumento de la función `open`, pero *una vez que el archivo está abierto, siempre se hace referencia a él nombrando el flujo que está conectado a ese archivo*. Así pues, dentro de nuestro programa, el nombre del flujo hace las veces de segundo nombre del archivo.

nombre de archivo externo

El programa del cuadro 5.1 lee tres números de un archivo y escribe su suma, junto con algo de texto, en otro archivo.

Los archivos tienen dos nombres

Todos los archivos de entrada y de salida que nuestros programas usan tienen dos nombres. El nombre de **archivo externo** es el verdadero nombre del archivo, pero sólo se usa en la llamada a la función `open`, que conecta el archivo a un flujo. Después de la llamada a `open`, siempre usamos el nombre del flujo como nombre del archivo.

Todos los archivos deben cerrarse cuando el programa ha terminado de obtener entradas del archivo o de enviar salidas al archivo. Al cerrar un archivo desconectamos el flujo del archivo. Los archivos se cierran con una llamada a la función `close`. Las siguientes líneas del programa del cuadro 5.1 ilustran la forma de usar la función `close`:

`close`

```
flujo_in.close();
flujo_out.close();
```

Observe que la función `close` no recibe argumentos. Si nuestro programa termina normalmente pero sin cerrar un archivo, el sistema cerrará automáticamente el archivo. No obstante, es bueno acostumbrarse a cerrar los archivos por al menos dos razones. Primera, el sistema sólo cierra los archivos si el programa termina normalmente. Si el programa termina anormalmente a causa de un error, el archivo no se cerrará y podría contener datos no válidos. Si el programa cierra los archivos tan pronto como termina de usarlos, es menos probable que lleguen a contener datos no válidos. Una segunda razón para cerrar los archivos es que podríamos querer que nuestro programa envíe salidas a un archivo y más adelante lea otra vez esas salidas como entradas del programa. Para ello, el programa debe cerrar el

CUADRO 5.1 *Entrada/salida de archivos sencilla*

```
//Lee tres números del archivo archin.dat, los suma, y
//escribe la suma en el archivo archout.dat.
//(Se dará una versión mejor de este programa en el cuadro 5.2.)
#include <fstream>

int main()
{
    using namespace std;
    ifstream flujo_in;
    ofstream flujo_out;

    flujo_in.open("archin.dat");
    flujo_out.open("archout.dat");

    int primero, segundo, tercero;
    flujo_in >> primero >> segundo >> tercero;
    flujo_out << "La suma de los primeros\n"
              << "tres numeros de\n"
              << "archin.dat es " << (primero + segundo + tercero)
              <<endl;

    flujo_in.close();
    flujo_out.close();

    return 0;
}
```

archin.dat
(El programa no lo modifica.)

```
1
2
3
4
```

archout.dat
(Después de ejecutarse el programa.)

```
La suma de los primeros
tres números de
archin.dat es 6
```

No hay salidas a la pantalla ni entradas del teclado.

archivo después de terminar de escribir en él, y luego conectar el archivo a un flujo de entrada empleando la función `open`. (Es posible abrir un archivo para entrada y salida al mismo tiempo, pero esto se hace de una forma un tanto distinta y no analizaremos esa alternativa.)

Introducción a clases y objetos

Los flujos `flujo_in` y `flujo_out` que usamos en la sección anterior y los flujos predefinidos `cin` y `cout` son objetos. Un **objeto** es una variable que además de tener asociado un valor, tiene asociadas funciones. Por ejemplo, los flujos `flujo_in` y `flujo_out` tienen asociada una función llamada `open`. A continuación damos dos ejemplos de llamadas de esas funciones, junto con las declaraciones de los objetos `flujo_in` y `flujo_out`:

objeto

```
ifstream flujo_in;
ofstream flujo_out;
flujo_in.open("archin.dat");
flujo_out.open("archout.dat");
```

Hay una razón para esta notación tan peculiar. La función llamada `open` asociada al objeto `flujo_in` no es la misma función llamada `open` que está asociada al objeto `flujo_out`. Una función abre un archivo para entrada y la otra abre un archivo para salida. Desde luego, ambas funciones son similares; ambas "abren archivos". Cuando damos el mismo nombre a dos funciones es porque esas funciones tienen alguna similitud intuitiva. No obstante, estas dos funciones llamadas `open` son distintas, aunque las diferencias sean pequeñas. Cuando el compilador ve una llamada a una función llamada `open`, debe decidir a cuál de las dos funciones llamadas `open` se refiere. El compilador determina esto examinando el nombre del objeto que precede al punto, en este caso `flujo_in` o bien `flujo_out`. Una función que está asociada a un objeto se llama **función miembro**. Así, por ejemplo, `open` es una función miembro del objeto `flujo_in`, y otra función llamada `open` es miembro del objeto `flujo_out`.

función miembro

Como acabamos de ver, diferentes objetos pueden tener diferentes funciones miembro. Estas funciones pueden tener los mismos nombres, como en el caso de `open`, o pueden tener nombres totalmente distintos. El tipo de un objeto determina qué funciones miembro tiene ese objeto. Si dos objetos son del mismo tipo, podrían tener diferentes valores, pero tienen las mismas funciones miembro. Por ejemplo, supongamos que declaramos los siguientes objetos flujo:

```
ifstream flujo_in, flujo_in2;
ofstream flujo_out, flujo_out2;
```

Las funciones `flujo_in.open` y `flujo_in2.open` son la misma función. De forma similar, `flujo_out.open` y `flujo_out2.open` son la misma función (pero son diferentes de las funciones `flujo_in.open` y `flujo_in2.open`).

Un tipo cuyas variables son objetos —como `ifstream` y `ofstream`— se llama **clase**. Puesto que las funciones miembro de un objeto están totalmente determinadas por su clase (es decir, por su tipo), dichas funciones se llaman *funciones miembro de la clase* (además de ser *miembros del objeto*). Por ejemplo, la clase `ifstream` tiene una función miembro llamada `open` y la clase `ofstream` tiene una función miembro distinta llamada `open`. La clase `ofstream` también tiene una función miembro llamada `precision`, pero la clase `ifstream` no tiene ninguna función miembro llamada `precision`. Ya hemos estado usando la función miembro `precision` con el flujo `cout`, pero la estudiaremos con mayor detalle más adelante.

clase

llamada a una función miembro

Cuando invocamos una función miembro en un programa, siempre especificamos un objeto, por lo regular escribiendo el nombre del objeto y un punto antes del nombre de la función, como en el siguiente ejemplo:

```
flujo_in.open("archin.dat");
```

Una razón para nombrar el objeto es que la función podría tener algún efecto sobre el objeto. En el ejemplo anterior, la llamada a la función `open` conecta el archivo `archin.dat` al flujo `flujo_in`, por lo que necesita saber el nombre de dicho flujo.

En una llamada de función, como en

```
flujo_in.open("archin.dat");
```

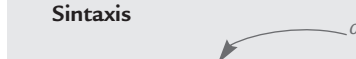
operador punto
objeto invocador

el punto se llama **operador punto** y el objeto cuyo nombre precede al punto es el **objeto invocador**. En ciertos sentidos, el objeto invocador es como un argumento adicional de la función —la función puede modificar el objeto invocador como si fuera un argumento— pero dicho objeto desempeña un papel todavía más importante en la llamada. El objeto invocador determina el significado del nombre de la función. El compilador usa el tipo del objeto invocador para determinar el significado del nombre de la función. Por ejemplo, en la llamada anterior a `open`, el tipo del objeto `flujo_in` determina el significado del nombre de función `open`.

El nombre de función `close` es análogo a `open`. Tanto `ifstream` como `ofstream` tienen una función miembro llamada `close`. Ambas "cierran archivos", pero lo hacen de diferente manera, porque los archivos se abrieron de diferente manera y se manipularon de diferente manera. Hablaremos de más funciones miembro para las clases `ifstream` y `ofstream` en una sección posterior de este capítulo.

Cómo invocar una función miembro

Sintaxis



Objeto_Invocador.*Nombre_de_Funcion_Miembro*(*Lista_de_Argumentos*);

Ejemplos

```
flujo_in.open("archin.dat");
flujo_out.open("archout.dat");
flujo_out.precision(2);
```

El significado del *Nombre_de_Funcion_Miembro* está determinado por la clase (es decir, el tipo) del *Objeto_Invocador*.

Clases y objetos

Un **objeto** es una variable que tiene asociadas funciones. Estas funciones se llaman **funciones miembro**. Una **clase** es un tipo cuyas variables son objetos. La clase del objeto (es decir, su tipo) determina cuáles funciones miembro tiene dicho objeto.

TIP DE PROGRAMACIÓN

Verifique si se logró abrir un archivo satisfactoriamente

Una llamada a `open` puede fallar por varias razones. Por ejemplo, si abrimos un archivo de entrada y no hay ningún archivo con el nombre externo que especificamos, la llamada a `open` fallará. Cuando esto sucede, es posible que no se despliegue un mensaje de error y que el programa continúe y haga algo inesperado. Por ello, después de una llamada a `open` siempre debemos efectuar una prueba para determinar si la llamada a `open` tuvo éxito, y terminar el programa (o efectuar alguna otra acción apropiada) si la llamada falló.

la función miembro `fail`

Podemos emplear la función miembro `fail` para probar si una operación de flujo falló o no. Existe una función miembro llamada `fail` tanto para la clase `ifstream` como para la clase `ofstream`. La función `fail` no recibe argumentos y devuelve un valor `bool`. Una llamada a la función `fail` para un flujo llamado `flujo_in` tendría este aspecto:

```
flujo_in.fail()
```

Ésta es una expresión booleana que puede servir para controlar un ciclo *`while`* o una instrucción *`if-else`*.

Recomendamos colocar una llamada a `fail` inmediatamente después de cada llamada a `open`; si la llamada a `open` falla, la función `fail` devolverá *true* (es decir, compilará la expresión booleana). Por ejemplo, si la siguiente llamada a `open` falla, el programa desplegará un mensaje de error y terminará; si la llamada tiene éxito, la función `fail` devolverá *false*, y el programa continuará.

```
flujo_in.open("cosas.dat");
if (flujo_in.fail())
{
   cout << "No pudo abrirse el archivo de entrada.\n";
   exit(1);<------ Termina el programa
}
```

`fail` es una función miembro, por lo que se invoca usando el nombre del flujo y un punto. Desde luego, la llamada a `flujo_in.fail` sólo se refiere a una llamada a `open` de la forma `flujo_in.open`, y no a cualquier llamada a la función `open` efectuada con cualquier otro flujo como objeto invocador.

`exit`

La instrucción `exit` que aparece en el código anterior nada tiene que ver con las clases y tampoco tiene que ver nada directamente con flujos, pero se usa con frecuencia en este contexto. La instrucción `exit` hace que nuestro programa termine de inmediato. La función `exit` devuelve su argumento al sistema operativo. Para usar la instrucción `exit`, el programa debe contener la siguiente directiva `include`:

```
#include <cstdlib>
```

Generalmente, al utilizar `exit`, nuestro programa también deberá contener, al principio del archivo o al principio del cuerpo de la declaración (que utilice `exit`), lo siguiente:

```
using namespace std;
```

`exit` es una función predefinida que recibe un solo argumento entero; pero por convención, se usa `1` como argumento si la llamada a `exit` se debió a un error; de lo contrario se usa `0`.[1] Para nuestros fines, da lo mismo que se use cualquier entero, pero conviene seguir esta convención porque es importante en programación más avanzada.

La instrucción `exit`

La instrucción `exit` se escribe así:

```
exit(Valor_Entero);
```

Cuando se ejecuta la instrucción `exit`, el programa termina de inmediato. Se puede usar cualquier *Valor_Entero*, pero por convención se usa `1` para una llamada a `exit` causada por un error, y `0` en todos los demás casos. La instrucción `exit` es una llamada a la función `exit`, que está en la biblioteca cuyo archivo de encabezado es `cstdlib`. Por tanto, cualquier programa que use la instrucción `exit` deberá contener la siguiente directiva `include`:

```
#include <cstdlib>
using namespace std;
```

(Estas directivas no tienen que estar una después de la otra. Se colocan en la misma ubicación, de manera similar a las directivas que hemos visto.)

El cuadro 5.2 contiene el programa del cuadro 5.1 reescrito para incluir pruebas que revelan si los archivos de entrada y salida se abrieron con éxito. Los archivos se procesan exactamente de la misma manera que en el programa del cuadro 5.1. En particular, suponiendo que el archivo `archin.dat` existe y tiene el contenido que se muestra en el cuadro 5.1, el programa del cuadro 5.2 creará el archivo `archout.dat` que se muestra en el cuadro 5.1. Pero si hay un problema y alguna de las llamadas a `open` falla, el programa del cuadro 5.2 terminará y enviará a la pantalla un mensaje de error apropiado. Por ejemplo, si no existe ningún archivo llamado `archin.dat`, la llamada a `flujo_in.open` fallará, el programa terminará, y se escribirá un mensaje de error en la pantalla.

Observe que usamos `cout` para desplegar el mensaje de error. La razón es que queremos que el mensaje de error vaya a la pantalla, no a un archivo. Puesto que este programa usa `cout` para enviar salidas a la pantalla (además de realizar E/S con archivos), hemos añadido una directiva `include` para el archivo de encabezado `iostream`. (En realidad, un programa no necesita tener `#include <iostream>` si tiene `#include <fstream>`, pero no hay problema si la incluimos, y nos recuerda que el programa está usando salidas a la pantalla además de E/S con archivos.)

Técnicas de E/S de archivos

Como ya señalamos, los operadores `>>` y `<<` funcionan de la misma manera con flujos conectados a archivos que con `cin` y `cout`. Sin embargo, el estilo de programación para la E/S con archivos es diferente de aquél para E/S usando la pantalla y el teclado. Al leer

[1] UNIX, MSDOS y Windows utilizan 1 para indicar error y 0 para indicar éxito, mientras que VMS de Digital invierte esta convención. Pregunte a su profesor qué valores debe utilizar.

CUADRO 5.2 *E/S de archivos con verificación de* `open`

```
//Lee tres números del archivo archin.dat, los suma, y
//escribe la suma en el archivo archout.dat.
#include <fstream>
#include <iostream>
#include <cstdlib>

int main()
{
    using namespace std;
    ifstream flujo_in;
    ofstream flujo_out;

    flujo_in.open("archin.dat");
    if (flujo_in.fail())
    {
        cout << "No pudo abrirse el archivo de entrada.\n";
        exit(1);
    }
    flujo_out.open("archout.dat");
    if (flujo_out.fail())
    {
        cout << "No pudo abrirse el archivo de salida.\n";
        exit(1);
    }

    int primero, segundo, tercero;
    flujo_in >> primero >> segundo >> tercero;
    flujo_out << "La suma de los primeros \n"
              << "tres numeros de\n"
              << "archin.dat es " << (primero + segundo + tercero)
              << endl;

    flujo_in.close();
    flujo_out.close();

    return 0;
}
```

Salida a la pantalla (Si no existe el archivo `archin.dat`)

```
No pudo abrirse el archivo de entrada.
```

entradas del teclado es recomendable solicitar las entradas y hacer eco de ellas, de esta manera:

```
cout << "Escriba el numero: ";
cin >> el_numero;
cout << "El numero que escribio es " << el_numero;
```

Cuando nuestro programa recibe sus entradas de un archivo, no hay que incluir tales líneas de solicitud ni hacer eco de las entradas, porque no hay nadie que lea la solicitud y el eco y responda a ellos. Al leer entradas de un archivo, debemos estar seguros de que los datos del archivo son exactamente el tipo de datos que el programa espera. Entonces, el programa simplemente leerá el archivo de entrada suponiendo que los datos que necesita estarán ahí cuando se soliciten. Si `archivo_in` es una variable de flujo conectada a un flujo de entrada y queremos sustituir la E/S de teclado y pantalla anterior por entradas del archivo conectado a `archivo_in`, las tres líneas anteriores serían sustituidas por la siguiente línea:

```
archivo_in << el_numero;
```

Podemos tener cualquier cantidad de flujos abiertos para entrada o para salida. Así, un solo programa puede recibir entradas del teclado y también de uno o más archivos. El mismo programa podría enviar salidas a la pantalla y a uno o más archivos. Como alternativa, un programa podría recibir todas sus entradas del teclado y enviar sus salidas tanto a la pantalla como a un archivo. Se permite cualquier combinación de flujos de entrada y salida. En casi todos los ejemplos de este libro usaremos `cin` y `cout` para efectuar E/S con el teclado y la pantalla, pero es fácil modificar esos programas de modo que reciban sus entradas de un archivo y/o envíen sus salidas a un archivo.

Ejercicios de AUTOEVALUACIÓN

1. Suponga que está escribiendo un programa que usa un flujo llamado `fin` que se conectará a un archivo de entrada y un flujo llamado `fout` que se conectará a un archivo de salida. ¿Cómo declara `fin` y `fout`? ¿Qué directiva `include`, si acaso, necesita colocar en su archivo de programa?

2. Suponga que está continuando con la escritura del programa del que se habló en el ejercicio anterior y que quiere que el programa reciba sus entradas del archivo `cosas1.dat` y envíe sus salidas al archivo `cosas2.dat`. ¿Qué instrucciones necesita incluir en su programa para conectar el flujo `fin` al archivo `cosas1.dat` y para conectar el flujo `fout` al archivo `cosas2.dat`? No olvide incluir verificaciones para determinar si las aperturas tuvieron éxito.

3. Suponga que todavía está escribiendo el programa de los dos ejercicios anteriores y que llega a un punto en el que ya no necesita recibir entradas del archivo `cosas1.dat` y ya no necesita enviar salidas al archivo `cosas2.dat`. ¿Cómo cierra esos archivos?

4. Suponga que quiere modificar el programa del cuadro 5.1 de modo que envíe sus salidas a la pantalla en lugar de al archivo `archout.dat`. (Las entradas todavía se tomarán del archivo `archin.dat`.) ¿Qué cambios necesita hacer al programa?

5. ¿Qué directiva `include` necesita incluir en su archivo de programa si su programa usa la función `exit`?

Resumen de instrucciones de E/S con archivos

En esta muestra las entradas provienen de un archivo que tiene el nombre de directorio `archin.dat` y las salidas se envían a un archivo que tiene el nombre de directorio `archout.dat`.

- Coloque las siguientes directivas `include` en su archivo de programa:

```
#include <fstream>    <-- para E/S con archivos
#include <iostream>   <-- para cout
#include <cstdlib>    <-- para exit
```

- Escoja un nombre para el flujo de entrada, digamos `flujo_in`, y declárelo como variable de tipo `ifstream`. Escoja un nombre para el flujo de salida, digamos `flujo_out`, y declárelo como de tipo `ofstream`:

```
using namespace std;
ifstream flujo_in;
ofstream flujo_out;
```

- Conecte cada flujo a un archivo usando la función miembro `open` con el nombre de archivo externo como argumento. Recuerde usar la función miembro `fail` para determinar si la llamada a `open` tuvo éxito:

```
flujo_in.open("archin.dat");
if (flujo_in.fail())
{
   cout << "No pudo abrirse el archivo de entrada.\n";
   exit(1);
}
flujo_out.open("archout.dat");
if (flujo_out.fail())
{
   cout << "No pudo abrirse el archivo de salida.\n";
   exit(1);
}
```

- Use el flujo `flujo_in` para obtener entradas del archivo `archin.dat` del mismo modo que usa `cin` para obtener entradas del teclado. Por ejemplo:

```
flujo_in >> alguna_variable >> otra_variable;
```

- Use el flujo `flujo_out` para enviar salidas al archivo `archout.dat` del mismo modo que usa `cout` para enviar salidas a la pantalla. Por ejemplo:

```
flujo_out << "alguna_variable = "
          << alguna_variable << endl;
```

- Cierre los flujos usando la función `close`:

```
flujo_in.close();
flujo_out.close();
```

6. Continuando con la pregunta 5, ¿qué hace `exit(1)` con su argumento?
7. Suponga que `bla` es un objeto, `dobedo` es una función miembro del objeto `bla`, y `dobedo` recibe un argumento de tipo *`int`*. ¿Cómo escribe una llamada a la función miembro `dobedo` del objeto `bla` usando el argumento `7`?
8. ¿Qué características de los archivos comparten las variables de programa ordinarias? ¿Qué características de los archivos son diferentes de las de las variables ordinarias de un programa?

9. Mencione al menos tres funciones miembro asociadas a un objeto `ifstream`, y dé ejemplos de uso de cada una.
10. Un programa ha leído la mitad de las líneas de un archivo. ¿Qué debe hacer el programa con el archivo para poder leer la primera línea otra vez?
11. En el texto dice "un archivo que tiene 2 nombres" ¿Cuáles son? ¿Cuándo se utiliza cada uno?

Añadir a un archivo (*Opcional*)

Al mandar una salida a un archivo, nuestro código debe utilizar primero la función miembro `open` para abrir el archivo y conectarlo a un flujo de tipo `ofstream`. La forma en que hasta ahora hemos hecho esto (con un solo argumento para el nombre del archivo) abrirá siempre un archivo vacío. Si el archivo con el nombre especificado ya existe, el contenido anterior se perderá. Existe otra forma para abrir un archivo para que la salida de su programa se añada a su archivo después de cualquier dato que ya esté en el archivo.

Para añadir su salida a un archivo llamado `importante.txt`, deberá utilizar una versión de doble argumento de `open` como se muestra a continuación:

```
ofstream outStream;
outStream.open ("importante.txt", ios::app);
```

Si no existe el archivo `importante.txt` éste creará un archivo vacío con ese nombre para recibir la salida de su programa, pero si el archivo ya existe, entonces toda la salida del programa se añadirá al final del archivo, para que los datos anteriores del archivo no se pierdan, esto se ilustra en el cuadro 5.3.

El segundo argumento `ios::app` es una constante especial que se define en `iostream` y por lo tanto requiere la siguiente directiva `include`:

```
#include <iostream>
```

su programa también deberá incluir lo siguiente, ya sea al comienzo del archivo o al principio del cuerpo de la función que utilice `ios::app`:

```
using namespace std;
```

Nombres de archivo como entradas (*Opcional*)

Hasta aquí hemos escrito los nombres literales de nuestros archivos de entrada y salida en el código de nuestros programas. Lo hicimos dando el nombre de archivo como argumento en una llamada a la función `open`, como en el siguiente ejemplo:

```
flujo_in.open("archin.dat");
```

En ocasiones, esto puede resultar inconveniente. Por ejemplo, el programa del cuadro 5.2 lee números del archivo `archin.dat` y envía su suma al archivo `archout.dat`. Si queremos realizar el mismo cálculo con los números de otro archivo llamado `archin2.dat` y escribir la suma de dichos números en otro archivo llamado `archout2.dat`, tendremos que cambiar los nombres de archivo en las dos llamadas a la función miembro `open` y luego recompilar el programa. Una alternativa preferible es escribir el programa de modo que pida al usuario que teclee los nombres de los archivos de entrada y/o de salida. De esta forma, el programa puede usar diferentes archivos cada vez que se ejecuta.

CUADRO 5.3 *Anexando a un archivo (Opcional)*

```
//Añada datos al final del archivo datos.txt.
#include <fstream>
#include <iostream>

int main()
{
    using namespace std;

    cout << "Abriendo datos.txt para anexar.\n";
    ofstream fout;
    fout.open("datos.txt", ios::app);
    if (fout.fail())
    {
         cout << "No pudo abrirse el archivo de entrada.\n";
         exit(1);
    }
    fout << "5 6 recoger varitas.\n"
         << "7 8 no es C++ maravilloso!\n";

    fout.close();
    cout << "Termino de anexar datos al archivo.\n";

    return 0;
}
```

Diálogo de ejemplo

data.txt
(Antes de ejecutar el programa.)

```
1 2 ensucia mi zapato.
3 4 patea la puerta.
```

data.txt
(Después de ejecutarse el programa.)

```
1 2 ensucia mi zapato.
3 4 patea la puerta.
5 6 recoge varitas.
7 8 no es C++ maravilloso!
```

Salida de pantalla

```
Abriendo datos.txt para añadir datos.
Termino de anexar datos al archivo.
```

Añadir a un archivo

Si quiere añadir datos a un archivo de manera que queden debajo de la información existente, abra el archivo como se muestra a continuación.

Sintaxis

Output_Stream.`open`(*Nombre_Archivo*, `ios::app`);

Ejemplo

```
ofstream outStream;
outStream.open("importante.txt", ios::app);
```

Un nombre de archivo es una *cadena* y no estudiaremos el manejo de cadenas con detalle sino hasta el capítulo 10. No obstante, es fácil aprender lo suficiente acerca de las cadenas como para poder escribir programas que acepten un nombre de archivo como entrada. Una **cadena** no es más que una sucesión de caracteres. Ya hemos usado valores de cadena en instrucciones de salida como el siguiente:

cadena

```
cout << "Esta es una cadena.";
```

También hemos usado valores de cadena como argumentos de la función miembro `open`. Siempre que escribimos una cadena literal, como en el instrucción `cout` anterior, debemos encerrar la cadena entre comillas.

Para poder leer un nombre de archivo e introducirlo en un programa necesitamos una variable capaz de contener una cadena. Las variables que contienen un valor de cadena se declaran como en el siguiente ejemplo:

variable de cadena

```
char nombre_archivo[16];
```

Esta declaración es parecida a la que usamos para declarar una variable de tipo `char`, sólo que el nombre de la variable va seguido de un entero entre corchetes que especifica el número máximo de caracteres que puede haber en una cadena almacenada en esa variable. En realidad, este número debe ser *uno más que el número máximo de caracteres que tiene el valor de cadena*. Así, en nuestro ejemplo, la variable `nombre_archivo` puede contener cualquier cadena que tenga 15 o menos caracteres. El nombre `nombre_archivo` se puede sustituir por cualquier otro identificador (que no sea una palabra clave) y el número `16` se puede sustituir por cualquier otro entero positivo.

Podemos introducir un valor de cadena y colocarlo en una variable de cadena del mismo modo que introducimos valores de otros tipos. Por ejemplo, consideremos el siguiente fragmento de código:

entrada de cadena

```
cout << "Escribe el nombre de archivo (max. 15 caracteres):\n";
cin >> nombre_archivo;
cout << "Muy bien, editare el archivo " << nombre_archivo << endl;
```

Un posible diálogo para este código es:

```
"Escribe el nombre de archivo (max. 15 caracteres):
miarch.dat
Muy bien, editare el archivo miarch.dat
```

variables de cadena como argumentos de `open`

Una vez que el programa ha leído el nombre de un archivo y lo ha guardado en una variable de cadena, como la variable `nombre_archivo`, puede usar esta variable de cadena como argumento de la función miembro `open`. Por ejemplo, lo que sigue conecta el flujo de archivo de entrada `flujo_in` al archivo cuyo nombre está almacenado en la variable `nombre_archivo` (y usa la función miembro `fail` para verificar que la apertura se logró):

```
ifstream flujo_in;
flujo_in.open(nombre_archivo);
if (flujo_in.fail())
{
    cout << "No pudo abrirse el archivo de entrada.\n";
    exit(1);
}
```

Observe que al usar una variable de cadena como argumento de la función miembro `open` no se usan comillas.

En el cuadro 5.4 hemos reescrito el programa del cuadro 5.2 de modo que reciba sus entradas de y envíe sus salidas a los archivos que el usuario especifique. Los nombres de los archivos de entrada y de salida se leen del teclado y se colocan en las variables de cadena `nombre_arch_in` y `nombre_arch_out`, y luego esas variables se usan como argumentos de llamadas a la función miembro `open`. Observe la declaración de las variables de cadena. Es preciso incluir un número entre corchetes después de cada nombre de variable de cadena, como hicimos en el cuadro 5.4.

¡Advertencia!

Las variables de cadenas no son variables ordinarias y no se pueden usar en todas las formas en que podemos usar variables ordinarias. En particular, no podemos usar una instrucción de asignación para modificar el valor de una variable de cadena.

5.2 Herramientas para flujos de E/S

Los verás en una hermosa página en cuarto, donde un nítido arroyuelo de texto serpenteará por una pradera de margen.

Richard Brinsley Sheridan, *The School for Scandal*

Formateo de salidas con funciones de flujos

formato

La organización de las salidas de un programa es su **formato**. En C++ podemos controlar el formato con comandos que determinan detalles como el número de espacios entre los elementos y el número de dígitos después del punto decimal. Ya usamos tres instrucciones de formateo de salidas cuando aprendimos la fórmula para desplegar cantidades monetarias de la forma estándar con dos dígitos después del punto decimal (y no en notación `e`). Antes de desplegar cantidades monetarias, insertamos la siguiente "fórmula mágica" en nuestro programa:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

CUADRO 5.4 *Introducción de un nombre de archivo (Opcional) (parte 1 de 2)*

```
//Lee tres números del archivo especificado por el usuario, los suma
//y escribe la suma en otro archivo especificado por el usuario.
#include <fstream>
#include <iostream>
#include <cstdlib>

int main()
{
    using namespace std;
    char nombre_arch_in[16], nombre_arch_out[16];
    ifstream flujo_in;
    ofstream flujo_out;

    cout << "Sumara tres numeros tomados de un archivo de\n"
         << "entrada y escribira la suma en un archivo de salida.\n";
    cout << "Escribe el nombre de archivo de entrada (max. 15 caracteres):\n";
    cin >> nombre_arch_in;
    cout << "Escribe el nombre de archivo de salida (max. 15 caracteres):\n";
    cin >> nombre_arch_out;
    cout << "Leere numeros del archivo "
         << nombre_arch_in << " y\n"
         << "colocare la suma en el archivo "
         << nombre_arch_out << endl;

    flujo_in.open(nombre_arch_in);
    if (flujo_in.fail())
        cout << "No pudo abrirse el archivo de entrada.\n";
    flujo_out.open(nombre_arch_out);
    if (flujo_out.fail())
        cout << "No pudo abrirse el archivo de salida.\n";
```

CUADRO 5.4 *Introducción de un nombre de archivo (Opcional) (parte 2 de 2)*

```
    int primero, segundo, tercero;
    flujo_in >> primero >> segundo >> tercero;
    flujo_out << "La suma de los primeros tres \n"
              << "numeros de " << nombre_arch_in << endl
              << " es " << (primero + segundo + tercero)
              << endl;

    flujo_in.close();
    flujo_out.close();

    cout << "Fin del programa.\n";
    return 0;
}
```

numeros.dat

(El programa no lo modifica.)

```
1
2
3
4
```

suma.dat

(Después de ejecutarse el programa.)

```
La suma de los primeros
tres números de
numeros.dat es 6
```

Diálogo de ejemplo

```
Sumare tres numeros tomados de un archivo de
entrada y escribire la suma en un archivo de salida.
Escribe el nombre de archivo de entrada (max. 15 caracteres):
numeros.dat
Escribe el nombre de archivo de salida (max. 15 caracteres):
suma.dat
Leere numeros del archivo numeros.dat y
colocare la suma en el archivo suma.dat
Fin del programa.
```

Ahora que conocemos la notación de objetos para flujos podemos explicar esta fórmula mágica y unos cuantos comandos de formateo más.

Lo primero que debemos entender es que podemos usar estos comandos de formato con cualquier flujo de salida. Si nuestro programa está enviando salidas a un archivo conectado a un flujo de salida de nombre `flujo_out`, podemos usar estos mismos comandos para asegurar que los números con punto decimal se escriban en la forma en que normalmente escribimos cantidades de dinero. Basta con insertar lo siguiente en nuestro programa:

```
flujo_out.setf(ios::fixed);
flujo_out.setf(ios::showpoint);
flujo_out.precision(2);
```

Para explicar esta fórmula mágica consideraremos las instrucciones en el orden inverso.

precision

Todo flujo de salida tiene una función miembro llamada `precision`. Cuando nuestro programa ejecuta una llamada a `precision` como la que acabamos de mostrar para el flujo `flujo_out`, a partir de ese punto cualquier número que tenga punto decimal y se envíe como salida a ese flujo se escribirá con un total de dos cifras significativas, o con dos dígitos después del punto decimal, dependiendo de cuándo se haya escrito nuestro compilador. Las que siguen son posibles salidas de un compilador que establece dos cifras significativas:

```
23.    2.2e7    2.2    6.9e-1    0.00069
```

Las que siguen son posibles salidas de un compilador que establece dos dígitos después del punto decimal:

```
23.56    2.26e7    2.21    0.69    0.69e-4
```

En este libro supondremos que el compilador fija dos dígitos después del punto decimal.

Una llamada a `precision` aplica únicamente al flujo nombrado en la llamada. Si nuestro programa tiene otro flujo de salida llamado `flujo_out_dos`, la llamada a `flujo_out.precision` afectará las salidas al flujo `flujo_out`, pero no a las que se envíen a `flujo_out_dos`. Desde luego, también podemos invocar a `precision` con el `flujo flujo_out_dos`; hasta podemos especificar un número distinto de dígitos para los números que se envíen a `flujo_out_dos`, como se muestra a continuación:

```
flujo_out_dos.precision(3);
```

Las otras instrucciones de formateo de nuestra fórmula mágica son un poco más complicadas que la función miembro `precision`. A continuación presentamos dos llamadas a la función miembro `setf` con el flujo `flujo_out` como objeto invocador:

`setf`

```
flujo_out.setf(ios::fixed);
flujo_out.setf(ios::showpoint);
```

`setf` es una abreviatura de *set flags*, que significa establecer (o "encender") banderas. Una **bandera** es algo que indica que se debe efectuar algo de una de dos posibles maneras. Si damos una bandera como argumento de `setf`, la bandera le indicará a la computadora que escriba las salidas en ese flujo de alguna forma específica. El efecto sobre el flujo depende de la bandera.

bandera

En el ejemplo anterior hay dos llamadas a la función `setf` las cuales establecen las dos banderas `ios::fixed` e `ios::showpoint`. La bandera `ios::fixed` hace que el flujo

`ios::fixed`

envíe a la salida los números de tipo `double` en lo que se conoce como **notación de punto fijo**, que es una forma elegante de referirnos a la forma en que normalmente escribimos los números. Si se "establece" la bandera `ios::fixed` (con una llamada a `setf`), todos los números de punto flotante (como los de tipo `double`) que se envíen a ese flujo se escribirán en la notación ordinaria, no en la notación `e`.

notación de punto fijo

La bandera `ios::showpoint` le dice al flujo que siempre incluya un punto decimal en los números de punto flotante, como los de tipo `double`. Así pues, si el número que se enviará a la salida tiene el valor `2.0`, se enviará a la salida como `2.0` y no simplemente como `2`; es decir, la salida incluirá el punto decimal aunque todos los dígitos después del punto decimal sean cero. En el cuadro 5.5 se describen algunas banderas comunes y las acciones que causan.

ios::showpoint

CUADRO 5.5 *Banderas de formateo para* `setf`

Bandera	Significado	Estado predeterminado
`ios::fixed`	Si esta bandera está establecida, los números de punto flotante no se escriben en notación `e`. (Al establecer esta bandera se desactiva automáticamente la bandera `ios::scientific`.)	`desactivada`
`ios::scientific`	Si esta bandera está establecida, los números de punto flotante se escriben en notación `e`. (Establecer esta bandera no establece automáticamente la bandera `ios::fixed`.) Si no están establecidas ni `ios::fixed` ni `ios::scientific`, el sistema decidirá cómo desplegar cada número.	`desactivada`
`ios::showpoint`	Si esta bandera está establecida, siempre se muestran el punto decimal y los ceros a la izquierda de los números de punto flotante; si no, un número que sólo tiene ceros después del punto decimal podría desplegarse sin el punto y sin los ceros.	`desactivada`
`ios::showpos`	Si esta bandera está establecida, se coloca un signo de más antes de los valores positivos enteros.	`desactivada`
`ios::right`	Si esta bandera está establecida y se da algún valor de anchura de campo con una llamada a la función miembro `width`, el siguiente número que se despliegue estará en el extremo derecho del espacio especificado por `width`. En otras palabras, si sobran espacios se colocarán antes del elemento desplegado. (Establecer esta bandera desactiva automáticamente la bandera `ios::left`.)	`establecida`
`ios::left`	Si esta bandera está establecida y se da algún valor de anchura de campo con una llamada a la función miembro `width`, el siguiente número que se despliegue estará en el extremo izquierdo del espacio especificado por `width`. En otras palabras, si sobran espacios se colocarán después del elemento desplegado. (Establecer esta bandera desactiva automáticamente la bandera `ios::right`.)	`desactivada`

ios::showpos

Otra bandera útil es `ios::showpos`. Si esta bandera se establece para un flujo, los números positivos que se envíen a ese flujo se escribirán precedidos por el signo más. Si queremos que aparezca un signo de más a la izquierda de los números positivos, insertamos lo siguiente:

```
cout.setf(ios::showpos);
```

El signo menos aparece antes de los números negativos sin necesidad de establecer banderas.

width

Una función de formateo de uso muy común es `width`. Por ejemplo, consideremos la siguiente llamada a `width` hecha por el flujo `cout`:

```
cout << "Comenzar aqui";
cout.width(4);
cout << 7 << endl;
```

Este código hará que aparezca la siguiente línea en la pantalla

```
Comenzar aqui   7
```

Esta salida tiene exactamente tres espacios entre la letra `'i'` y el número `7`. La función `width` le dice al flujo cuántos espacios debe usar al enviar un elemento a la salida. En este caso el elemento, que es el número `7`, ocupa un solo espacio, y `width` indicó que hay que usar cuatro espacios, así que tres de los espacios están en blanco. Si la salida requiere más espacio que el que se especificó en el argumento de `width`, se usará tanto espacio adicional como se necesite. Siempre se despliega el elemento completo, sea cual sea el argumento que hayamos dado a `width`.

Una llamada a `width` aplica sólo al siguiente elemento que se envía a la salida. Si queremos desplegar doce números, utilizando cuatro espacios para cada número, debemos invocar `width` doce veces. Si esto es una lata, podríamos usar el manipulador `setw` que se describe en la subsección siguiente.

unsetf

Cualquier bandera establecida puede desactivarse. Para desactivar una bandera usamos la función `unsetf`. Por ejemplo, lo que sigue hará que el programa deje de incluir signos de más antes de los enteros positivos que se envían al flujo `cout`:

```
cout.unsetf(ios::showpos);
```

Manipuladores

manipulador

Un **manipulador** es una función que se invoca de manera no tradicional. A su vez, la función manipuladora invoca una función miembro. Los manipuladores se colocan después del operador de inserción `<<`, como si la función manipuladora fuera un elemento que se enviará a la salida. Al igual que las funciones tradicionales, los manipuladores pueden tener argumentos o no. Ya hemos visto un manipulador, `endl`. En esta subsección veremos dos manipuladores llamados `setw` y `setprecision`.

setw

El manipulador `setw` y la función miembro `width` (que ya vimos) hacen exactamente lo mismo. Invocamos el manipulador `setw` escribiéndolo después del operador de inserción `<<`, como si lo fuéramos enviar al flujo `cout`, y éste a su vez invoca a la función miembro `width`. Por ejemplo, lo que se muestra a continuación despliega los números `10`, `20` y `30` usando las anchuras de campo especificadas:

```
cout << "Inicio" << setw(4) << 10
     << setw(4) << 20 << setw(6) << 30;
```

Terminología de banderas

¿Por qué llamamos *banderas* a los argumentos de `setf`, como `ios::showpoint`? ¿Y qué significa la extraña notación `ios::`?

Originalmente se usó la palabra **bandera** para referirse a bits individuales que servían para indicar algo, dependiendo de si contenían 1 o 0, por analogía con una bandera real que indica algo al estar establecida o desactivada. En inglés se dice que un bit se "establece" (*set*) si se coloca un 1 en él, y se "despeja" (*clear*) si se coloca un 0 en él. De ahí que se acostumbre decir que la bandera se "establece" cuando se "enciende" o "activa". En español preferimos usar los términos "establecer" y "desactivar" porque son más evocativos. Así pues, cuando la bandera `ios::showpoint` (por ejemplo) está establecida (es decir, cuando es argumento de `setf`), el flujo que invocó la función `setf` se comporta como se describe en el cuadro 5.5; cuando se establece cualquier otra bandera (es decir, cuando se da como argumento de `setf`), esto indica al flujo que se comporte como se especifica en el cuadro 5.5 para esa bandera.

La explicación de la notación `ios::` es un tanto prosaica. La palabra `ios` es abreviatura de ***i**nput/**o**utput **s**tream*, e indica que el significado de términos como `fixed` o `showpoint` es el que tienen cuando se usan con un *flujo de entrada o salida*. La notación `::` significa "usar el significado de lo que sigue a `::` en el contexto de lo que precede a `::`". Hablaremos más acerca de esta notación `::` posteriormente.

La instrucción anterior produce las siguientes salidas:

```
Inicio  10  20    30
```

(Hay dos espacios antes del `10`, dos antes de `20` y cuatro antes del `30`.)

El manipulador `setprecision` hace exactamente lo mismo que la función miembro `precision` (que ya vimos). Sin embargo, las llamadas a `setprecision` se escriben después del operador de inserción `<<`, tal como se hace con el manipulador `setw`. Por ejemplo, lo que sigue despliega los números que se listan empleando el número de dígitos después del punto decimal que indica la llamada a `setprecision`:

`setprecision`

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout << "$" << setprecision(2) << 10.3 << endl
     << "$" << 20.5 << endl;
```

La instrucción anterior produce esta salida:

```
$10.30
$20.50
```

Al igual que con la función miembro `precision`, cuando fijamos el número de dígitos después del punto decimal empleando el manipulador `setprecision` el efecto persiste hasta que otra llamada a `setprecision` o `precision` cambia dicho número de dígitos.

iomanip

Para emplear los manipuladores `setw` o `setprecision` es preciso incluir la siguiente directiva en el programa:

```
#include <iomanip>
```

Su programa también debe incluir lo siguiente:

```
using namespace std;
```

Ejercicios de AUTOEVALUACIÓN

12. ¿Qué salidas se producen cuando se ejecutan las siguientes líneas (suponiendo que forman parte de un programa completo y correcto con las directivas `include` apropiadas)?

```
cout << "*";
cout.width(5);
cout << 123
     << "*" << 123 << "*" << endl;
cout << "*" << setw(5) << 123
     << "*" << 123 << "*" << endl;
```

13. ¿Qué salidas se producen cuando se ejecutan las siguientes líneas (suponiendo que forman parte de un programa completo y correcto con las directivas `include` apropiadas)?

```
cout << "*" << setw(5) << 123;
cout.setf(ios::left);
cout << "*" << setw(5) << 123;
cout.setf(ios::right);
cout << "*" << setw(5) << 123 << "*" << endl;
```

14. ¿Qué salidas se producen cuando se ejecutan las siguientes líneas (suponiendo que forman parte de un programa completo y correcto con las directivas `include` apropiadas)?

```
cout << "*" << setw(5) << 123 << "*";
     << 123 << "*" << endl;
cout.setf(ios::showpos);
cout << "*" << setw(5) << 123 << "*";
     << 123 << "*" << endl;
cout.unsetf(ios::showpos);
cout.setf(ios::left);
cout << "*" << setw(5) << 123 << "*";
     << setw(5) << 123 << "*" << endl;
```

15. ¿Qué salidas se enviarán al archivo `cosas.dat` cuando se ejecutan las siguientes líneas (suponiendo que forman parte de un programa completo y correcto con las directivas `include` apropiadas)?

```
ofstream fout;
fout.open("cosas.dat");
fout << "*" << setw(5) << 123 << "*"
     << 123 << "*" << endl;
fout.setf(ios::showpos);
```

```
fout << "*" << setw(5) << 123 << "*"
     << 123 << "*" << endl;
fout.unsetf(ios::showpos);
fout.setf(ios::left);
fou << "*" << setw(5) << 123 << "*"
    << setw(5) << 123 << "*" << endl;
```

16. ¿Qué salidas se producen cuando se ejecuta la siguiente línea (suponiendo que forma parte de un programa completo y correcto con las directivas `include` apropiadas)?

```
cout << "*" << setw(3) << 12345 << "*" endl;
```

17. Al formatear salidas, se usan las siguientes constantes de bandera con la función miembro `setf`. ¿Qué efecto tiene cada una?

a) `ios::fixed`
b) `ios::scientific`
c) `ios::showpoint`
d) `ios::showpos`
e) `ios::right`
f) `ios::left`

18. He aquí un segmento de código que lee la entrada de `archin.dat` y envía la salida a `archout.dat`. ¿Qué cambios hay que efectuar para que las salidas vayan a la pantalla? (Las entradas seguirán proviniendo de `archin.dat`.)

```
//Problema para autoevaluación. Copia tres números int
//de un archivo a otro.
#include <fstream>
int main()
{
    using namespace std;

    ifstream flujoin;
    ofstream flujoout;

    flujoin.open("archin.dat");
    flujoout.open("archout.dat");
    int primero, segundo, tercero;
    flujoin >> primero >> segundo >> tercero;
    flujoout << "La suma de los primeros 3" << endl
             << "numeros de archin.dat es " << endl
             << (primero + segundo + tercero) << endl;
    flujoin.close();
    flujoout.close();
    return 0;
}
```

Flujos como argumentos de funciones

los parámetros de flujo deben ser de llamada por referencia

Un flujo puede ser argumento de una función. La única restricción es que el parámetro formal de la función debe ser de llamada por referencia. Un parámetro de flujo no puede ser de llamada por valor. Por ejemplo, la función `asear` del cuadro 5.6 tiene dos parámetros de flujo: uno es de tipo `ifstream` y es para un flujo conectado a un archivo de entrada; el otro es de tipo `ofstream` y es para un flujo conectado a un archivo de salida. Analizaremos las otras características del programa del cuadro 5.6 en las dos subsecciones que siguen.

TIP DE PROGRAMACIÓN

Verifique el fin de un archivo

> *Eso es todo lo que hay; no hay más.*
> Ethel Barrymore (1879-1959)

Cuando escribimos un programa que recibe sus entradas de un archivo, es común querer que el programa lea todos los datos del archivo. Por ejemplo, si el archivo contiene números, podríamos querer que el programa calcule la media de todos los números del archivo. Puesto que podríamos ejecutar el programa con diferentes archivos de datos en diferentes ocasiones, el programa no puede suponer que sabe cuántos números hay en el archivo. Nos gustaría escribir el programa de modo que siga leyendo números del archivo hasta que no queden más números que leer. Si `flujo_in` es un flujo conectado al archivo de entrada, el algoritmo para calcular la media se puede expresar así:

```
double siguiente, suma = 0;
int conteo = 0;
while (Todavía hay números que leer)
{
    flujo_in >> siguiente;
    suma = suma + siguiente;
    conteo++;
}
La media es suma/conteo.
```

Este algoritmo ya es casi todo código C++, pero falta expresar la siguiente prueba en C++:

```
(Todavía hay números que leer)
```

Aunque tal vez no parezca correcta a primera vista, una forma de expresar la prueba anterior es ésta:

```
(flujo_in >> siguiente)
```

CUADRO 5.6 *Formateo de salidas (parte 1 de 3)*

```
//Ilustra las instrucciones de formateo de salidas.
//Lee todos los números del archivo sucios.dat y los escribe en la
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <iomanip>
using namespace std;

//pantalla y en el archivo limpios.dat con un formato uniforme.
//se necesita para setw
//Los parámetros de flujo deben ser de llamada por referencia
void asear(ifstream& arch_sucio, ofstream& arch_limpio,
          int decimales, int anchura_campo);
//Precondición: Los flujos arch_sucio y arch_limpio se han
//conectado a archivos usando la función open.
//Postcondición: Los números del archivo conectado a arch_sucio se han
//escrito en la pantalla y en el archivo conectado al flujo arch_limpio.
//Se escribe un número por línea, en notación de punto fijo (no en
//notación e), con decimales dígitos después del punto decimal; cada
//número va precedido de un signo de más o menos y está en un campo de
//anchura anchura_campo. (Esta función no cierra el archivo.)

int main()
{
    ifstream fin;
    ofstream fout;

    fin.open("sucios.dat");
    if (fin.fail())
    {
        cout << "No se pudo abrir el archivo de entrada.\n";
        exit(1);
    }
    fout.open("limpios.dat");
    if (fout.fail())
    {
        cout << "No se pudo abrir el archivo de salida.\n";
        exit(1);
    }
```

se necesita para `setw`

Los parámetros de flujo deben ser de llamada por referencia

CUADRO 5.6 *Formateo de salidas (parte 2 de 3)*

```
    asear(fin, fout, 5, 12);

    fin.close();
    fout.close();

    cout << "Fin del programa.\n";
    return 0;
}

//Usa iostream, fstream y iomanip:
void asear(ifstream& arch_sucio, ofstream& arch_limpio,
           int decimales, int anchura_campo)
{
    arch_limpio.setf(ios::fixed);<-------- no en notación e
    arch_limpio.setf(ios::showpoint);<-------- mostrar punto decimal
    arch_limpio.setf(ios::showpos);<-------- mostrar signo +
    //no en notación e
    //mostrar punto decimal
    //mostrar signo +
    arch_limpio.precision(decimales);
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.setf(ios::showpos);
    cout.precision(decimales);

    double siguiente;
    while (arch_sucio >> siguiente)<-------- Se satisface si hay un número más que leer.
    //Se satisface si hay un número más que leer.
    {
        cout << setw(anchura_campo) << siguiente << endl;
        arch_limpio << setw(anchura_campo) << siguiente << endl;
    }
}
```

CUADRO 5.6 *Formateo de salidas (parte 3 de 3)*

sucios.dat
(El programa no lo modifica.)

```
10.37     −9.89897
2.313   −8.950  15.0

  7.33333    92.8765
−1.237568432e2
```

limpios.dat
(Después de ejecutarse el programa.)

```
+10.37000
 −9.89897
 +2.31300
 −8.95000
+15.00000
 +7.33333
+92.87650
−123.75684
```

Salidas a la pantalla

```
+10.37000
 −9.89897
 +2.31300
 −8.95000
+15.00000
 +7.33333
+92.87650
−123.75684
Fin del programa.
```

De modo que el algoritmo anterior se puede reescribir como el siguiente código C++ (más una última línea en pseudocódigo que no nos interesa aquí):

```
double siguiente, suma = 0;
int conteo = 0;
while (flujo_in >> siguiente)
{
    suma = suma + siguiente;
    conteo++;
}
```

La media es `suma/conteo`.

Observe que el cuerpo del ciclo no es idéntico como lo habíamos escrito en nuestro pseudocódigo. Puesto que `flujo_in >> siguiente` ya está en la expresión booleana, ya no aparece en el cuerpo del ciclo.

El ciclo anterior tal vez parezca un tanto extraño, porque `flujo_in >> siguiente` es tanto la forma de introducir un número desde el flujo `flujo_in` como la expresión booleana que controla el ciclo *`while`*. Una expresión en la que interviene el operador de extrac-

ción >> es al mismo tiempo una acción y una condición booleana.[2] (Es una instrucción que toma un número del flujo de entrada y también es una expresión booleana que se satisface o no.) Si hay otro número que introducir, se lee ese número y la expresión booleana se satisface, así que el cuerpo del ciclo se ejecuta una vez más. Si no hay más números que leer, no se introduce nada y la expresión booleana no se satisface, así que el ciclo termina. En este ejemplo el tipo de la variable de entrada `siguiente` fue *`double`*, pero este método de verificar si se llegó al final del archivo funciona igual con otros tipos de datos, como *`int`* y *`char`*.

Nota sobre namespaces

Hemos tratado de mantener a nuestras directivas locales *`using`* para una definición de función. Esto es un objetivo admirable pero nos encontramos con un problema, funciones cuyo tipo de parámetro se encuentra en namespace. En los próximos ejemplos necesitaremos los nombres de tipo de flujos que se encuentran en namespace `std`. Así que necesitaremos una directiva *`using`* (o algo) fuera del cuerpo de la definición de función para que C++ entienda los nombres de tipo de parámetros como `ifstream`. La manera más sencilla es colocar una directiva *`using`* al principio del archivo (después de las directivas `include`). Esto ya lo realizamos en el cuadro 5.6.

Colocar una directiva *`using`* al principio del archivo es la solución más sencilla; sin embargo, según los expertos no es la más recomendable, ya que no permitiría el uso de 2 namespaces que tengan nombres en común y éste es el propósito de los namespaces. En este punto no hay problema ya que sólo estamos utilizando el namespace `std`.[3] En el capítulo 9 le enseñaremos otra forma de sortear este problema con los parámetros y namespaces. Esta nueva forma le permitirá utilizar cualquier clase de namespaces múltiples.

Muchos programadores prefieren colocar las directivas *`using`* al principio del archivo del programa, consideremos el siguiente ejemplo:

```
using namespace std;
```

Varios de los programas de este libro no tienen la directiva *`using`* al principio del archivo, sino al principio de cada definición de función que necesite el namespace `std` (inmediatamente después de la llave de apertura). Un ejemplo de esto lo vemos en el cuadro 5.3 y un mejor ejemplo lo podemos ver en el cuadro 4.11. Todos los programas que han aparecido hasta ahora (en el libro) y los que siguen se comportarían de la misma forma si sólo hubiera una directiva *`using`* para cada espacio de nombres `std` y esa directiva se colocara inmediatamente después de las directivas `include` como en el cuadro 5.6. Para el espacio de nombre `std` la directiva *`using`* se puede colocar sin problema al principio del archivo (en casi todos los casos). Para algunos otros espacios de nombre una sola directiva *`using`* no siempre será suficiente, pero no verá ninguno de estos casos por algún tiempo.

[2] Técnicamente, la condición booleana funciona como sigue. La sobrecarga del operador >> para las clases de flujo de entrada se efectúa con funciones asociadas al flujo. Esta función se llama `operator >>`. El valor devuelto por esta función operador es una referencia a un flujo de entrada (`istream&` o `ifstream&`). Se proporciona una función para convertir la referencia a un flujo en un valor `bool`. El valor resultante es `true` si el flujo logra extraer datos, y `false` en caso contrario.

[3] Estamos usando dos espacios de nombre: el espacio de nombre `std` y uno llamado el espacio de nombre global, el cual es uno que consiste en todos los nombres que no se encuentran en algún otro nombre de espacio. Pero no nos preocupemos por ahora con este detalle técnico.

Abogamos por poner las directivas *using* dentro de las definiciones de función (o dentro de cualesquiera otras pequeñas unidades de código) para que no interfiera con otras posibles directivas *using*. Esto lo capacita para utilizar correctamente los espacios de nombres cuando escriba un código más complicado en su carrera de programador. Mientras tanto, nosotros mismos violaremos algunas veces las reglas cuando el seguirlas se convierta en una molestia para los demás temas que vayamos discutiendo. Si usted está tomando un curso haga lo que el instructor le diga. De todas maneras tiene cierta latitud en donde coloquen sus directivas *using*.

EJEMPLO DE PROGRAMACIÓN

Aseo de un formato de archivo

El programa del cuadro 5.6 recibe sus entradas del archivo `sucios.dat` y escribe sus salidas, en un formato uniforme, tanto en la pantalla como en el archivo `limpios.dat`. El programa copia números del archivo `sucios.dat` en el archivo `limpios.dat`, pero usa instrucciones de formateo para escribirlos de forma aseada. Los números se escriben uno en cada línea en un campo de anchura 12, lo que implica que cada número va precedido por suficientes espacios en blanco para que los espacios y el número ocupen 12 posiciones. Los números se escriben en notación ordinaria; es decir, no se escriben en notación `e`. Cada número se escribe con cinco dígitos después del punto decimal y con un signo más o menos. Las salidas a la pantalla son las mismas que al archivo `limpios.dat`, sólo que la salida a la pantalla tiene una línea adicional que anuncia que el programa está terminando. El programa usa una función llamada `asear` que tiene parámetros formales para el flujo del archivo de entrada y el flujo del archivo de salida.

Ejercicios de AUTOEVALUACIÓN

19. ¿Qué salidas se producen cuando se ejecutan las siguientes líneas, suponiendo que el archivo `lista.dat` contiene los datos que se muestran (y suponiendo que las líneas forman parte de un programa completo y correcto con las directivas `include` apropiadas)?

```
ifstream ins;
ins.open("lista.dat");
int conteo = 0, siguiente;
while (ins >> siguiente)
{
    conteo++;
    cout << siguiente << endl;
}
ins.close();
cout << conteo;
```

El archivo `lista.dat` contiene estos tres números (y nada más):

```
1  2
3
```

20. Escriba la definición de una función *void* llamada a_pantalla. La función a_pantalla tiene un parámetro formal llamado flujo_archivo, que es de tipo ifstream. La precondición y la postcondición de la función son:

```
//Precondición: El flujo flujo_archivo se ha conectado a un
//archivo con una llamada a la función miembro open. El archivo
//contiene una lista de enteros (y nada más).
//Postcondición: Los números del archivo conectado a
//flujo_archivo se han escrito en la pantalla, uno por línea.
//(Esta función no cierra el archivo.)
```

21. (Este ejercicio es para los que estudiaron la sección opcional sobre **Nombres de archivo como entradas**.) Suponga que le dan la siguiente declaración de variable de cadena y la siguiente instrucción de entrada.

```
#include <iostream>
using namespace std;
// ...
char nombre[21];
cout << nombre;
```

Suponga que este segmento de código forma parte de un programa correcto. ¿Qué longitud máxima puede tener un nombre introducido en la variable de cadena nombre?

5.3 E/S de caracteres

Polonio: ¿Qué leéis, mi señor?
Hamlet: Palabras, palabras, palabras.
William Shakespeare, *Hamlet*

Todos los datos se introducen y se envían a la salida como caracteres. Cuando un programa envía a la salida el número 10, en realidad lo que envía son los dos caracteres '1' y '0'. Así mismo, cuando el usuario quiere teclear el número 10, teclea el carácter '1' seguido del carácter '0'. Que la computadora interprete este 10 como dos caracteres o como el número diez dependerá de cómo se escribió el programa. No obstante, sea como sea que se haya escrito el programa, el hardware de la computadora siempre lee los caracteres '1' y '0', no el número diez. Esta conversión entre caracteres y números suele ser automática, por lo que no tenemos que preocuparnos por tales detalles, pero a veces toda esta ayuda automática estorba. Por ello, C++ ofrece algunos recursos de bajo nivel para la entrada y salida de datos de caracteres. Estos recursos de bajo nivel no realizan conversiones automáticas, por lo que nos permiten pasar por alto los recursos automáticos y realizar la entrada/salida a nuestro total arbitrio. Incluso podríamos escribir funciones de entrada y salida capaces de leer y escribir números en notación de números romanos, si no tenemos nada mejor que hacer.

Las funciones miembro get y put

La función get permite a un programa leer un carácter de entrada y guardarlo en una variable de tipo *char*. Todos los flujos de entrada, sean flujos de archivo de entrada o el flujo cin, tienen una función miembro llamada get. Describiremos a get como función miembro del flujo cin, pero se comporta exactamente de la misma manera con flujos de

archivo de entrada que con `cin`, por lo que el lector puede aplicar todo lo que digamos acerca de `get` a los flujos de archivo de entrada, además de al flujo `cin`.

Hasta ahora, hemos usado `cin` con el operador de extracción `>>` para leer un carácter de entrada (o cualquier otra entrada). Cuando usamos el operador de extracción `>>`, como hemos estado haciendo, se llevan a cabo algunas tareas automáticas, como saltarse los espacios en blanco. Con la función miembro `get`, nada se hace automáticamente. Por ejemplo, si queremos saltarnos los espacios en blanco usando `cin.get`, tendremos que escribir código para leer y desechar los espacios en blanco.

`Nombre_de_Flujo.get`

La función miembro `get` recibe un argumento, que debe ser una variable de tipo `char`. Ese argumento recibe el carácter de entrada que se lee del flujo de entrada. Por ejemplo, lo que sigue lee el siguiente carácter de entrada del teclado y lo guarda en la variable `siguiente_simbolo`:

```
char siguiente_simbolo;
cin.get(siguiente_simbolo);
```

lectura de espacios y `'\n'`

Es importante tomar nota de que nuestro programa puede leer cualquier carácter de esta manera. Si el siguiente carácter introducido es un espacio en blanco, el código anterior no lo pasará por alto, sino que lo leerá y hará que el valor de `siguiente_simbolo` sea el carácter de espacio en blanco. Si el siguiente carácter es el carácter de salto de línea `'\n'`, es decir, si el programa ha llegado al final de una línea de entrada, la llamada anterior a `cin.get` hará que el valor de `siguiente_simbolo` sea `'\n'`. Aunque lo escribimos como dos símbolos en C++ `'\n'` es un solo carácter. Con la función miembro `get`, el carácter `'\n'` se puede introducir y enviar a la salida igual que cualquier otro carácter. Por ejemplo, supongamos que nuestro programa contiene el código siguiente:

```
char c1, c2, c3;
cin.get(c1);
cin.get(c2);
cin.get(c3);
```

y supongamos que tecleamos estas dos líneas de entrada para que ese código las lea:

```
AB
CD
```

Es decir, supongamos que tecleamos `AB`, oprimimos la tecla Entrar, tecleamos `CD` y oprimimos la tecla Entrar otra vez. Como es de esperar, se asigna el valor `'A'` a `c1` y el valor `'B'` a `c2`. Nada nuevo aquí; pero cuando este código llena la variable `c3` las cosas son diferentes de lo que serían si hubiéramos usado el operador de extracción `>>` en lugar de la función miembro `get`. Cuando se ejecuta el código anterior con las entradas que mostramos, el valor de `c3` se establece al `'\n'`, es decir, el carácter de salto de línea. `c3` no se establece al valor `'C'`.

detección del final de un archivo de entrada

Una cosa que podemos hacer con la función miembro `get` es hacer que el programa detecte el final de una línea. El siguiente ciclo lee una línea de entrada y se detiene después de pasar el carácter de salto de línea `'\n'`. Cualquiera de las entradas subsecuentes se leerán a partir del principio de la siguiente línea. Como primer ejemplo, simplemente hemos hecho eco de las entradas, pero la misma técnica nos permitiría hacer lo que queramos con las entradas:

```
cout "Escriba una linea de entrada y yo hare eco de ella:\n";
char simbolo;
do
```

```
{
    cin.get(simbolo);
    cout << simbolo;
} while (simbolo != '\n');
cout << "Eso es todo para esta demostracion.";
```

Este ciclo lee cualquier línea de entrada y la repite exactamente, con todo y espacios en blanco. Lo que sigue es una muestra del diálogo que este código produce:

```
Escriba una linea de entrada y yo hare eco de ella:
Do Be Do 1 2          34
Do Be Do 1 2          34
Eso es todo para esta demostracion.
```

Observe que el carácter de salto de línea `'\n'` se lee y también se escribe. Puesto que `'\n'` se envía a la salida, la cadena que comienza con las palabras `"Eso es"` comienza en una nueva línea.

La función miembro `get`

Todo flujo de entrada tiene una función miembro llamada `get` que puede servir para leer un carácter de entrada. A diferencia del operador de extracción `>>`, `get` lee el siguiente carácter de entrada, sea cual sea. En particular, `get` lee un espacio en blanco o el carácter de salto de línea `'\n'`, si cualquiera de ellos es el siguiente carácter de entrada. La función `get` toma un argumento que debe ser una variable de tipo *`char`*. Cuando se invoca `get`, se lee el siguiente carácter de entrada, y el valor de la variable argumento (que llamamos *Variable_Char* en lo que sigue) se hace igual a dicho carácter.

Sintaxis

Flujo_de_Entrada`.get(`*Variable_Char*`);`

Ejemplo

```
char siguiente_simbolo;
cin.get(siguiente_simbolo);
```

Si queremos usar `get` para leer de un archivo, empleamos un flujo de archivo de entrada en lugar del flujo `cin`. Por ejemplo, si `flujo_in` es un flujo de entrada conectado a un archivo, lo que sigue leerá un carácter del archivo de entrada y lo colocará en la variable `char siguiente_simbolo`:

```
flujo_in.get(siguiente_simbolo);
```

Antes de poder usar `get` con un flujo de archivo de entrada como `flujo_in`, el programa deberá conectar el flujo al archivo de entrada con una llamada a `open`.

`Nombre_de_Flujo.put`

La función miembro `put` es análoga a la función miembro `get` sólo que se emplea para salida en lugar de para entrada. Mediante `put` un programa puede enviar a la salida un carácter. La función miembro `put` recibe un argumento que debe ser una expresión de tipo *`char`*, digamos una constante o una variable de tipo *`char`*. El valor del argumento se envía

'\n' y "\n"

Podría parecernos que '\n' y "\n" son la misma cosa. En una instrucción cout producen el mismo efecto, pero no podemos usarlos indistintamente en todas las situaciones. '\n' es un valor de tipo *char* y se puede guardar en una variable de tipo *char*. En cambio, "\n" es una cadena que incluye un solo carácter. Por tanto, "\n" no es de tipo *char* y no se puede guardar en una variable de tipo *char*.

al flujo de salida cuando se invoca la función. Por ejemplo, lo que sigue envía la letra 'a' a la pantalla:

```
cout.put('a');
```

La función cout.put no nos permite hacer nada que no podamos hacer con los métodos que vimos anteriormente, pero la incluimos para que nuestra explicación sea completa.

Si nuestro programa usa cin.get y/o cout.put, entonces, como siempre que usamos cin y cout, el programa debe incluir la siguiente directiva:

```
#include <iostream>
```

Así mismo, si nuestro programa usa get con un flujo de archivo de entrada o put con un flujo de archivo de salida, entonces, como siempre que realizamos E/S con archivos, el programa debe contener la siguiente directiva:

```
#include <fstream>
```

La función miembro put

Todo flujo de salida tiene una función miembro llamada put. La función miembro put recibe un argumento que debe ser una expresión de tipo *char*. Cuando se invoca put, el valor de su argumento (que llamamos *Expresion_Char* en lo que se muestra a continuación) se envía al flujo de salida.

Sintaxis

Flujo_de_Salida.put(*Expresion_Char*);

Ejemplos

```
cout.put(siguiente_simbolo);
cout.put('a');
```

Si queremos usar put para enviar salidas a un archivo, usamos un flujo de archivo de salida en lugar del flujo cout. Por ejemplo, si flujo_out es un flujo de salida conectado a un archivo, lo que sigue enviará el carácter 'Z' al archivo conectado a flujo_out:

```
flujo_out.put('Z');
```

Antes de poder usar put con un flujo de archivo de salida, como flujo_out, el programa debe conectar el flujo al archivo de salida con una llamada a la función miembro open.

Al utilizar cualquiera de estas directivas `include`, su programa también deberá contener lo siguiente:

```
using namespace std;
```

La función miembro `putback` (*Opcional*)

Hay ocasiones en que nuestro programa necesita conocer el siguiente carácter del flujo de entrada. Sin embargo, después de leer el siguiente carácter podríamos decidir que no queremos procesar ese carácter y nos gustaría volverlo a colocar en el flujo de entrada. Por ejemplo, si queremos que nuestro programa lea hasta encontrar el siguiente espacio en blanco de un flujo de entrada, *pero sin incluir dicho espacio*, el programa tendrá que leer ese primer espacio para saber cuándo debe dejar de leer, pero entonces el espacio ya no estará en el flujo de entrada. Alguna otra parte del programa podría necesitar leer y procesar ese espacio en blanco. Hay varias formas de manejar este tipo de situaciones, pero la más fácil es usar la función miembro `putback`. Ésta es miembro de todos los flujos de entrada; recibe un argumento de tipo *`char`* y coloca el valor de ese argumento de vuelta en el flujo de entrada. El argumento puede ser cualquier expresión que al evaluarse dé un valor de tipo *`char`*.

Por ejemplo, lo que sigue lee caracteres del archivo conectado al flujo de entrada `fin` y los escribe en el archivo conectado al archivo de salida `fout`. El código lee caracteres hasta el primer espacio en blanco que encuentra, pero sin incluir dicho espacio.

```
fin.get(siguiente);
while (siguiente != ' ')
{
    fout.put(siguiente);
    fin.get(siguiente);
}
fin.putback(siguiente);
```

Observe que después de que se ejecuta este código, el espacio en blanco que se leyó sigue estando en el flujo de entrada `fin`, porque el código lo vuelve a meter ahí después de leerlo.

Cabe señalar que `putback` coloca un carácter en un flujo de *entrada*, mientras que `put` coloca un carácter en un flujo de *salida*.

El carácter que se vuelve a colocar en el flujo de entrada con la función miembro `putback` no tiene que ser el último carácter que se leyó; puede ser cualquier otro. Si metemos un carácter distinto del último que se leyó, `putback` no modificará el texto del archivo de entrada, aunque el programa se comportará como si así hubiera sido.

EJEMPLO DE PROGRAMACIÓN
Verificación de entradas

`obtener_entero`

Si un usuario introduce entradas incorrectas, la ejecución del programa podría ser inútil. Para asegurarnos de que nuestro programa no procese entradas indebidas, es recomendable usar funciones de entrada que permitan al usuario volver a introducir las entradas hasta que éstas sean correctas. La función `obtener_entero` del cuadro 5.7 pregunta al usuario si la entrada es correcta y pide un nuevo valor si el usuario dice que no lo es. El programa del cuadro 5.7 es sólo un controlador para probar la función `obtener_entero`, pero la función, o una muy parecida, se puede usar en casi cualquier tipo de programa que obtenga sus entradas del teclado.

CUADRO 5.7 *Verificación de entradas (parte 1 de 2)*

```
//Programa para demostrar las funciones nueva_linea y obtener_entero.
#include <iostream>
using namespace std;

void nueva_linea();
//Desecha todo lo que queda en la línea de entrada actual.
//También desecha el '\n' al final de la línea.
//Esta versión sólo funciona para entradas desde el teclado.

void obtener_entero(int& numero);
//Postcondición: La variable numero ha recibido un valor
//aprobado por el usuario.

int main()
{
    int n;

    obtener_entero(n);
    cout << "Valor definitivo leido = " << n << endl
         << "Fin de la demostracion.\n";
    return 0;
}


//Usa iostream:
void nueva_linea()
{
    char simbolo;
    do
    {
         cin.get(simbolo);
    } while (simbolo != '\n');
}
```

CUADRO 5.7 *Verificación de entradas (parte 2 de 2)*

```
//Usa iostream:
void obtener_entero(int& numero)
{
    char resp;
    do
    {
            cout << "Escribe el numero de entrada: ";
            cin >> numero;
            cout << "Escribiste " << numero
                 << " Es correcto? (si/no): ";
            cin >> resp;
            nueva_linea();
    } while ((resp != 'S') && (resp != 's'));
}
```

Diálogo de ejemplo

```
Escribe el numero de entrada: 57
Escribiste 57 Es correcto? (si/no): no
Escribe el número de entrada: 75
Escribiste 75 Es correcto? (si/no): si
Valor definitivo leido = 75
Fin de la demostracion.
```

`nueva_linea ()`

Observe la llamada a la función `nueva_linea()`. Esta función lee todos los caracteres del resto de la línea en curso pero no hace nada con ellos. Esto equivale a desechar el resto de la línea. Así pues, si el usuario teclea **No**, el programa leerá la primera letra, que es **N**, y luego invocará la función `nueva_linea`, que desecha el resto de la línea de entrada. Esto implica que si el usuario teclea **75** en la siguiente línea de entrada, como en el diálogo de ejemplo, el programa leerá el número **75** y no intentará leer la letra **o** de la palabra **No**. Si el programa no incluyera una llamada a la función `nueva_linea`, el siguiente elemento leído sería la **o** de la línea que contiene **No**, en lugar del número **75** de la siguiente línea.

Si tienes duda, vuelve a introducir las entradas

Observe la expresión booleana que termina el ciclo *`do-while`* en la función `obtener_entero`. Si la entrada no es correcta, se supone que el usuario tecleará **No** (o alguna variante como **no**) y esto causará una iteración más del ciclo. Sin embargo, en lugar de verificar si el usuario teclea una palabra que comienza con `'N'`, el ciclo *`do-while`* verifica si la primera letra de la respuesta del usuario *no* es igual a `'S'` (y tampoco es igual a `'s'`). En tanto el usuario no se equivoque y responda con alguna forma de **Sí** o **No**, pero nunca con

otra cosa, verificar si se respondió **No** o si se respondió **Sí** es la misma cosa. Sin embargo, dado que el usuario podría responder de alguna otra forma, es más seguro verificar que la respuesta no fue **Sí**. Para entender por qué esto es más seguro, supongamos que el usuario se equivoca al introducir el número de entrada. La computadora hará eco del número y preguntará si es correcto. El usuario deberá teclear **No**, pero supongamos que otra vez se equivoca y teclea **Bo**, lo cual no es descabellado porque la B está junto a la N en el teclado. Puesto que `'B'` no es igual a `'S'`, se ejecutará el cuerpo del ciclo *`do-while`*, y el usuario tendrá oportunidad de volver a introducir el número.

Pero, ¿qué sucede si la respuesta correcta es **Sí** y el usuario por error teclea algo que no comienza con `'S'` ni con `'s'`? En ese caso el ciclo no debe repetirse, pero lo hará. Esto no es lo correcto, pero no es tan grave como haberse equivocado al teclear la entrada y perder la oportunidad de corregirla. Lo único que pasa es que el usuario tiene que volver a teclear la entrada innecesariamente, pero no se echa a perder toda la ejecución del programa. Al verificar entradas, es mejor arriesgarse a que el ciclo se repita una vez de más y no arriesgarse a continuar con entradas incorrectas.

RIESGO `'\n'` *inesperado en las entradas*

Al usar la función miembro `get` hay que tener en cuenta todos los caracteres de entrada, incluso los que no consideramos símbolos, como los espacios en blanco y el carácter de salto de línea `'\n'`. Un problema común al usar `get` es olvidarnos de procesar el `'\n'` con que termina cada línea de la entrada. Si hay un carácter de salto de línea en el flujo de entrada y no se lee y se desecha, como casi siempre se hace, entonces después, cuando el programa espere leer un símbolo "real" empleando la función miembro `get`, leerá el carácter `'\n'`. Si queremos eliminar semejante `'\n'` sobrante del flujo de entrada podemos usar la función `nueva_linea` que definimos en el cuadro 5.7. Examinemos un ejemplo concreto.

Está permitido combinar las diferentes formas de `cin`. Por ejemplo, lo siguiente es válido:

```
cout << "Escribe un numero:\n";
int numero;
cin >> numero;
cout << "Ahora escribe una letra:\n";
char simbolo;
cin.get(simbolo);
```

Sin embargo, esto puede causar problemas, como ilustra el siguiente diálogo:

```
Escribe un numero:
21
Ahora escribe una letra:
A
```

Con este diálogo, el valor de `numero` será `21`, como esperamos, pero si creemos que el valor de la variable `simbolo` es `'A'` sufriremos una decepción. El valor que se coloca en `simbolo` es `'\n'`. Después de leer el número `21`, el siguiente carácter del flujo de entrada es el de salto de línea, `'\n'`, y ése es el que se lee a continuación. Recuerde que `get` no pasa por alto los saltos de línea ni los espacios. (De hecho, dependiendo de lo que siga en el programa, tal vez ni tengamos oportunidad de escribir la `A`. Una vez que la variable `simbolo` se llena con el carácter `'\n'`, el programa continúa con la siguiente instrucción del programa. Si esa instrucción envía salidas a la pantalla, dichas salidas aparecerán en la pantalla antes de que tengamos tiempo de escribir la `A`.)

Cualquiera de las siguientes versiones del código anterior hará que el diálogo de muestra llene la variable `numero` con `21` y la variable `simbolo` con `'A'`:

```
cout << "Escribe un numero:\n";
int numero;
cin >> numero;
cout << "Ahora escribe una letra:\n";
char simbolo;
cin >> simbolo;
```

Como alternativa, podemos usar la función `nueva_linea` que definimos en el cuadro 5.7, así:

```
cout << "Escribe un numero:\n";
int numero;
cin >> numero;
nueva_linea();
cout << "Ahora escribe una letra:\n";
char simbolo;
cin.get(simbolo);
```

Como vemos en esta segunda versión, es posible combinar las dos formas de `cin` y que el programa funcione correctamente, pero se requiere cuidado extra.

Ejercicios de AUTOEVALUACIÓN

22. Supongamos que `c` es una variable de tipo *`char`*. ¿Cuál es la diferencia entre las siguientes instrucciones?

    ```
    cin >> c;
    ```

    y

    ```
    cin.get (c);
    ```

23. Supongamos que `c` es una variable de tipo *`char`*. ¿Cuál es la diferencia entre las siguientes instrucciones?

    ```
    cout << c;
    ```

    y

    ```
    cout.put (c);
    ```

24. (Esta pregunta es para aquellos que han leído la sección opcional sobre la función miembro `putback`.) La función miembro `putback` "vuelve a poner" un símbolo en el flujo de entrada, el símbolo que se pone nuevamente debe ser el último símbolo del flujo. Por ejemplo, si su programa lee una `'a'` desde el flujo de entrada ¿puede utilizar la función putback para volver a poner una `'b'`, o sólo puede volver a poner una `'a'`?

25. Considere el siguiente código (y suponga que se incorpora en un programa completo y correcto que después se ejecuta):

```
char c1, c2, c3, c4;
cout << "Escriba una linea de entrada:\n";
cin.get (c1);
cin.get (c2);
cin.get (c3);
cin.get (c4);
cout << c1 << c2 << c3 << c4 << "FIN DE LA SALIDA";
```

Si el diálogo comienza como sigue, ¿cuál será la siguiente línea de salida?

```
Escriba una linea de entrada:
a b c d e f g
```

26. Considere el siguiente código (y suponga que se incorpora en un programa completo y correcto que después se ejecuta):

```
char siguiente;
int conteo = 0;
cout << "Escriba una linea de entrada:\n";
cin.get(siguiente);
while (siguiente != '\n')
{
    if ((conteo%2) == 0)
       cout << siguiente;
    conteo++;
    cin.get(siguiente);
}
```

Verdadero si `conteo` *es par.*

Si el diálogo comienza como sigue, ¿cuál será la siguiente línea de salida?

```
Escriba una linea de entrada:
abcdef gh
```

27. Suponga que se ejecuta el programa descrito en el ejercicio 26 y que el diálogo comienza como sigue (en lugar de comenzar como se muestra en el ejercicio 26). ¿Cuál será la siguiente línea de salida?

```
Escriba una linea de entrada:
0 1 2 3 4 5 6 7 8 9 10 11
```

28. Considere el siguiente código (y suponga que se incorpora en un programa completo y correcto que entonces se ejecuta):

```
char siguiente;
int conteo = 0;
cout << "Escriba una linea de entrada:\n";
cin >> siguiente;
while (siguiente != '\n')
{
    if ((conteo%2) == 0)
       cout << siguiente;
    conteo++;
    cin  >> siguiente;
}
```

Si el diálogo comienza como sigue, ¿cuál será la siguiente línea de salida?

```
Escriba una linea de entrada:
0 1 2 3 4 5 6 7 8 9 10 11
```

La función miembro `eof`

Todo flujo de archivo de entrada tiene una función miembro, llamada `eof`, que puede servir para determinar si ya se leyó todo el archivo y no hay más entradas para el programa. Ésta es la segunda técnica que hemos presentado para determinar si un programa ya leyó todo el contenido de un archivo.

La palabra `eof` es una abreviatura de *end of file* (fin de archivo). La función `eof` no recibe argumentos, de modo que si el flujo de entrada se llama `fin`, una llamada a la función `eof` se escribe así:

```
fin.eof()
```

Ésta es una expresión booleana que se puede utilizar para controlar un ciclo *`while`*, un ciclo *`do_while`* o una instrucción *`if-else`*. Esta expresión se satisface (es decir, es *`true`*) si el programa ya leyó más allá del final del archivo de entrada; de lo contrario, la expresión no se satisface, es decir, es *`false`*.

`eof` suele usarse con "not"

Debido a que generalmente queremos verificar que *no* hemos llegado al final del archivo, las llamadas a la función miembro `eof` suelen ir precedidas con un *not*. Recuerde que en C++ se emplea el símbolo `!` que expresa *not*. Por ejemplo, considere la siguiente instrucción:

```
if (! fin.eof())
    cout << "No hemos terminado.";
else
    cout << "Fin del archivo.";
```

La expresión booleana que sigue al *`if`* significa "no en el final del archivo conectado a `fin`". Así pues, la instrucción *`if-else`* anterior desplegará lo siguiente en la pantalla:

```
No hemos terminado.
```

si el programa todavía no ha leído más allá del final del archivo que está conectado al flujo `fin`. La instrucción *`if-else`* desplegará lo que sigue si el programa ya leyó más allá del final del archivo:

```
Fin del archivo.
```

terminación de un ciclo de entrada mediante la función `eof`

Como ejemplo adicional del uso de la función miembro `eof`, supongamos que el flujo de entrada `flujo_in` se ha conectado a un archivo de entrada con una llamada a `open`. Entonces podremos escribir todo el contenido del archivo en la pantalla con el siguiente ciclo *`while`*:

```
flujo_in.get(siguiente);
while (! flujo_in.eof())
{
    cout << siguiente;
    flujo_in.get(siguiente);
}
```

Si lo prefiere, puede usar `cout.put(siguiente)` *aquí.*

El ciclo *`while`* anterior lee cada uno de los caracteres del archivo de entrada y los va colocando uno por uno en la variable *`char`* `siguiente` usando la función miembro `get`, y luego escribe el carácter en la pantalla. Una vez que el programa haya pasado el final del archivo, el valor de `flujo_in.eof()` cambiará de `false` a `true`. Por lo tanto,

```
(! flujo_in.eof())
```

cambiará de `true` a `false` y el ciclo terminará.

Cabe señalar que `flujo_in.eof()` no se vuelve `true` sino hasta que el programa intenta leer un carácter más allá del final del archivo. Por ejemplo, supongamos que el archivo contiene lo siguiente (sin salto de línea después de la `c`):

```
ab
c
```

En realidad, ésta es la siguiente lista de cuatro caracteres:

```
ab<el caracter de nueva linea '\n'>c
```

El ciclo anterior leerá una `'a'` y la escribirá en pantalla, luego leerá una `'b'` y la escribirá en la pantalla, luego leerá el carácter de salto de línea `'\n'` y lo escribirá en la pantalla, luego leerá una `'c'` y la escribirá en la pantalla. En ese punto el ciclo habrá leído todos los caracteres del archivo. Sin embargo, `flujo_in.eof()` todavía será `false`. El valor de `flujo_in.eof()` no cambiará de `false` a `true` hasta que el programa trate de leer un carácter más. Es por ello que el ciclo *`while`* anterior termina con `flujo_in.get(siguiente)`. El ciclo necesita leer un carácter extra para poder terminar.

Existe un marcador especial al final de un archivo. La función miembro `eof` no cambia de `false` a `true` en tanto no se lea este marcador de fin de archivo. Es por ello que el ciclo *`while`* anterior puede leer un carácter más allá de lo que para nosotros es el último carácter del archivo. Sin embargo, este marcador de fin de archivo no es un carácter ordinario y no debemos manipularlo como si lo fuera. Podemos leerlo, pero no debemos escri-

birlo después. Si lo hacemos el resultado es impredecible. El sistema coloca automáticamente este marcador al final de cada uno de nuestros archivos.

El siguiente ejemplo de programación usa la función miembro `eof` para determinar si el programa ya leyó todo el archivo de entrada.

cómo decidir la forma de probar si se llegó al final de un archivo de entrada

Ahora tenemos dos métodos para detectar el final de un archivo. Podemos usar la función miembro `eof` o el método que describimos en el Tip de programación "Verifique el fin de un archivo". En la mayoría de las situaciones podemos usar cualquiera de los dos métodos, pero muchos programadores usan los dos métodos en diferentes situaciones. Si usted no tiene alguna otra razón para preferir uno de estos dos métodos, puede seguir esta regla general: emplee la función miembro `eof` cuando esté tratando las entradas como texto y leyéndolas con la función miembro `get`; use el otro método si está procesando datos numéricos.

Ejercicios de AUTOEVALUACIÓN

29. Suponga que `ins` es un archivo de flujo de entrada que ha sido conectado a un archivo con la función miembro `open`.
    Supongamos que el programa acaba de leer el último carácter en el archivo. En este punto ¿`ins.eof()` evaluaría a *`true`* o *`false`*?

30. Escriba la definición de una función *`void`* llamada `texto_a_pantalla` que tenga un parámetro formal llamado `flujo_archivo` de tipo `ifstream`. La precondición y la postcondición de la función son:

```
//Precondición: El flujo flujo_archivo se ha conectado a un
//archivo con una llamada a la función miembro open.
//Postcondición: El contenido del archivo conectado a
//flujo_archivo se ha copiado en la pantalla carácter por
//carácter de modo que la salida a la pantalla es igual al
//contenido del texto del archivo.
//(Esta función no cierra el archivo.)
```

EJEMPLO DE PROGRAMACIÓN
Edición de un archivo de texto

El programa que veremos aquí es un ejemplo muy sencillo de edición de textos aplicada a archivos. Lo podría utilizar una compañía de software para actualizar su material publicitario. La compañía ha estado comercializando compiladores para el lenguaje de programación C y hace poco introdujo una línea de compiladores para C++. El programa puede servir para generar automáticamente material publicitario para C++ a partir del material existente para C. El programa recibe sus entradas de un archivo que contiene un texto que dice cosas buenas acerca de C y escribe en otro archivo un texto publicitario similar pero acerca de C++. El archivo que contiene el texto publicitario de C se llama `cpub.dat`, y el nuevo archivo que recibe el texto publicitario de C++ se llama `cmaspub.dat`. Dicho programa se muestra en el cuadro 5.8.

Lo único que el programa hace es leer todos los caracteres del archivo `cpub.dat` y los copia en el archivo `cmaspub.dat`. Todos los caracteres se copian tal cual, excepto que cuando se lee la letra mayúscula `'C'` del archivo de entrada el programa escribe la cadena

"C++" en el archivo de salida. Este programa supone que siempre que la letra C ocurre en el archivo de entrada se refiere al lenguaje de programación C, por lo que este cambio es exactamente lo que se necesita para producir el texto publicitario actualizado.

Cabe señalar que los saltos de línea se conservan cuando el programa lee caracteres del archivo de entrada y escribe los caracteres en el archivo de salida. El carácter de nueva línea `'\n'` se trata igual que cualquier otro carácter: se lee del archivo de entrada con la función miembro `get` y se escribe en el archivo de salida usando el operador de inserción `<<`. Debemos utilizar la función miembro `get` para leer las entradas. Si en vez de ello usamos el operador de extracción `>>` para leer las entradas, el programa pasará por alto todo el espacioblanco, lo que quiere decir que no se leerán del archivo de entrada ni los espacios en blanco ni los caracteres de nueva línea `'\n'`, y por tanto no se copiarán en el archivo de salida.

Observe también el uso de la función miembro `eof` para detectar el fin del archivo de entrada y terminar el ciclo *while*.

Funciones de caracteres predefinidas

En el procesamiento de textos es común querer convertir letras minúsculas en mayúsculas o viceversa. La función predefinida `toupper` sirve para convertir una letra minúscula en una mayúscula. Por ejemplo, `toupper('a')` devuelve `'A'`. Si el argumento de la función `toupper` es otra cosa que no sea una letra minúscula, `toupper` simplemente devolverá el argumento sin cambio. Por tanto, `toupper('A')` también devuelve `'A'`. La función `tolower` es similar sólo que convierte una letra mayúscula en su versión minúscula.

Las funciones `toupper` y `tolower` están en la biblioteca cuyo archivo de encabezado es `cctype`, por lo que cualquier programa que use estas funciones, o cualesquier otras funciones de dicha biblioteca, deberá contener la siguiente directiva `include`:

```
#include<cctype>
```

El cuadro 5.9 contiene descripciones de algunas de las funciones de uso más común de la biblioteca `cctype`.

La función `isspace` devuelve `true` si su argumento es un carácter de *espacioblanco*. Los caracteres de **espacioblanco** son todos los que aparecen como espacio en blanco en la pantalla, incluido el carácter de blanco, el de tabulación y el de nueva línea `'\n'`. Si el argumento de `isspace` no es un carácter de espacioblanco, la función devuelve `false`. Así pues, `isspace(' ')` devuelve `true` e `isspace('a')` devuelve `false`.

espacioblanco

Por ejemplo, lo que sigue lee una instrucción que termina con un punto y hace eco de la cadena sustituyendo todos los caracteres de espacioblanco por el símbolo `'-'`:

```
char siguiente;
do
{
    cin.get(siguiente);
    if (isspace(siguiente))
       cout << '-';
    else
       cout << siguiente;
} while (siguiente != '.');
```

Verdadero si el carácter en `siguiente` *es espacioblanco.*

CUADRO 5.8 *Edición de un archivo de texto (parte 1 de 2)*

```
//Programa para crear un archivo cmaspubl.dat idéntico al archivo
//cpub.dat sólo que cada 'C' se sustituye por "C++"
//Supone que la letra mayúscula 'C' sólo se presenta en cpub.dat como
//el nombre del lenguaje de programación C.

#include <fstream>
#include <iostream>
#include <cstdlib>
using namespace std;

void agregar_mas_mas(ifstream& flujo_in, ofstream& flujo_out);
//Precondición: flujo_in se ha conectado a un archivo de entrada con open.
//flujo_out se ha conectado a un flujo de salida con open.
//Postcondición: El contenido del archivo conectado a flujo_in se
//ha copiado en el archivo conectado a flujo_out, pero con cada 'C'
//sustituida por "C++". (Esta función no cierra los archivos.)

int main()
{
    ifstream fin;
    ofstream fout;

    cout << "Inicia edicion de archivos.\n";

    fin.open("cpub.dat");
    if (fin.fail())
    {
        cout << "No pudo abrirse el archivo de entrada.\n";
        exit(1);
    }
    fout.open("cmaspub.dat");
    if(fout.fail())
    {
        cout << "No pudo abrirse el archivo de salida.\n";
        exit(1);
    }
    agregar_mas_mas(fin, fout);
```

CUADRO 5.8 *Edición de un archivo de texto (parte 2 de 2)*

```
    fin.close();
    fout.close();

    cout << "Termina edicion de archivos.\n";
    return 0;
}

void agregar_mas_mas(ifstream& flujo_in, ofstream& flujo_out)
{
    char siguiente;

    flujo_in.get(siguiente);
    while (! flujo_in.eof())
    {
         if (siguiente == 'C')
             flujo_out << "C++";
         else
             flujo_out << siguiente;
         flujo_in.get(siguiente);
    }
}
```

cpub.dat
(El programa no lo modifica.)

```
C es uno de los lenguajes de programacion mas modernos
del mundo. No hay lenguaje tan versatil como C, y usar C
es divertido.
```

cmaspub.dat
(Después de ejecutarse el programa.)

```
C++ es uno de los lenguajes de programacion mas modernos
del mundo. No hay lenguaje tan versatil como C++, y usar
C++ es divertido.
```

Salida a la pantalla

```
Inicia edicion de archivos.
Termina edicion de archivos.
```

Por ejemplo, si el código anterior recibe las siguientes entradas:

```
Ahh  do be do.
```

producirá las siguientes salidas:

```
Ahh---do-be-do.
```

RIESGO `toupper` y `tolower` devuelven valores `int`

En muchos sentidos C++ considera a los caracteres como números enteros, similares a los números de tipo *int*. A cada carácter se le asigna un número, y cuando el carácter se almacena en una variable de tipo *char* es ese número lo que se coloca en la memoria de la computadora. En C++ podemos emplear un valor de tipo *char* como número, por ejemplo, colocándolo en una variable de tipo *int*. También podemos guardar un número de tipo *int* en una variable de tipo *char* (siempre que el número no sea demasiado grande). Así pues, el tipo *char* se puede usar como tipo para caracteres o como tipo para números enteros pequeños.

Normalmente no tenemos que ocuparnos de tales detalles, y podemos pensar en los valores de tipo *char* simplemente como caracteres sin preocuparnos por su uso como números. Sin embargo, cuando empleamos las funciones de `cctype` este detalle puede ser importante. Las funciones `toupper` y `tolower` en realidad devuelven valores de tipo *int*, no valores de tipo *char*; es decir, devuelven el número que corresponde al carácter que creemos que están devolviendo, no el carácter mismo. Por lo tanto, lo que se muestra a continuación no despliega la letra `'A'`, sino el número que se asigna a `'A'`:

```
cout << toupper ('a');
```

CUADRO 5.9 *Algunas funciones de carácter predefinidas de* `cctype` *(parte 1 de 2)*

Función	Descripción	Ejemplo

`toupper(`*Exp_Char*`)`	Devuelve la versión mayúscula de *Exp_Char*.	`char c= toupper('a');`
		`cout<< c;`
		Salidas: `A`

`tolower(`*Exp_Char*`)`	Devuelve la versión minúscula de *Exp_Char*.	`char c= tolower('A');`
		`cout<<c;`
		Salidas: `a`

`isupper(`*Exp_Char*`)`	Devuelve *true* si *Exp_Char* es una letra mayúscula; si no, devuelve *false*.	`if (isupper(c))`
		`  cout << c << " es mayúscula.";`
		`else`
		`  cout << c`
		`  << " no es mayúscula.";`

CUADRO 5.9 *Algunas funciones de carácter predefinidas de* `cctype` ***(parte 2 de 2)***

Función	Descripción	Ejemplo

`islower(`*Exp_Char*`)`	Devuelve *true* si *Exp_Char* es una letra minúscula; si no, devuelve *false*.	`char c='a';`
		`if (islower(c))`
		`  cout << c << " es minúscula.";`
		Salidas: `a es minúscula.`

`isalpha(`*Exp_Char*`)`	Devuelve *true* si *Exp_Char* es una letra del alfabeto; si no, devuelve *false*.	`char c= '$';`
		`if (isalpha(c))`
		`  cout << c << " es una letra.";`
		`else`
		`  cout << c`
		`  << " no es una letra.";`
		Salidas: `$ no es una letra.`

`isdigit(`*Exp_Char*`)`	Devuelve *true* si *Exp_Char* es uno de los dígitos `'0'` a `'9'`; si no, devuelve *false*.	`if (isdigit('3'))`
		`  cout <<"Es un dígito.";`
		`else`
		`  cout << " No es un dígito.";`
		Salidas: `Es un dígito.`

`isspace(`*Exp_Char*`)`	Devuelve *true* si *Exp_Char* es un carácter de espacio, blanco como el blanco o el símbolo de nueva línea; si no, devuelve *false*.	`//Salta una "palabra" y hace`
		`//a c igual al primer`
		`//carácter de espacioblanco`
		`//después de la "palabra".`
		`do`
		`{`
		`  cin.get(c);`
		`} while (¡isspace(c));`

Si queremos hacer que la computadora trate el valor devuelto por `toupper` o `tolower` como un valor de tipo *char* (y no uno de tipo *int*), necesitamos indicar que queremos un valor de tipo *char*. Una forma de hacerlo es colocar el valor devuelto en una variable de tipo *char*. Lo que sigue despliega el carácter `'A'`, que normalmente es lo que se quiere:

```
char c = toupper('a');
cout << c;
```

Coloca `'A'` *en la variable* `c`.

Otra forma de hacer que la computadora trate el valor devuelto por `toupper` o `tolower` como un valor de tipo *char* es usar mutación de tipo:

```
cout << static_cast<char>(toupper('a') );
```

Despliega el carácter `'A'`.

(Vimos las mutaciones de tipo en el capítulo 3 en la sección "Funciones de cambio de tipo".)

Ejercicios de AUTOEVALUACIÓN

31. Considere el siguiente código (y suponga que forma parte de un programa completo y correcto que después se ejecuta):

```
cout << "Teclea una línea de entrada:\n";
char siguiente;
do
{
   cin.get(siguiente);
   cout << siguiente;
} while ( (! isdigit(siguiente)) && (siguiente != '\n') );
cout << "<FIN DE LA SALIDA";
```

Si el diálogo comienza como sigue, ¿cuál será la siguiente línea de salida?

```
Escriba una linea de entrada:
Te vere a las 10:30 AM.
```

32. Escriba código C++ que lea una línea de texto y haga eco de la línea pero eliminando todas las letras mayúsculas.

5.4 Herencia

Una de las características más potentes de C++ es el uso de *clases derivadas*. Cuando decimos que una clase se derivó de otra queremos decir que la clase derivada se obtuvo de la otra clase añadiéndole características. Por ejemplo, la clase de flujos de *archivos* de entrada se derivó de la clase de *todos* los flujos de entrada añadiéndole funciones miembro adicionales como `open` y `close`. El flujo `cin` pertenece a la clase de *todos* los flujos de entrada, pero *no* pertenece a la clase de flujos de *archivo* de entrada porque `cin` no tiene funciones miembro llamadas `open` y `close`. Un flujo que declaramos como de tipo `ifstream` es un flujo de *archivo* de entrada porque tiene funciones miembro adicionales como `open` y `close`.

En esta sección presentaremos el concepto de clase derivada aplicado a los flujos. La palabra *herencia* se refiere al tema de las clases derivadas. Tal vez usted necesite un poco de tiempo para sentirse totalmente cómodo con la idea de clase derivada, pero es fácil aprender lo suficiente acerca de las clases derivadas como para comenzar a usarlas en algunas aplicaciones muy sencillas y útiles.

Herencia entre clases de flujos

A fin de tener clara la terminología básica, recordemos que un **objeto** es una variable que tiene funciones miembro, y que una **clase** es un tipo cuyas variables son objetos. Los flujos (como `cin`, `cout`, los flujos de archivo de entrada y los de archivo de salida) son objetos, de modo que los tipos de flujo, como `ifstream` y `ofstream`, son clases. Después de este breve repaso, consideremos algunos ejemplos de flujos y clases de flujos.

Tanto el flujo predefinido `cin` como un flujo de archivo de entrada son flujos de entrada, por lo que en cierto sentido son similares. Por ejemplo, podemos usar el operador de extracción `>>` con ambos tipos de flujos. Por otra parte, un flujo de archivo de entrada se puede conectar a un archivo empleando la función miembro `open`, pero el flujo `cin` no tiene una función miembro llamada `open`. Un flujo de archivo de entrada es una especie de flujo similar a `cin` pero diferente. Un flujo de archivo de entrada es de tipo `ifstream`. Como veremos en breve, `cin` es de tipo `istream` (sin la `'f'`). Las clases `ifsteram` e `istream` son tipos diferentes pero emparentados. La clase `ifstream` es una *clase derivada* de la clase `istream`. En esta subsección explicaremos qué significa que una clase sea una clase derivada de otra.

Considere la siguiente función, que lee dos enteros del flujo de entrada `archivo_fuente` y escribe su suma en la pantalla:

```
void sumar_dos(ifstream& archivo_fuente)
{
     int n1, n2;
     archivo_fuente >> n1 >> n2;
     cout << n1 << " + " << n2 << " = " << (n1 + n2) << endl;
}
```

Supongamos que nuestro programa contiene la definición de función anterior y la siguiente declaración de flujo:

```
ifstream fin;
```

Si `fin` se conecta a un archivo con una llamada a `open`, podremos usar la función `sumar_dos` para leer dos enteros de ese archivo y escribir su suma en la pantalla. La llamada sería la siguiente:

```
sumar_dos(fin);
```

Supongamos ahora que, más adelante en el mismo programa, queremos que el programa lea dos números del teclado y escriba su suma en la pantalla. Puesto que todos los flujos de entrada son similares, podríamos pensar que podemos usar `cin` como argumento de una segunda llamada a `sumar_dos`, así:

```
sumar_dos(cin); //NO FUNCIONARÁ
```

Como indica el comentario, esto producirá un mensaje de error cuando compilemos el programa. `cin` no es de tipo `ifstream`; `cin` es de tipo `istream` (sin la `'f'`). Si queremos usar `cin` como argumento de una función, el parámetro correspondiente de la función deberá ser de tipo `istream` (no de tipo `ifstream`). La siguiente versión de `sumar_dos` acepta `cin` como argumento:

`istream` **e** `ifstream`

```
void sumar_dos_mejor(istream& archivo_fuente)
{
    int n1, n2;
    archivo_fuente >> n1 >> n2;
    cout << n1 << " + " << n2 << " = " << (n1 + n2) << endl;
}
```

Fuera del cambio en el tipo del parámetro, esta función `sumar_dos_mejor` es idéntica a `sumar_dos`. Puesto que el parámetro de la función `sumar_dos_mejor` concuerda con

el tipo de `cin`, podemos usar la siguiente llamada de función en lugar de la anterior llamada no válida a `sumar_dos`:

```
sumar_dos_mejor(cin);
```

Ahora le tenemos una buena y tal vez sorprendente noticia: la función `sumar_dos_mejor` se puede usar con cualquier tipo de flujo de entrada, no sólo con el flujo de entrada `cin`. Lo que sigue también es válido:

```
sumar_dos_mejor(fin);
```

Esto tal vez nos sorprenda porque `fin` es de tipo `ifstream` y el argumento de `sumar_dos_mejor` debe ser de tipo `istream`. Al parecer, `fin` es de tipo `ifstream` y también de tipo `istream`. Y no es sólo apariencia, ¡así es! El flujo `fin` tiene dos tipos. ¿Cómo puede ser esto? Los tipos en cuestión tienen una relación especial. El tipo `ifstream` es una clase derivada de la clase `istream`.

clase derivada

Cuando decimos que una clase A es una **clase derivada** de alguna otra clase B, queremos decir que la clase A tiene todas las características de la clase B pero también tiene *características adicionales*. Por ejemplo, todo flujo de tipo `istream` (sin la `'f'`) se puede usar con el operador de extracción `>>`. La clase `ifstream` (con `'f'`) es una clase derivada de la clase `istream`, por lo que un objeto de tipo `ifstream` se puede usar con el operador de extracción `>>`. Sin embargo, `ifstream` tiene características adicionales, y por ello podemos hacer más con un objeto de tipo `ifstream` que con uno de tipo `istream`. Por ejemplo, una característica adicional es que un flujo de tipo `ifstream` se puede usar con la función `open`. El flujo `cin` es sólo de tipo `istream`, así que no podemos usar `cin` con la función `open`.

qué tipo usar para un parámetro de flujo

Cualquier flujo que sea del tipo `ifstream` también es del tipo `istream`, por lo que un parámetro formal de tipo `istream` se puede sustituir con un argumento de tipo `ifstream` en una llamada de función. Si estamos definiendo una función con un parámetro formal para un flujo de entrada y damos como tipo de ese parámetro `istream`, nuestra función será más versátil. Con un parámetro formal de tipo `istream`, el argumento empleado en una llamada de función puede ser un flujo de entrada conectado a un archivo, o el flujo `cin`.

Si definimos una función con un parámetro de tipo `istream`, entonces ese parámetro sólo podrá usar funciones miembro de `istream`. En particular, no podrá usar las funciones `open` y `close`. Así mismo, un parámetro de tipo `ostream` sólo podrá usar funciones miembro de `ostream`. Con parámetros de tipo `istream` y `ostream`, toda apertura de archivos se deberá efectuar antes de la llamada a la función, y el cierre deberá efectuarse después de la llamada.

La herencia puede parecer extraña al principio, pero la idea de una clase derivada en realidad es muy común. Tal vez un ejemplo de la vida cotidiana aclare la idea. La clase de todos los convertibles, por ejemplo, es una clase derivada de la clase de todos los automóviles. Todo convertible es un automóvil, pero un convertible no es sólo un automóvil. Un convertible es un tipo especial de automóvil con propiedades especiales que otros tipos de automóvil no tienen. Si tenemos un convertible, podemos abatir la capota para que el coche quede abierto. (Podríamos decir que un convertible tiene una función "`open`" como característica adicional.) De forma similar, la clase `ifstream` de flujos de archivo de entrada es una clase derivada de la clase `istream`, que consiste en *todos* los flujos de entrada. Todo flujo de archivo de entrada es un flujo de entrada, pero un flujo de archivo de entrada tiene propiedades adicionales (como la función `open`) que otros tipos de flujos de entrada (como `cin`) no tienen.

Las clases derivadas a menudo se explican usando la metáfora de la herencia y las relaciones familiares. Si la clase B es una clase derivada de la clase A, entonces decimos que la clase B es una **hija** de la clase A y la clase A es la **madre** (o **progenitora**) de la clase B. Decimos que la clase derivada **hereda** las funciones miembro de su clase madre. Por ejemplo, todo convertible hereda de la clase de todos los automóviles el hecho de que tiene cuatro ruedas, y todo flujo de archivo de entrada hereda de la clase de todos los archivos de entrada el operador de extracción >>. Es por esto que el tema de las clases derivadas se conoce como *herencia*.

hija, madre, herencia

Si todavía no está totalmente cómodo con la idea de clase derivada, lo estará una vez que comience a usarla. El recuadro "Cómo hacer versátiles los parámetros de flujo" nos dice todo lo que es indispensable saber para usar las clases derivadas que vimos en esta subsección.

Hasta ahora hemos hablado de dos clases de flujos de salida, `istream` y su clase derivada `ifstream`. La situación con los flujos de salida es similar. La clase `ostream` es la clase de todos los flujos de salida. El flujo `cout` es de tipo `ostream`. En contraste con `cout`, un flujo de archivo de salida se declara como de tipo `ofstream`. La clase `ofstream` de flujos de archivo de salida es una clase derivada de la clase `ostream`. Por ejemplo, la siguiente función escribe la palabra `"Hola"` en el flujo de salida que se da como su argumento.

`ostream` y `ofstream`

```
void saluda(ostream& cualquier_flujo_de_salida)
{
    cualquier_flujo_de_salida << "Hola";
}
```

La primera de las siguientes llamadas escribe `"Hola"` en la pantalla; la segunda escribe `"Hola"` en el archivo cuyo nombre externo es `unarch.dat`:

```
ofstream fout;
fout.open("unarch.dat");
saluda(cout);
saluda(fout);
```

Tenga presente que un flujo de archivo de salida es de tipo `ofstream` *y también* de tipo `ostream`.

Cómo hacer versátiles los parámetros de flujo

Si queremos definir una función que recibe un flujo de entrada como argumento y queremos que ese argumento sea `cin` en algunos casos y un flujo de archivo de entrada en otros, debemos usar un parámetro formal de tipo `istream` (sin la `'f'`). Sin embargo, un flujo de archivo de entrada, aunque se use como argumento de tipo `istream`, siempre debe declararse como de tipo `ifstream` (con `'f'`).

De forma similar, si queremos definir una función que tome un flujo de salida como argumento y queremos que ese argumento sea `cout` en algunos casos y un flujo de archivo de salida en otros, debemos usar un parámetro formal de tipo `ostream`. Sin embargo, un flujo de archivo de salida, aunque se use como argumento de tipo `ostream`, siempre debe declararse como de tipo `ofstream`. No podemos abrir ni cerrar un parámetro de flujo de tipo `istream` u `ostream`. Hay que abrir esos objetos antes de pasarlos a la función, y cerrarlos después de la llamada a dicha función.

EJEMPLO DE PROGRAMACIÓN
Otra función nueva_linea

Como ejemplo adicional de la forma de hacer más versátiles las funciones de flujos, consideremos la función nueva_linea del cuadro 5.7. Dicha función sólo actua con entradas del teclado, que son entradas del flujo predefinido cin. La función nueva_linea del cuadro 5.7 no tiene argumentos. A continuación hemos reescrito la función nueva_linea de modo que tenga un parámetro formal de tipo istream para el flujo de entrada:

```
//Usa iostream:
void nueva_linea(istream& flujo_in)
{
     char simbolo;
     do
     {
        flujo_in.get(simbolo);
     } while (simbolo != '\n');
}
```

Ahora supongamos que nuestro programa contiene esta nueva versión de la función nueva_linea. Si el programa recibe entradas de un flujo de entrada llamado fin (que está conectado a un archivo de entrada), lo que sigue desechará todas las entradas que quedan en la línea que se está leyendo del archivo de entrada:

```
nueva_linea(fin);
```

Por otra parte, si el programa también está leyendo entradas del teclado, lo que sigue desechará el resto de la línea de entrada que se tecleó:

```
nueva_linea(cin);
```

cómo usar ambas versiones de nueva_linea

Si nuestro programa sólo tiene la versión reescrita de nueva_linea, que toma un argumento de flujo como fin o cin, siempre hay que dar el nombre del flujo, aunque el flujo sea cin. Sin embargo, gracias a la sobrecarga, podemos tener ambas versiones de la función nueva_linea en el mismo programa: la versión sin argumentos que se da en el cuadro 5.7 y la versión con un argumento de tipo istream, que acabamos de definir. En un programa que tiene ambas definiciones de nueva_linea, las siguientes dos llamadas son equivalentes

```
nueva_linea(cin);
```

y

```
nueva_linea();
```

En realidad no necesitamos dos versiones de la función nueva_linea. La versión con un argumento de tipo istream satisface todas nuestras necesidades. No obstante, muchos programadores prefieren tener una versión sin argumentos para entradas desde el teclado, porque éstas se usan con mucha frecuencia.

Argumentos predeterminados para funciones (*Opcional*)

argumentos predeterminados

En vez de tener dos versiones de la función `nueva_linea`, una cosa que podemos hacer es usar **argumentos predeterminados**. En el código que sigue, hemos reescrito una vez más la función `nueva_linea`:

```
//Usa iostream:
void nueva_linea(istream& flujo_in = cin)
{
   char simbolo;
   do
   {
      flujo_in.get(simbolo);
   } while (simbolo != '\n');
}
```

Si invocamos esta función como

```
nueva_linea();
```

el parámetro formal tomará el argumento predeterminado `cin`. En cambio, si hacemos la llamada

```
nueva_linea(fin);
```

el parámetro formal tomará el argumento que se proporciona en la llamada, `fin`. Podemos usar este método con cualquier tipo de argumento y cualquier cantidad de argumentos.

Si se dan argumentos predeterminados para algunos parámetros pero no para otros, todos los parámetros formales que tengan argumentos predeterminados deberán aparecer juntos al final de la lista de argumentos. Si se proporcionan varios argumentos predeterminados y varios no predeterminados, la llamada debe incluir como mínimo tantos argumentos como haya argumentos no predeterminados, y como máximo tantos argumentos como parámetros haya. Los argumentos se aplicarán en orden a los parámetros que no tengan argumentos predeterminados, y luego a los parámetros que sí tengan argumentos predeterminados, hasta agotar los argumentos.

He aquí un ejemplo:

```
//Prueba del comportamiento de argumentos predeterminados
//Usa iostream:
void args_predeter(int arg1, int arg2, int arg3 = -3,
                   int arg4 = -4)
{
    cout << arg1 << ' ' << arg2 << ' ' << arg3 << ' ' << arg4
         << endl;
}
```

Podemos llamar esta función con dos, tres o cuatro argumentos:

```
args_predeter(5, 6);
```

Esta llamada proporciona los argumentos no predeterminados y usa los dos argumentos predeterminados. La salida es

```
5 6 -3 -4
```

Ahora, consideremos

```
args_predeter(6, 7, 8);
```

Esta llamada proporciona los argumentos no predeterminados y el primer argumento predeterminado; el último argumento es el valor predeterminado. Esta llamada produce la salida:

```
6 7 8 -4
```

La llamada

```
args_predeter(5, 6, 7, 8)
```

asigna todos los argumentos desde la lista de argumentos, y produce la salida:

```
5 6 7 8
```

Ejercicios de AUTOEVALUACIÓN

33. ¿Qué tipo tiene el flujo `cin`? ¿Qué tipo tiene el flujo `cout`?

34. Defina una función llamada `copiar_car` que reciba un argumento, el cual es un flujo de entrada. Al invocarse, `copiar_car` leerá un carácter del flujo de entrada que se da como argumento, y escribirá ese carácter en la pantalla. Debe ser posible invocar esta función usando `cin` o bien un flujo de archivo de entrada como argumento. (Si el argumento es un flujo de archivo de entrada, el flujo debe conectarse a un archivo antes de que se invoque la función, por lo que `copiar_car` no abrirá ni cerrará ningún archivo.) Por ejemplo, la primera de las siguientes dos llamadas a `copiar_car` copia un carácter del archivo `cosas.dat` a la pantalla, y la segunda copia un carácter del teclado a la pantalla:

```
ifstream fin;
fin.open("cosas.dat");
copiar_car(fin);
copiar_car(cin);
```

35. Defina una función llamada `copiar_linea` que reciba un argumento, el cual es un flujo de entrada. Al invocarse, `copiar_linea` leerá una línea de entrada del flujo de entrada que se da como argumento, y escribirá esa línea en la pantalla. Debe ser posible invocar esta función usando `cin` o bien un flujo de archivo de entrada como argumento. (Si el argumento es un flujo de archivo de entrada, el flujo debe conectarse un archivo antes de que se invoque la función, por lo que `copiar_linea` no abrirá ni cerrará ningún archivo.) Por ejemplo, la primera de las siguientes dos llamadas a `copiar_linea` copia una línea del archivo `cosas.dat` a la pantalla, y la segunda copia una línea del teclado a la pantalla:

```
ifstream fin;
fin.open("cosas.dat");
copiar_linea(fin);
copiar_linea(cin);
```

36. Defina una función llamada `enviar_linea` que reciba un argumento, el cual es un flujo de salida. Al invocarse, `enviar_linea` leerá una línea de entrada del teclado y la enviará al flujo de salida que se da como argumento. Debe ser posible invocar esta función usando `cout` o bien un flujo de archivo de salida como argumento. (Si el argumento es un flujo de archivo de salida, el flujo debe conectarse un archivo antes de que se invoque la función, por lo que `enviar_linea` no abrirá ni cerrará ningún archivo.) Por

ejemplo, la primera de las dos siguientes llamadas a `enviar_linea` copia una línea del teclado al archivo `mascosas.dat`, y la segunda copia una línea del teclado a la pantalla:

```
ofstream fout;
fout.open("mascosas.dat");
cout << "Teclea dos líneas de entrada:\n";
enviar_linea(fout);
enviar_linea(cout);
```

37. (Este ejercicio es para quienes hayan estudiado la sección opcional sobre argumentos predeterminados.) ¿Qué salidas produce la siguiente función en respuesta a las llamadas que se dan después?

```
void func(double x, double y = 1.1 double z = 2.3)
{
   cout << x << " " << z << endl;
}
```

Llamadas:

a) `func(2.0);`
b) `func(2.0, 3.0);`
c) `func(2.0, 3.0, 4.0);`

38. (Este ejercicio es para quienes hayan estudiado la sección opcional sobre argumentos predeterminados.) Escriba varias funciones que sobrecarguen el nombre de función `func` y con las cuales se pueda obtener el mismo efecto que con todas las llamadas con argumentos predeterminados del ejercicio anterior.

39. ¿Es falso o cierto lo que sigue? Si es falso, corríjalo, y en todo caso explíquelo detalladamente.

Una función escrita empleando un parámetro `ifstream` se puede invocar con un argumento `istream`.

Resumen del capítulo

- Es posible conectar un flujo de tipo `ifstream` a un archivo con una llamada a la función miembro `open`. Una vez hecho eso, el programa puede tomar entradas de ese archivo.
- Es posible conectar un flujo de tipo `ofstream` a un archivo mediante una llamada a la función miembro `open`. Una vez hecho eso, el programa puede enviar salidas a ese archivo.
- Es recomendable emplear la función miembro `fail` para verificar si tuvo éxito o no una llamada a `open`.
- Un **objeto** es una variable que tiene funciones asociadas. Éstas se llaman **funciones miembro**. Una **clase** es un tipo cuyas variables son objetos. Un flujo es un ejemplo de objeto. Los tipos `ifstream` y `ofstream` son ejemplos de clases.
- La que sigue es la sintaxis que usamos para escribir una llamada a una función miembro de un objeto:

Objeto_Invocador.*Nombre_de_Funcion_Miembro*(*Lista_de_Argumentos*);

El siguiente es un ejemplo con el flujo `cout` como objeto invocador y `precision` como función miembro:

```
cout.precision(2);
```

- Las funciones miembro de flujos, como `width`, `setf` y `precision`, pueden servir para formatear salidas. Estas funciones de salida operan igual con el flujo `cout`, que está conectado a la pantalla, que con flujos de salida conectados a archivos.

- Todo flujo de entrada tiene una función miembro llamada `get` que puede servir para leer un carácter de entrada. La función miembro `get` no se salta el espacioblanco. Todo flujo de salida tiene una función miembro llamada `put` que puede servir para escribir un carácter en el flujo de salida.

- La función miembro `eof` puede servir para probar si un programa ya llegó al final de un archivo de entrada o no. La función miembro `eof` opera satisfactoriamente para procesamiento de texto, y para procesar datos numéricos es preferible probar el final de un archivo empleando el otro método que vimos en el capítulo.

- Una función puede tener parámetros formales de un tipo de flujo, pero deben ser parámetros de llamada por referencia; no pueden ser parámetros de llamada por valor. Podemos emplear el tipo `ifstream` para un flujo de archivo de entrada y el tipo `ofstream` para un flujo de archivo de salida. (En el siguiente punto del resumen se mencionan otras posibilidades de tipo.)

- Si usamos `istream` (escrito sin la `'f'`) como tipo de un parámetro de flujo de entrada, el argumento que corresponde a este parámetro formal puede ser el flujo `cin` o bien un flujo de archivo de entrada de tipo `ifstream` (escrito con `'f'`). Si usamos `ostream` (sin `'f'`) como tipo de un parámetro de flujo de salida, el argumento que corresponde a ese parámetro formal puede ser el flujo `cout` o un flujo de archivo de salida de tipo `ofstream` (con `'f'`).

- Los parámetros de funciones pueden tener argumentos predeterminados que suministran valores para los parámetros si se omite el argumento correspondiente en la llamada. Estos argumentos deben ir después de cualesquier parámetros para los que no se den argumentos predeterminados. Las llamadas a tales funciones deben proporcionar primero argumentos para los parámetros que no tienen argumentos predeterminados. Los argumentos subsecuentes se usan en lugar de los argumentos predeterminados, hasta el número de parámetros que la función tiene.

Respuestas a los ejercicios de autoevaluación

1. Los flujos `fin` y `fout` se declaran como sigue:

```
ifstream fin;
ofstream fout;
```

La directiva `include` que va al principio del archivo es:

```
#include <fstream>
```

Su código también necesita

```
using namespace std;
```

2.
```
fin.open ("stuff1.dat");
if (fin.fail())
{
    cout <<"Error al abrir el archivo de entrada.\n";
    exit(1);
}
fout.open("stuff2.dat");
if (fout.fail())
{
    cout <<"Error al abrir el archivo de salida.\n";
    exit(1);
}
```

3.
```
fin.close();
fout.close();
```

4. Es necesario sustituir el flujo `flujo_out` por el flujo `cout`. Cabe señalar que no es necesario declarar `cout`, no es necesario llamar a `open` con `cout`, y no es necesario cerrar `cout`.

5. `#include <stdlib>`

   Su código también necesita

   ```
   using namespace std;
   ```

6. La función `exit(1)` devuelve el argumento al sistema operativo. Por convención, el sistema operativo usa un `1` como indicación de estado de error y `0` como indicación de éxito. Lo que se haga realmente dependerá del sistema.

7. `bla.dobedo(7);`

8. Tanto los archivos como las variables de programa almacenan valores y podemos recuperar valores de ellos. Las variables de programa sólo existen mientras se ejecuta el programa, en tanto que los archivos pueden existir antes de que se ejecute un programa, y seguir existiendo después de que el programa termina. En síntesis, los archivos pueden ser permanentes, las variables no. Los archivos permiten almacenar grandes cantidades de datos, mientras que las variables de programa no ofrecen una capacidad tan grande.

9. Hasta ahora hemos visto las funciones miembro `open`, `close` y `fail`. A continuación ilustramos su uso.

   ```
   int c;
   ifstream in;
   ofstream out;
   in.open("in.dat");
   if (in.fail())
   {
      cout << "Error al abrir el archivo.\n";
      exit(1);
   }
   in >> c;

      out.open("out.dat");
      if (out.fail())
   ```

```
{
   cout << "Error al abrir el archivo de salida.\n";
   exit(1);
}
out << c
out.close();
in.close();
```

10. Este es el "volver a empezar" que se describe al principio del capítulo. Es necesario cerrar el archivo y volverlo a abrir. Esta acción coloca la posición de lectura al principio del archivo, lista para leerlo otra vez.

11. Los dos nombres son *external file name* y *stream name*. El nombre de archivo externo es el que se utiliza en el sistema operativo. Es el verdadero nombre del archivo, pero se utiliza sólo en la llamada a la función `open`, la cual conecta el archivo a un flujo. El nombre flujo es una variable de flujo (normalmente de tipo `ifstream` u `ofstream`). Después de la llamada `open`, su programa siempre utiliza el nombre de flujo como el nombre del archivo.

12.
```
* 123*123*
* 123*123*
```

Cada uno de los espacios contiene exactamente dos caracteres de blanco. Observe que una llamada a `width` o a `setw` sólo afecta el envío a la salida de un elemento.

13.
```
*  123*123  *  123*
```

Cada espacio contiene exactamente dos caracteres de blanco.

14.
```
*  123*123*
* +123*+123*
*123  *123  *
```

Sólo hay un espacio entre el * y el + en la segunda línea. Los demás espacios contienen exactamente dos caracteres en blanco cada uno.

15. La salida al archivo `cosas.dat` será exactamente la misma que se dio en la respuesta al Ejercicio 14.

16.
```
*12345*
```

Observe que se envía a la salida el entero completo aunque ello requiere más espacio que el que especifica `setw`.

17. a) `ios::fixed` Establecer esta bandera hace que los números de punto flotante no se desplieguen en notación `e`, es decir, no en "notación científica". Al establecer esta bandera, se desactiva `ios::scientific`.

    b) `ios::scientific` Establecer esta bandera hace que los números de punto tablecer esta bandera se desactiva `ios::fixed`.

c) `ios::showpoint` Establecer esta bandera hace que siempre se desplieguen el punto decimal y los ceros a la derecha.

d) `ios::showpos` Establecer esta bandera hace que se despliegue un signo de más antes de los valores positivos enteros.

e) `ios::right` Establecer esta bandera hace que la siguiente salida se coloque en el extremo derecho de cualquier campo que se establezca con la función miembro `width`. Es decir, si sobran espacios en blanco, se colocan antes de la salida. Al establecerse esta bandera se desactiva `ios::left`.

f) `ios::left` Establecer esta bandera hace que la siguiente salida se coloque en el extremo izquierdo de cualquier campo que se establezca con la función miembro `width`. Es decir, si sobran espacios en blanco, se colocan después de la salida. Al establecerse esta bandera se desactiva `ios::right`.

18. Es preciso sutituir `flujoout` por `cout`, y eliminar las llamadas `open` y `close` para `flujoout`. No es necesario declarar `cout`, abrir `cout` ni cerrar `cout`. La directiva `#include <fstream>` incluye todos los miembros de `iostream` que se necesitan para enviar salidas a la pantalla, pero no hay problema si añadimos `#include <iostream>`, y ello podría hacer al programa más claro.

19.
```
1
2
3
3
```

20.
```
void a_pantalla(ifstream& flujo_archivo)
{
     int siguiente;
     while (flujo_archivo >> siguiente)
          cout << siguiente << endl;
}
```

21. El número máximo de caracteres que podemos teclear para guardarlos en una variable de cadena es uno menos que el tamaño declarado. Aquí el valor es 20.

22. La instrucción

```
cin >> c;
```

lee el siguiente carácter sin espacio, mientras que

```
cin.get(c);
```

lee el siguiente carácter sea o no un espacio en blanco

23. Las dos instrucciones son equivalente ambas despliegan el valor de la variable `c`.

24. El carácter que "devuelve" al flujo de entrada con la función miembro `putback` no necesariamente debe ser el último carácter que se lea. Si su programa lee `'a'` del flujo de entrada puede utilizar la función `putback` para regresar una `'b'` (el texto en el archivo de entrada no se cambiará por `putback` aun y cuando su programa se comportara como si el texto fuese cambiado).

25. El diálogo completo es

```
Escriba una linea de entrada:
a b c d e f g
a b FIN DE LA SALIDA
```

26. El diálogo completo es:

```
Escriba una linea de entrada:
abcdef gh
ace h
```

La salida no es más que cada segundo carácter de la entrada. Observe que el espacio en blanco se trata igual que cualquier otro carácter.

27. El diálogo completo es:

```
Escriba una linea de entrada:
0 1 2 3 4 5 6 7 8 9 10 11
01234567891 1
```

Observe que sólo se envía a la salida el `'1'` de la cadena de entrada `10`. La razón es que `cin.get` está leyendo caracteres, no números, y por ello lee la entrada `10` como los dos caracteres `'1'` y `'0'`. Puesto que el código se escribió de modo que hiciera eco sólo de cada segundo carácter, el `'0'` no se envía a la salida. En cambio, el siguiente carácter, que es un espacio en blanco, sí se envía a la salida, como se aprecia. Por lo mismo, sólo se despliega uno de los dos caracteres `'1'` de `11`. Si no le queda claro esto, escriba las entradas en una hoja de papel y use un cuadrito para indicar el carácter de blanco. Luego, tache cada segundo carácter; lo que queda es la salida que se muestra aquí.

28. Este código contiene un ciclo infinito y continuará en tanto el usuario le siga proporcionando entradas. La expresión booleana `(siguiente != '\n')` siempre es verdad porque `siguiente` se llena con la instrucción

```
cin >> siguiente
```

y esta instrucción siempre se salta el carácter de nueva línea `'\n'` (lo mismo que cualquier espacio en blanco). El código se ejecutará y, si el usuario no proporciona más entradas, el diálogo será el siguiente:

```
Escriba una linea de entrada:
0 1 2 3 4 5 6 7 8 9 10 11
0246811
```

Observe que el código del Ejercicio 27 usaba `cin.get`, y por ello leía todos los caracteres, *fueran espacios en blanco o no*, y luego enviaban a la salida cada segundo carácter. Por ello, el código del Ejercicio 27 despliega cada segundo carácter aunque ese carácter sea el espacio en blanco. En cambio, el código de este ejercicio usa `cin` y `>>`, por lo que *se salta todos los blancos* y sólo considera los caracteres que no son espacios en blanco (que en este caso son los dígitos `'0'` a `'9'`). Así pues, este código despliega cada segundo carácter *que no sea espacio blanco*. Los dos caracteres `'1'` de la salida son el primer carácter de las entradas `10` y `11`.

29. Esto se evaluará como *falso*. Su programa deberá leer un carácter más (más allá del último carácter) antes de que cambie a *verdadero*.

30.
```
void texto_a_pantalla(ifstream& flujo_archivo)
{
      char siguiente;
      flujo_archivo.get(siguiente);
      while (! flujo_archivo.eof())
      {
          cout << siguiente;
          flujo_archivo.get(siguiente);
      }
}
```

Si lo prefiere, puede utilizar `cout.put(siguiente);` en lugar de `cout << siguiente;`.

31. El diálogo completo es el siguiente:

```
Escribe una linea de entrada:
Te vere a las 10:30 AM.
Te vere a las 1<FIN DE LA SALIDA
```

32.
```
cout << "Escriba una linea de entrada:\n";
char siguiente;
do
{
   cin.get(siguiente);
   if (! isupper(siguiente))
      cout << siguiente;
} while (siguiente != '\n');
```

Observe que debe usarse `! isupper(siguiente)` y no `islower(siguiente)`. La razón es que `islower(siguiente)` es falso si `siguiente` contiene un carácter que no es una letra (digamos un espacio en blanco o una coma).

33. `cin` es de tipo `istream`; `cout` es de tipo `ostream`.

34.
```
void copiar_car(istream& archivo_fuente)
{
     char siguiente;
     archivo_fuente.get(siguiente);
     cout << siguiente;
}
```

35.
```
void copiar_linea(istream& archivo_fuente)
{
char siguiente;
do
```

```
    {
        archivo_fuente.get(siguiente);
        cout << siguiente;
    }while (siguiente != '\n');
}
```

36.
```
void enviar_linea(ostream& flujo_destino)
{
    char siguiente;
    do
    {
        cin.get(siguiente);
        flujo_destino << siguiente;
    } while (siguiente != '\n');
}
```

37. a) 2.0 1.1 2.3
    b) 2.0 3.0 2.3
    c) 2.0 3.0 4.0

38. He aquí un conjunto de funciones:

```
void func(double x)
{
    double y = 1.1;
    double z = 2.3;
    cout << x << " " << y << " " << z << endl;
}
void func(double x, double y)
{
    double z = 2.3;
    cout << x << " " << y << " " << z << endl;
}
void func(double x, double y, double z)
{
    cout << x << " " << y << " " << z << endl;
}
```

39. Falso. La situación que se plantea aquí es lo opuesto de la situación correcta. Cualquier flujo de tipo `ifstream` es también de tipo `istream`, por lo que un parámetro formal de tipo `istream` puede ser sustituido por un argumento de tipo `ifstream` en una llamada de función, y lo mismo sucede con los flujos `ostream` y `ofstream`.

Proyectos de programación

1. Escriba un programa que lea un archivo de números de tipo `int` y escriba el más grande y el más pequeño de esos números en la pantalla. El archivo contiene únicamente números de tipo `int` separados por blancos o saltos de línea. Si esto se hace como tarea, pregunte el nombre del archivo a su profesor.

2. Escriba un programa que reciba sus entradas de un archivo de números de tipo `double` y escriba en la pantalla la media de los números del archivo. Lo único que contiene el archivo es números de tipo `double` separados por blancos y/o saltos de línea. Si esto se hace como tarea, pregunte el nombre del archivo a su profesor.

3. a) Calcule la mediana de un archivo de datos. La mediana es el número tal que hay tantos datos mayores que él como menores que él. Para los fines de este programa, suponga que los datos están ordenados (digamos, de menor a mayor). La mediana es el elemento que está a la mitad del archivo si hay un número impar de elementos, o la media de los dos elementos que están a la mitad si el archivo contiene un número par de elementos. Hay que abrir el archivo, contar los miembros, cerrar el archivo y calcular dónde está el punto medio del archivo, abrir otra vez el archivo (recuerde lo que dijimos de "volver a comenzar" al principio de este capítulo), contar los datos hasta llegar al requerido y calcular la mediana.

   Si su profesor le dejó este problema de tarea, pídale un archivo de datos para probar el programa. O bien construya varios archivos, uno con un número par de datos, de menor a mayor, y otro con un número impar de datos, de menor a mayor.

   b) Si tenemos un archivo ordenado, un cuartil es uno de tres números. El primero tiene 1/4 de los datos antes de él (o sea que son menores que él); entre el primero y el segundo hay otra cuarta parte de los datos, entre el segundo y el tercero hay otra cuarta parte, y la cuarta parte final está después del tercero. Encuentre los tres cuartiles para el archivo que usó en la parte (a).

   Sugerencia: Dése cuenta de que si ya hizo la parte (a) ya completó la mitad del trabajo (ya tiene el segundo cuartil). También dése cuenta de que ya efectuó casi todo el trabajo necesario para encontrar los otros dos cuartiles.

4. Escriba un programa que reciba sus entradas de un archivo de números de tipo `double` y escriba en la pantalla la media y la desviación estándar de los números del archivo. Lo único que contiene el archivo es números de tipo `double` separados por blancos y/o saltos de línea. La desviación estándar de una lista de números n_1, n_2, n_3, etc. se define como la raíz cuadrada del promedio de los siguientes números:

   $(n_1 - a)^2$, $(n_2 - a)^2$, $(n_3 - a)^2$, etcétera.

   El número a es la media de los números n_1, n_2, n_3, etc. Si esto se hace como tarea, pregunte el nombre del archivo a su profesor.

   Sugerencia: Escriba su programa de modo que primero lea todo el archivo y calcule la media de todos los números, luego cierre el archivo, luego vuelva a abrir el archivo y calcule la desviación estándar. Le resultará útil realizar primero el Proyecto de Programación 2 y luego modificar el programa para obtener el programa de este proyecto.

5. Escriba un programa que dé y reciba asesoría acerca de la escritura de programas. El programa inicialmente escribe un consejo en la pantalla y pide al usuario que teclee un consejo diferente. Entonces, el programa termina. La siguiente persona que ejecuta el programa recibe el consejo dado por la última persona que ejecutó el programa. El consejo se guarda en un archivo y el contenido del archivo cambia después de cada ejecución del programa. Puede usar su editor para introducir el primer consejo en el archivo de modo que la primera persona que ejecute el archivo reciba algún consejo. Permita al usuario teclear un consejo de cualquier longitud, que abarque cualquier cantidad de líneas. Se debe indicar al usuario que termine su consejo oprimiendo la tecla Enter dos veces. El programa entonces podrá detectar que ha llegado al fin de las entradas si lee dos veces el carácter `'\n'` consecutivamente.

6. Escriba un programa que lea texto de un archivo y escriba una versión editada del mismo texto en otro archivo. La versión editada es idéntica a la original sólo que todas las cadenas de dos o más blancos consecutivos son sustituidas por un solo blanco. Es decir, el texto se edita eliminando todos los espacios sobrantes. El programa debe definir una función que se invoca con los archivos de entrada y de salida como argumentos. Si esto se hace como tarea, pregunte los nombres de archivo a su profesor.

7. Escriba un programa que fusione los números de dos archivos y escriba todos los números en un tercer archivo. El programa recibe sus entradas de dos archivos distintos y escribe sus salidas en un tercer archivo. Cada archivo de entrada contiene una lista de números de tipo `int` ordenados de menor a mayor. Después de ejecutarse el programa, el archivo de salida contiene todos los números de los dos archivos de entrada en una lista más grande ordenada de menor a mayor. El programa debe definir una función que se llama con los dos flujos de archivo de entrada y el flujo de archivo de salida como sus tres argumentos. Si esto se hace como tarea, pregunte los nombres de archivo a su profesor.

8. Escriba un programa para generar "correo chatarra" personalizado. El programa recibe sus entradas tanto de un archivo como del teclado. El archivo de entrada contiene el texto de una carta, excepto que el nombre del destinatario se indica con los tres caracteres #N#. El programa pide al usuario un nombre y luego escribe la carta en un segundo archivo pero sustituyendo los tres caracteres #N# por el nombre. La cadena de tres caracteres #N# debe ocurrir exactamente una vez en la carta.

   Sugerencia: Haga que su programa lea del archivo de entrada hasta encontrar los tres caracteres #N#, y que copie lo que lee en el archivo de salida. Cuando el programa encuentra los tres caracteres #N#, envía salidas a la pantalla que solicitan teclear un nombre. El resto de los detalles no deberá ser difícil de imaginar. El programa debe definir una función que se llama con los flujos de archivos de entrada y de salida como argumentos. Si esto se hace como tarea, pregunte los nombres de archivo a su profesor.

   Versión más difícil (usando material de la sección "Nombres de archivo como entrada (opcional)"): Permita que la cadena #N# ocurra cualquier número de veces en el archivo. En este caso el nombre se almacena en dos variables de cadena. Para esta versión suponga que hay un nombre de pila y un apellido únicamente.

9. Escriba un programa que calcule calificaciones numéricas para un curso. Los expedientes del curso están en un archivo que sirve como archivo de entrada, el cual tiene exactamente este formato: cada línea contiene el apellido de un estudiante, luego un espacio, luego el nombre de pila del estudiante, luego un espacio y luego diez puntajes de examen en una línea. Los puntajes son números enteros separados por un espacio. El programa toma sus entradas de este archivo y envía sus salidas a otro archivo. Los datos del archivo de salida son exactamente los mismos que los del archivo de entrada sólo que hay un número adicional (de tipo `double`) al final de cada línea. Este número es la media de los diez puntajes de examen del estudiante. Si esto se hace como tarea, pregunte los nombres de archivo a su profesor. Use al menos una función que tenga flujos de archivo como argumentos.

10. Mejore el programa que escribió para el Proyecto de Programación 9 de las siguientes maneras.

    a) La lista de puntajes en cada línea puede contener diez o menos puntajes. (Si hay menos de diez puntajes quiere decir que el estudiante no presentó uno o más exámenes.) El puntaje medio sigue siendo la suma de los puntajes de examen dividida entre 10. Esto equivale a asignar al estudiante un puntaje de 0 por cualquier examen no presentado.

    b) El archivo de salida contendrá al principio una línea (o líneas) que explican las salidas. Use instrucciones de formateo para que la explicación tenga buena presentación y sea fácil de leer.

    c) Después de colocar las salidas deseadas en un archivo de salida, el programa cerrará todos los archivos y luego copiará el contenido del archivo de "salida" en el archivo de "entrada" de modo que el efecto neto sea modificar el contenido del archivo de entrada.

    Use al menos dos funciones que tengan flujos de archivo como argumentos. Si esto se hace como tarea, pregunte los nombres de archivo a su profesor.

11. Escriba un programa que calcule la longitud media de palabra (número promedio de caracteres por palabra) de un archivo que contiene algo de texto. Una palabra se define como cualquier cadena de símbolos precedida y seguida por una de las siguientes cosas en cada extremo: un blanco, una coma, un punto, el principio de una línea o el final de una línea. El programa deberá definir una función que se invoque con el flujo de archivo de entrada como argumento. Esta función también deberá funcionar con el flujo `cin` como flujo de entrada, aunque la función no se invocará con `cin` como argumento en este programa. Si esto se hace como tarea, pregunte el nombre del archivo a su profesor.

12. Escriba un programa para corregir un programa C++ que tenga errores relacionado con el uso de los operadores << y >> con `cin` y `cout`. El programa sustituye cada ocurrencia (incorrecta) de

    `cin <<`

    por la versión corregida

    `cin >>`

    y cada ocurrencia (incorrecta) de

    `cout >>`

    por la versión corregida

    `cout <<`

    Si quiere que el programa sea más fácil, suponga que siempre hay exactamente un espacio en blanco entre cualquier ocurrencia de `cin` y un << subsecuente, y también que hay exactamente un espacio en blanco entre cada ocurrencia de `cout` y un >> subsecuente.

    Si prefiere una versión más difícil, contemple la posibilidad de que haya cualquier número de espacios, o incluso ninguno, entre `cin` y << y entre `cout` y >>; en este caso más difícil, la versión corregida sólo tiene un espacio entre `cin` o `cout` y el operador que sigue. El programa a corregir está en un archivo y la versión corregida se envía a un segundo archivo. El programa debe definir una función que se invoque con los flujos de archivo de entrada y de salida como argumentos.

    Si esto se hace como tarea, pregunte los nombres de archivo a su profesor y pregúntele si debe hacer la versión fácil o la difícil.

    Sugerencia: Aunque haga la versión difícil, probablemente le resultará más fácil hacer primero la versión fácil y luego modificar el programa para que realice la tarea difícil.

13. Escriba un programa que permita al usuario teclear cualquier pregunta de una línea y que conteste la pregunta. El programa no se fijará realmente en la pregunta; se limitará a leerla y a desechar todo lo leído. La respuesta siempre será una de las siguientes:

```
No estoy seguro pero creo que encontraras la respuesta en el Capitulo #N.
Me parece una pregunta muy buena.
En su lugar, yo no me preocuparia por tales cosas.
Esa pregunta ha intrigado a los filosofos durante siglos.
Yo que se. Solo soy una maquina.
Meditalo y encontraras la respuesta.
Sabia la respuesta a esa pregunta, pero se me olvido.
La respuesta se puede encontrar en un sitio secreto en el bosque.
```

Estas respuestas se almacenan en un archivo (una respuesta por línea) y el programa simplemente lee la siguiente respuesta del archivo y la despliega para contestar a la pregunta. Una vez que el programa ha leído todo el archivo simplemente lo cierra, lo vuelve a abrir y vuelve a comenzar con la lista de respuestas.

Siempre que el programa despliega la primera respuesta, sustituye los dos símbolos #N por un número entre 1 y 14 inclusive. Para escoger el número entre 1 y 14, el programa deberá inicializar una variable con 14 y decrementar su valor en uno cada vez que despliega el número, de modo que los números de capítulo vayan del 14 al 1. Cuando la variable llegue al valor 0, el programa deberá asignarle otra vez el valor 14. Dé al número 14 el nombre NUMERO_DE_CAPITULOS con una declaración de constante global con nombre, usando el modificador const.

Sugerencia: use la función nueva_linea que definimos en este capítulo.

14. Este proyecto es igual al anterior excepto que aquí el programa usa un método más sofisticado para escoger la respuesta a una pregunta. Cuando el programa lee una pregunta, cuenta el número de caracteres que tiene y almacena la cuenta en una variable llamada cuenta. Luego, el programa despliega la respuesta número cuenta%RESPUESTAS. La primera respuesta del archivo es la número 0, la que sigue es la número 1, la siguiente es la 2, etc. RESPUESTAS se define en una declaración de constante, como se muestra en seguida, de modo que sea igual al número de respuestas que hay en el archivo de respuestas:

```
const int RESPUESTAS = 8;
```

De esta forma, podremos modificar el archivo de respuestas de modo que contenga más o menos respuestas; sólo será necesario cambiar la declaración de la constante para que el programa funcione correctamente con diferentes números de posibles respuestas. Suponga que la respuesta que aparece primero en el archivo siempre será la que sigue, aunque se modifique el archivo de respuestas.

```
No estoy seguro pero creo que encontrarás la respuesta en el Capítulo #N.
```

Al sustituir los dos caracteres #N por un número, utilice el número (cuenta%NUMERO_DE_CAPITULOS + 1), donde cuenta es la variable antes mencionada y NUMERO_DE_CAPITULOS es una constante global con nombre definida como igual al número de capítulos de este libro.

15. Este programa numera las líneas de un archivo de texto. Escriba un programa que lea texto de un archivo y envíe a la salida cada línea precedida por un número de línea. Escriba el número de línea ajustado a la derecha en un campo de 3 espacios. Después del número de línea, escriba un signo de dos puntos, un espacio y el texto de la línea. Obtenga un carácter a la vez, y escriba código que se salte los espacios iniciales de cada línea. Puede suponer que las líneas son lo bastante cortas como para caber en una línea de la pantalla. Si no lo son, permita el comportamiento predeterminado de la pantalla o la impresora (es decir, continuar en otra línea o truncar).

    Una versión un poco más difícil determina el número de espacios que se requieren en el campo de número de línea contando las líneas del archivo antes de procesarlas. Esta versión del programa deberá insertar una nueva línea después de la última palabra completa que quepa en una línea de 72 caracteres.

16. Escriba un programa que realice las siguientes estadísticas para un archivo y las envíe tanto a la pantalla como a otro archivo: el total de las ocurrencias de caracteres en el archivo, caracteres sin espacio en blanco y el de las ocurrencia de letras que hay en el archivo.

6 Definición de clases

6 Definición de clases

Ha llegado la hora, dijo la morsa,
de que hablemos de muchas cosas, de barcos,
lacres y zapatos, de reyes y repollos.

LEWIS CARROLL, *Alicia a través del espejo*

Introducción

En el capítulo 5 aprendimos a utilizar clases y objetos, pero no a definir dichas clases. En este capítulo aprenderemos a definir nuestras propias clases. Una clase es un tipo de datos. Podemos usar las clases que definimos de la misma forma que usamos los tipos de datos predefinidos, como `int`, `char` e `ifstream`. Sin embargo, a menos que definamos nuestras clases debidamente, no tendrán tan buena conducta como los tipos de datos predefinidos. Por lo tanto, nos tomaremos un tiempo para explicar lo que se hace para una buena definición de una clase y lo ayudaremos con algunas técnicas para definir clases en una manera consistente con prácticas de programación modernas.

Antes de explicar las clases, presentaremos primero las estructuras. Cuando las estructuras se utilizan de la manera que las presentamos aquí, serán como clases simplificadas y nos servirán de base para entender las clases.

Prerrequisitos

Este capítulo utiliza material de los capítulos 2 al 5. Si desea completar las estructuras de control antes de leer este capítulo, puede hacerlo. El capítulo 7, que habla más de estructuras de control, no necesitará material de este capítulo.

6.1 Estructuras

Como dijimos en el capítulo 5, un objeto es una variable que tiene funciones miembro, y una clase es un tipo de datos cuyas variables son objetos. Por lo tanto, la definición de una clase debe ser una definición de tipo de datos que describa dos cosas: 1) los tipos de valores que las variables pueden contener y 2) las funciones miembro. Haremos esto en dos pasos. Primero explicaremos cómo dar una definición de tipo de una *estructura*. Podemos considerar una estructura (de la índole que veremos aquí) como un objeto que no tiene funciones miembro. Una vez que entendamos las estructuras, la definición de clases será una extensión natural.

Estructuras para datos diversos

En ocasiones resulta útil tener una colección de valores de diferentes tipos y tratar la colección como una sola cosa. Por ejemplo, consideremos un certificado de depósito bancario, que suele abreviarse CD. Un CD es una cuenta bancaria que no permite retiros durante cierto número de meses. Naturalmente, un CD tiene asociados tres elementos de información: el saldo de la cuenta, la tasa de interés de la cuenta y el plazo, que es el número de meses hasta el vencimiento. Los dos primeros elementos pueden representarse como valores de tipo `double`, y el número de meses puede representarse como un valor de tipo `int`. En el cuadro 6.1 se muestra la definición de una estructura llamada `CuentaCD` que podemos usar para este tipo de cuenta. La definición está incrustada en un programa completo que demuestra esta definición de tipo de estructura. Como se puede ver por el diálogo de ejemplo, el banco en cuestión se especializa en CD a corto plazo, por lo que el plazo siempre será de 12 o menos meses. Veamos cómo se define y utiliza esta estructura de ejemplo.

`struct`

La definición de la estructura es la siguiente:

```
struct CuentaCD
{
    double saldo;
    double tasa_interes;
    int plazo;  //meses hasta el vencimiento
};
```

La palabra clave `struct` indica que se trata de una definición de tipo de estructura. El identificador `CuentaCD` es el nombre del tipo de estructura; y a este nombre se le conoce como **etiqueta de estructura**. Esta etiqueta puede ser cualquier identificador válido (pero no una palabra clave). Aunque el lenguaje C++ no lo requiere, las etiquetas de estructura por lo regular se escriben con una combinación de mayúsculas y minúsculas, comenzando con mayúsculas. Los identificadores que se declaran dentro de las llaves `{}` son los **nombres de los miembros**. Como ilustra este ejemplo, una definición de tipo de estructura termina con una llave `}` y con un signo de punto y coma.

etiqueta de estructura

nombres de miembros

Las definiciones de estructuras normalmente se colocan fuera de cualesquier definiciones de funciones (así como las declaraciones de las constantes definidas globalmente se colocan fuera de todas las definiciones de funciones). Así, el tipo de estructura puede utilizarse en todo el código que sigue a la definición de la estructura.

dónde colocar una definición de estructura

Una vez definido un tipo de estructura, éste puede usarse igual que los tipos predefinidos `int`, `char`, etcétera. Por ejemplo, lo siguiente declara dos variables, llamadas `mi_cuenta` y `tu_cuenta`, ambas de tipo `CuentaCD`:

variables de estructura

```
CuentaCD mi_cuenta, tu_cuenta;
```

CUADRO 6.1 *Una definición de estructura (parte 1 de 2)*

```
//Programa para demostrar el tipo de estructura CuentaCD
#include <iostream>
using namespace std;

//Estructura para un certificado de depósito bancario:
struct CuentaCD
{
    double saldo;
    double tasa_interes;
    int plazo; //meses hasta el vencimiento
};

void obtener_datos(CuentaCD& la_cuenta);
//Postcondición: Se han dado a la_cuenta.saldo y a la_cuenta.tasa_interes
//valores que el usuario introdujo con el teclado.

int main()
{
    CuentaCD cuenta;
    obtener_datos(cuenta);

    double fraccion_tasa, intereses;
    fraccion_tasa = cuenta.tasa_interes/100.0;
    intereses = cuenta.saldo*fraccion_tasa*(cuenta.plazo/12.0);
    cuenta.saldo = cuenta.saldo + intereses;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Cuando su Cuenta Bancaria venza en "
         << cuenta.plazo << " meses,\n"
         << "tendra un saldo de $"
         << cuenta.saldo << endl;
    return 0;
}
```

CUADRO 6.1 *Una definición de estructura (parte 2 de 2)*

```
//Usa iostream:
void obtener_datos(CuentaCD& la_cuenta)
{
    cout << "Escriba el saldo de la cuenta: $";
    cin >> la_cuenta.saldo;
    cout << "Escriba la tasa de interes de la cuenta: ";
    cin >> la_cuenta.tasa_interes;
    cout << "Escriba el numero de meses al vencimiento\n"
         << "(deben ser 12 o menos meses): ";
    cin >> la_cuenta.plazo;
}
```

Diálogo de ejemplo

```
Escriba el saldo de la cuenta: $100.00
Escriba la tasa de interes de la cuenta: 10.0
Escriba el numero de meses al vencimiento
(deben ser 12 o menos meses): 6
Cuando su Cuenta Bancaria venza en 6 meses,
tendra un saldo de $105.00
```

valor de estructura
valores miembros

Una variable de estructura puede contener valores igual que cualquier otra variable. Un **valor de estructura** es una colección de valores más pequeños llamados **valores miembro**. Hay un valor miembro por cada nombre de miembro declarado en la definición de la estructura. Por ejemplo, un valor de tipo `CuentaCD` es una colección de tres valores miembro, dos de tipo *`double`* y uno de tipo *`int`*. Los valores miembro que juntos constituyen el valor de estructura se guardan en *variables miembro*, que veremos a continuación.

Cada tipo de estructura especifica una lista de nombres de miembros. En el cuadro 6.1 la estructura `CuentaCD` tiene los tres nombres de miembro `saldo`, `tasa_interes` y `plazo`. Cada uno de estos nombres de miembro puede servir para distinguir una variable más pequeña que forma parte de las variables de estructura mayor. Estas variables más pequeñas se llaman **variables miembro,** y se especifican dando el nombre de la variable de estructura seguida de un punto y del nombre del miembro. Por ejemplo, si `cuenta` es una variable de estructura del tipo `CuentaCD` (según la declaración del cuadro 6.1), tendrá las tres siguientes variables miembro:

variables miembro

```
cuenta.saldo
cuenta.tasa_interes
cuenta.plazo
```

Las dos primeras variables miembro son de tipo `double` y la última es de tipo `int`. Estas variables miembro se pueden emplear igual que cualquier otra variable de esos tipos. Por ejemplo, podemos dar valores a las variables miembro anteriores con las tres instrucciones de asignación que se muestran a continuación:

```
cuenta.saldo = 1000.00;
cuenta.tasa_interes = 4.7;
cuenta.plazo = 11;
```

El resultado de estas tres instrucciones se muestra de forma diagramática en el cuadro 6.2. Las variables miembro pueden emplearse de todas las formas que se pueden usar las variables ordinarias. Por ejemplo, la siguiente línea del programa del cuadro 6.1 suma el valor contenido en la variable miembro `cuenta.saldo` y el valor contenido en la variable ordinaria `intereses` y luego coloca el resultado en la variable miembro `cuenta.saldo`:

```
cuenta.saldo = cuenta.saldo + intereses;
```

operador punto

Observe que una variable miembro de una variable de estructura se especifica usando el operador punto exactamente de la misma forma que lo usamos en el capítulo 5, donde servía para especificar una función miembro de una clase. La única diferencia es que en el caso de las estructuras los miembros son variables en lugar de funciones.

reutilización de nombres de miembros

Dos o más tipos de estructura pueden emplear los mismos nombres de miembros. Por ejemplo, es perfectamente válido tener en el mismo programa las dos siguientes definiciones de tipo:

```
struct ReservaFertilizante
{
    double cantidad;
    double contenido_de_nitrogeno;
};
```

y

```
struct Cosecha
{
    int cantidad;
    double magnitud;
};
```

Esta coincidencia de nombres no produce problemas. Por ejemplo, si declaramos estas dos variables de estructura:

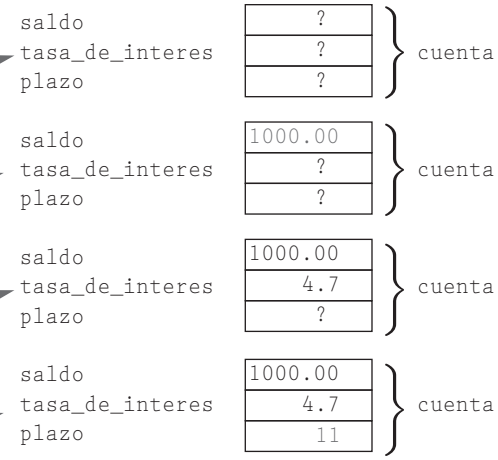
```
ReservaFertilizante super_crece;
Rendimiento manzanas;
```

la cantidad del fertilizante `super_crece` se almacenará en la variable miembro `super_crece.cantidad`, y la cantidad de manzanas cosechadas se almacenará en la variable miembro `manzanas.cantidad`. El operador punto y la variable de estructura especifican a cuál `cantidad` nos referimos en cada caso.

Podemos considerar un valor de estructura como una colección de valores miembro. Visto de esta forma, un valor de estructura consiste en muchos valores distintos. También

CUADRO 6.2 *Valores miembro*

```
struct CuentaCD
{
    double saldo;
    double tasa_interes;
    int plazo; //meses hasta el vencimiento
};
int main()
{
    CuentaCD cuenta;
      ...
```




```
    cuenta.saldo = 1000.00;
```



```
    cuenta.tasa_de_interes = 4.7;
```



```
    cuenta.plazo = 11;
```


variables de estructura en instrucciones de asignación

lo podemos ver como un solo valor (complejo) que está formado por valores miembro. Puesto que podemos ver un valor de estructura como un solo valor, los valores y las variables de estructura se pueden usar de las mismas formas que usamos valores y variables sencillos de tipos predeterminados como *int*. En particular, podemos asignar valores de estructura usando el signo igual. Por ejemplo, si `manzanas` y `naranjas` son variables de estructura del tipo `Cosecha` que definimos en la página anterior, lo que sigue es perfectamente válido:

```
manzanas = naranjas;
```

La instrucción de asignación anterior equivale a:

```
manzanas.cantidad = naranjas.cantidad;
manzanas.magnitud = naranjas.magnitud;
```

RIESGO *Olvido de un punto y coma en una definición de estructura*

Al agregar la llave final `}` a una definición de estructura podríamos sentir que ya terminamos la definición, pero no es así. También debemos colocar un signo de punto y coma después de dicha llave. Hay una razón para esto, aunque se trata de una característica que no tendremos ocasión de usar. Una definición de estructura es más que una definición; también puede servir para declarar variables de estructura. Podemos enumerar nombres de variables de estructura entre la llave final y el punto y coma final. Por ejemplo, lo que sigue define una estructura llamada `DatosClima` y declara dos variables de estructura `punto_dato1` y `punto_dato2`, ambos de tipo `DatosClima`:

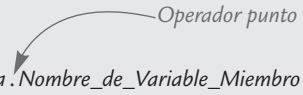
```
struct DatosClima
{
    double temperatura;
    double velocidad_viento;
} punto_dato1, punto_dato2;
```

Sin embargo, como dijimos, siempre separaremos la definición de una estructura y la declaración de variables de ese tipo de estructura, por lo que nuestras definiciones de estructuras siempre tendrán un signo de punto y coma inmediatamente después de la llave final.

El operador punto

El **operador punto** sirve para especificar una variable miembro de una variable de estructura.

Sintaxis



Ejemplos

```
struct RegistroEstudiante
{
    int numero_estudiante;
    char calif;
};
int main()
{
    RegistroEstudiante su_registro;
    su_registro.numero_estudiante = 2001;
    su_registro.calif = 'A';
```

Algunos escritores llaman al operador punto *operador de acceso a miembros de estructuras*, pero no usaremos ese término.

Estructuras como argumentos de función

argumentos de estructura

Una función puede tener parámetros de llamada por valor de un tipo de estructura y/o parámetros de llamada por referencia de un tipo de estructura. Por ejemplo, el programa

del cuadro 6.1, incluye una función llamada `obtener_datos` que tiene un parámetro de llamada por referencia del tipo de estructura `CuentaCD`.

Las funciones pueden devolver estructuras

Un tipo de estructura también puede ser el tipo de un valor devuelto por una función. Por ejemplo, lo que se muestra a continuación define una función que recibe tres argumentos apropiados y devuelve un valor de tipo `CuentaCD`:

```
CuentaCD empaquetar(double el_saldo,
                    double la_tasa, int el_plazo)
{
      CuentaCD temp;
      temp.saldo = el_saldo;
      temp.tasa_de_interes = la_tasa;
      temp.plazo = el_plazo;
      return temp;
}
```

Observe la variable local `temp` de tipo `CuentaCD`; la utilizamos para construir un valor de estructura completo, que es devuelto por la función. Una vez que la función `empaquetar` está definida podremos dar un valor a una variable de tipo `CuentaCD` como se ilustra a continuación:

```
CuentaCD nueva_cuenta;
nueva_cuenta = empaquetar(1000.00, 5.1, 11);
```

TIP DE PROGRAMACIÓN

Utilice estructuras jerárquicas

estructuras dentro de estructuras

En ocasiones es razonable tener estructuras cuyos miembros son a su vez estructuras más pequeñas. Por ejemplo, un tipo de estructura llamado `InfoPersona` en el que podríamos almacenar la estatura, peso y fecha de nacimiento de una persona se podría definir como sigue:

```
struct Fecha
{
     int dia;
     int mes;
     int anio;
};
struct InfoPersona
{
     double estatura;//en centímetros
     int peso;//en kilogramos
     Fecha fecha_nacimiento;
};
```

Una variable de estructura de tipo `InfoPersona` se declara de la forma acostumbrada:

```
InfoPersona persona1;
```

Tipos de estructura simples

Definimos un **tipo de estructura** como se muestra a continuación. La *Etiqueta_de_Estructura* es el nombre del tipo de estructura.

Sintaxis

```
struct Etiqueta_de_Estructura
{
  Tipo_1 Nombre_de_Variable_Miembro_1;
  Tipo_2 Nombre_de_Variable_Miembro_2;
                    .
                    .
                    .

  Ultimo_Tipo Ultimo_Nombre_de_Variable_Miembro;
};
```

No olvide este punto y coma.

Ejemplo:

```
struct Automovil
{
    int modelo;
    int puertas;
    double potencia;
    char tipo;
};
```

Aunque no usaremos esta característica, podemos combinar nombres de miembros del mismo tipo en una sola lista, separándolos con comas. Por ejemplo, lo que se muestra a continuación es equivalente a la definición de estructura anterior:

```
struct Automovil
{
    int modelo, puertas;
    double potencia;
    char tipo;
};
```

Podemos declarar **variables de un tipo de estructura** igual que declaramos variables de otros tipos. Por ejemplo:

```
Automovil mi_coche, tu_coche;
```

Las variables miembro se especifican empleando el **operador punto**. Por ejemplo:

`mi_coche.modelo`,
`mi_coche.puertas`, `mi_coche.potencia` y `mi_coche.tipo`.

Si asignamos a la variable de estructura `personal` un valor que registra la fecha de nacimiento de una persona, podemos desplegar en la pantalla el año en que nació esa persona, como se muestra a continuación:

```
cout << personal.fecha_nacimiento.anio;
```

La forma de leer este tipo de expresiones es de izquierda a derecha, y con mucho cuidado. Comenzando por el extremo izquierdo, `personal` es una variable de estructura de tipo `InfoPersona`. Para obtener la variable miembro cuyo nombre es `fecha_nacimiento`, utilizamos el operador punto como sigue:

```
personal.fecha_nacimiento
```

Esta variable miembro es también una variable de estructura de tipo `Fecha`; por lo tanto, ella misma tiene variables miembro. Obtenemos una variable miembro de la variable de estructura `personal.fecha_nacimiento` mediante la adición de un punto y el nombre de la variable miembro, digamos `anio`, lo que produce la expresión `personal.fecha_nacimiento.anio` que vimos antes.

Inicialización de estructuras

Podemos inicializar una estructura en el momento de declararla. Para dar un valor a una variable de estructura, colocamos después de su nombre un signo de igual y una lista de los valores miembro encerrados en llaves. Por ejemplo, en la subsección anterior dimos la siguiente definición de un tipo de estructura para guardar fechas:

```
struct Fecha
{
    int dia;
    int mes;
    int anio;
};
```

Una vez definido el tipo `Fecha`, podemos declarar e inicializar una variable de estructura llamada `vencimiento` de la siguiente manera:

```
Fecha_vencimiento = {31, 12, 2004};
```

Observe que los valores inicializadores deben darse en el orden que corresponde al orden de las variables miembro en la definición de tipo de estructura. En este ejemplo, `vencimiento.dia` recibe el primer valor inicializador de `31`, `vencimiento.mes` recibe el segundo valor inicializador de `12` y `vencimiento.anio` recibe el tercer valor de `2004`.

Es un error si hay más inicializadores que miembros de la estructura. Si hay menos valores inicializadores que miembros de la estructura, los valores proporcionados se usan para inicializar en orden los datos miembros. Los datos miembros que no tienen inicializador se inicializan con un valor cero de un tipo apropiado para la variable.

Ejercicios de AUTOEVALUACIÓN

1. Dada la estructura y la declaración de variable de estructura:

```
struct CuentaCD
{
    double saldo;
    double tasa_de_interes;
    int plazo;
    char inicial1;
    char inicial2;
};
CuentaCD cuenta;
```

¿qué tipo tiene cada una de las siguientes variables? Marque las que no sean correctas.

a) `cuenta.saldo`
b) `cuenta.tasa_de_interes`
c) `CuentaCD.plazo`
d) `cuenta_ahorros.inicial1`
e) `cuenta.inicial2`
f) `cuenta`

2. Considere la siguiente definición de tipo:

```
struct TipoZapato
{
    char estilo;
    double precio;
};
```

Dada la definición de tipo de estructura anterior, ¿qué salidas producirá el siguiente código?

```
TipoZapato zapato1, zapato2;
zapato1.estilo = 'A';
zapato1.precio = 9.99;
cout << zapato1.estilo << " $" << zapato1.precio << endl;
zapato2 = zapato1;

zapato2.precio = zapato2.precio/9;
cout << zapato2.estilo << " $" << zapato2.precio << endl;
```

3. ¿Qué error tiene la siguiente definición de estructura? ¿Qué mensaje produce el compilador para este error? Describa el error en sus propias palabras.

```
struct Cosas
{
    int b;
    int c;
}
int main()
```

```
{
    Cosas x;
    //más código
}
```

4. Dada la siguiente definición de `struct`:

```
struct A
{
    int miembro_b;
    int miembro_c;
};
```

declare `x` como una estructura de este tipo. Inicialice los miembros de `x`, el `miembro_b` y el `miembro_c`, con los valores `1` y `2`, respectivamente.

Nota: Se solicita una inicialización, no una asignación a los miembros. Esta distinción es importante y se hará en el texto en un capítulo posterior.

5. Lo que se muestra a continuación son inicializaciones de un tipo de estructura. Diga qué sucede con cada inicialización. Indique si hay problemas con ellas.

```
struct Fecha
{
   int dia;
   int mes;
   int anio;
};
a) Fecha vencimiento = {21, 12};
b) Fecha vencimiento = {21, 12, 2022};
c) Fecha vencimiento = {21, 12, 20, 22};
d) Fecha vencimiento = {21, 12, 22};
```

6. Escriba una definición de un tipo de estructura para registros que consisten en el sueldo de una persona, las vacaciones acumuladas (cierto número de días) y el tipo de sueldo (por hora o mensual). Represente el tipo de sueldo como uno de los dos valores `char 'H'` o `'M'`. Llame al tipo `RegistroEmpleado`.

7. Dé una definición de función que corresponda a la siguiente declaración de función. (El tipo `TipoZapato` se da en el ejercicio de autoevaluación 2.)

```
void leer_registro_zapato(TipoZapato& zapato_nuevo);
//Llena zapato_nuevo con valores que se leen del teclado.
```

8. Dé una definición de función que corresponda a la siguiente declaración de función. (El tipo `TipoZapato` se da en el ejercicio de autoevaluación 2.)

```
TipoZapato descuento(TipoZapato registro_viejo);
//Devuelve una estructura igual a su argumento, pero con el
//precio reducido en un 10%.
```

9. Dé una definición de estructura para un tipo llamado `RegistroExistencias` que tiene dos variables miembro, una llamada `info_zapato` del tipo `TipoZapato`, el cual se vio en el ejercicio de autoevaluación 2, y la otra variable llamada `fecha_llegada` del tipo `Fecha` que se dio en el ejercicio de autoevaluación 5.

10. Declare una variable de tipo `RegistroExistencias` (del ejercicio anterior) y escriba una instrucción que establecerá a 2004 como el año de la fecha de llegada.

6.2 Clases

No quisiera pertenecer a ningún club que me acepte como miembro.

Groucho Marx, *The Groucho letters*

Definición de clases y funciones miembro

clase
objeto

Una **clase** es un tipo de datos cuyas variables son objetos. En el capítulo 5 describimos un **objeto** como una variable que tiene funciones miembro además de la capacidad para contener valores de datos.[1] Así pues, dentro de un programa en C++, la definición de una clase debe ser una definición de tipo de datos que describa los tipos de valores que las variables pueden contener y también la naturaleza de las funciones miembro. Una definición de estructura describe algunas de estas cosas. Una estructura es un tipo definido que nos permite definir valores del tipo de estructura definiendo variables miembro. Para obtener una clase a partir de una estructura, lo único que falta es añadir algunas funciones miembro.

En el programa del cuadro 6.3 se presenta un ejemplo de definición de clase. El tipo `DiaDelAnio` que se presenta ahí es una definición de clase para objetos cuyos valores son fechas, como enero 1 o julio 4. Estos valores pueden servir para registrar días feriados, cumpleaños y otras fechas especiales. En esta definición de `DiaDelAnio`, el mes se registra como un valor *`int`*, y `1` representa enero, `2` representa febrero, etcétera. El día del mes se registra en otra variable *`int`*. La clase `DiaDelAnio` tiene una función miembro llamada `salida`, que no tiene argumentos y envía a la pantalla los valores del mes y el día. Veamos con detalle la definición de la clase `DiaDelAnio`.

La definición de la clase `DiaDelAnio` aparece casi al principio del cuadro 6.3. Por el momento, haga caso omiso de la línea que contiene la palabra clave *`public`*. Esta línea simplemente dice que las variables y funciones miembro no tienen restricciones. Explicaremos esta línea más adelante en el capítulo. El resto de la definición de la clase `DiaDelAnio` se parece mucho a una definición de estructura, excepto que usa la palabra clave *`class`* en lugar de *`struct`* y enumera la función miembro `salida` (además de las variables miembro `mes` y `dia`). Observe que la función miembro `salida` se enumera dando su declaración de función. Las definiciones de las funciones miembro aparecen en cualquier lugar. (En una definición de clase en C++ podemos mezclar las variables miembro y las funciones miembro como queramos, pero siempre deberemos tratar de enumerar las funciones miembro antes que las variables miembro.) Los objetos (es decir, variables) de un tipo de clase se declaran de la misma manera que las variables de los tipos predefinidos y que las variables de estructura.

función miembro

[1] El objeto es actualmente el valor de la variable más que la variable por sí misma, pero dado que usamos la variable para nombrar el valor que contiene, podemos simplificar esta discusión ignorando lo anterior y hablar como si la variable y el valor fueran lo mismo.

CUADRO 6.3 *Clase con una función miembro (parte 1 de 2)*

```
//Programa para demostrar un ejemplo de clase muy sencillo.
//En el cuadro 6.4 se dará una versión mejor de DiaDelAnio.
#include <iostream>
using namespace std;

class DiaDelAnio
{
public:
    void salida();
    int mes;
    int dia;
};

int main()
{
    DiaDelAnio hoy, cumpleanios;

    cout << "Escriba la fecha de hoy:\n";
    cout << "Escriba el mes como un numero: ";
    cin >> hoy.mes;
    cout << "Escriba el dia del mes: ";
    cin >> hoy.dia;
    cout << "Escriba su cumpleanios:\n";
    cout << "Escriba el mes como un numero: ";
    cin >> cumpleanios.mes;
    cout << "Escriba el dia del mes: ";
    cin >> cumpleanios.dia;

    cout << "La fecha de hoy es ";
    hoy.salida();
    cout << "Su cumpleanios es ";
    cumpleanios.salida();

    if (hoy.mes == cumpleanios.mes
       && hoy.dia == cumpleanios.dia)
       cout << "Felicidades!\n";
    else
       cout << "Feliz no cumpleanios!\n";
    return 0;
}
```

Declaración de función miembro.

Llamadas a la función miembro `salida`.

CUADRO 6.3 *Clase con una función miembro (parte 2 de 2)*

```
//Usa iostream:
void DiaDelAnio::salida()
{
    cout << "mes = "   << mes
         << ", dia = " << dia << endl;
}
```

Diálogo de ejemplo

```
Escriba la fecha de hoy:
Escriba el mes como un numero: 10
Escriba el dia del mes: 15
Escriba su cumpleanios:
Escriba el mes como un numero: 2
Escriba el dia del mes: 21
La fecha de hoy es mes = 10, dia = 15
Su cumpleanios es mes = 2, dia = 21
Feliz no cumpleanios!
```

llamado de funciones miembro

Las funciones miembro de las clases que definimos se invocan de la misma manera que las de las clases predefinidas (vea el capítulo 5). Por ejemplo, el programa del cuadro 6.3 declara dos objetos de tipo `DiaDelAnio` de la siguiente manera:

```
DiaDelAnio hoy, cumpleanios;
```

La función miembro `salida` se invoca con el objeto `hoy` así:

```
hoy.salida();
```

y luego se invoca con el objeto `cumpleanios` como se muestra a continuación:

```
cumpleanios.salida();
```

Encapsulamiento

encapsulamiento

Combinar varias cosas, digamos variables o funciones, en un solo "paquete", digamos un objeto de alguna clase, se denomina **encapsulamiento**.

definición de funciones miembro

Cuando se define una función miembro, la definición debe incluir el nombre de la clase, porque podría haber dos o más clases que tengan funciones miembro con el mismo nombre. En el cuadro 6.3 sólo hay una definición de clase, pero en otras situaciones podríamos tener muchas definiciones de clase, y cada clase podría tener una función miembro llamada `salida`. La definición de la función miembro `salida` de la clase `DiaDelAnio` se muestra en la parte 2 del cuadro 6.3. La definición es similar a una definición de función ordinaria, pero hay algunas diferencias.

El encabezado de la definición de la función miembro `salida` es el siguiente:

```
void DiaDelAnio::salida()
```

operador de resolución de alcance

El operador `::` es el **operador de resolución de alcance**, y tiene un propósito similar al del operador punto. Tanto uno como el otro sirven para indicar de quién es parte una función miembro. Sin embargo, el operador de resolución de alcance `::` se usa con un nombre de clase, mientras que el operador punto se utiliza con objetos (es decir, con variables de clase). El operador de resolución de alcance consiste en dos signos de dos puntos sin espacio entre ellos. El nombre de clase que precede al operador de resolución de alcance suele llamarse **calificador de tipo**, porque especializa ("califica") el nombre de función a un tipo en particular.

calificador de tipo

variables miembro en definiciones de funciones

Examine la definición de la función miembro `DiaDelAnio::salida` que se muestra en el cuadro 6.3. Observe que en esa definición de función empleamos los nombres de miembro `mes` y `dia` solos, sin dar primero el objeto y el operador punto. Esto no es tan extraño como podría parecer a primera vista. A estas alturas simplemente estamos definiendo la función miembro `salida`. Esta definición de `salida` aplicará a todos los objetos de tipo `DiaDelAnio`, pero a estas alturas no conocemos los nombres de los objetos de ese tipo que usaremos, por lo que no podemos dar sus nombres. Cuando invoquemos la función miembro, como en

```
hoy.salida();
```

todos los nombres de miembros de la definición de función se especializarán al nombre de la clase invocadora. Por tanto, la llamada de función anterior equivale a lo siguiente:

```
{
   cout << "mes = "   << mes.hoy
        << ", dia = " << dia.hoy << endl;
}
```

En la definición de una función miembro podemos emplear los nombres de todos los miembros de esa clase (tanto los datos miembro como las funciones miembro) sin usar el operador punto.

Definición de función miembro

Una función miembro se define de la misma manera que todas las funciones, excepto que en el encabezado de la función se dan el *Nombre_de_Clase* y el operador de resolución de alcance `::`.

Sintaxis

Tipo_Devuelto Nombre_de_Clase`::`*Nombre_de_Funcion*`(`*Lista_de_Parametros*`)`
`{`
    Instrucciones_del_Cuerpo_de_la_Funcion
`}`

Ejemplo

```
//Utiliza iostream:
void DiaDelAnio::salida()
{
     cout << "mes = "   << mes
          << ", dia = " << dia << endl;
}
```

En el cuadro 6.3 se muestra la definición de la clase `DiaDelAnio` que usamos como ejemplo, donde `mes` y `dia` se definen como nombres de variables miembro de la clase `DiaDelAnio`. Observe que `mes` y `dia` no van precedidas de un nombre de objeto y un punto.

El operador punto y el de resolución de alcance

Tanto el operador punto como el de resolución de alcance se emplean con los nombres de miembros para especificar de qué cosa son miembros. Por ejemplo, supongamos que hemos declarado una clase llamada `DiaDelAnio` y declaramos un objeto llamado `hoy` de la siguiente manera:

```
DiaDelAnio hoy;
```

Usamos el **operador punto** para especificar un miembro del objeto `hoy`. Por ejemplo, `salida` es una función miembro de la clase `DiaDelAnio` (que se define en el cuadro 6.3); la siguiente llamada de función desplegará los valores de datos almacenados en el objeto `hoy`:

```
hoy.salida();
```

Empleamos el **operador de resolución de alcance** `::` para especificar el nombre de clase cuando damos la definición de una función miembro. Por ejemplo, el encabezado de la definición de la función miembro `salida` sería el siguiente:

```
void DiaDelAnio::salida()
```

Recuerde que el operador de resolución de alcance `::` se usa con un nombre de clase, mientras que el operador punto se usa con un objeto de esa clase.

Ejercicios de AUTOEVALUACIÓN

11. A continuación hemos redefinido la clase `DiaDelAnio` del cuadro 6.3 de modo que tenga una función miembro adicional llamada `entrada`. Escriba una definición apropiada para la función miembro `entrada`.

```
class DiaDelAnio
{
public:
    void entrada();
    void salida();
    int dia;
    int mes;
};
```

12. Dada la siguiente definición de clase, escriba una definición apropiada para la función miembro `fijar`:

```
class Temperatura
{
public:
     void fijar(double nuevos_grados, char nueva_escala);
     //Asigna a las variables miembro los valores que se dan
     //como argumentos.

    double grados;
    char escala; //'F' para Fahrenheit y 'C' para Celsius.
};
```

13. Explique detalladamente las diferencias de significado y uso del operador punto y el de resolución de alcance `::`.

Miembros públicos y privados

Los tipos predefinidos como `double` no se implementan como clases en C++, pero la gente que escribió el compilador de C++ que utilizamos sí diseñó una forma de representar valores de tipo `double` en su computadora. Es posible implementar el tipo `double` de diversas formas. De hecho, diferentes versiones de C++ implementan el tipo `double` de formas ligeramente distintas, pero si trasladamos nuestro programa en C++ de una computadora a otra que tiene una implementación diferente del tipo `double`, el programa deberá funcionar correctamente.[2] Las clases son tipos que definimos, y tales tipos deben comportarse tan bien como los tipos predefinidos. Podemos construir una biblioteca con nuestras propias definiciones de tipos de clases y usar nuestros tipos como si fueran tipos predefinidos. Por ejemplo, podríamos colocar cada definición de clase en un archivo aparte y copiarla en cualquier programa que use ese tipo.

Es recomendable que nuestras definiciones de clase separen las reglas para usar la clase y los detalles de la implementación de la clase de forma tan definitiva como se hizo con los tipos predefinidos. Si modificamos la implementación de una clase (por ejemplo, alterando algunos detalles de la definición de una función miembro con el fin de que las llamadas

[2] Hay ocasiones en que no se alcanza plenamente este ideal, pero en el mundo ideal debe alcanzarse y, al menos en el caso de programas sencillos, se alcanza incluso en el mundo imperfecto en el que vivimos.

a esa función sean más rápidas) no deberá ser necesario modificar ninguna otra parte de los programas. Para alcanzar este ideal, necesitamos describir una característica más de las definiciones de clase.

Examinemos otra vez la definición del tipo `DiaDelAnio` que se da en el cuadro 6.3. El tipo `DiaDelAnio` está diseñado para contener valores que representan fechas como cumpleaños y días feriados. Optamos por representar tales fechas como dos enteros, uno para el mes y uno para el día del mes. Posteriormente, podríamos decidir cambiar la representación del mes, de una variable de tipo `int` a tres variables de tipo `char`. En esta versión modificada, los tres caracteres serían una abreviatura del nombre del mes. Por ejemplo, los tres valores `char` `'E'`, `'n'` y `'e'` representarían el mes de enero. Sin embargo, sea que usemos una sola variable miembro de tipo `int` para registrar el mes o tres variables miembro de tipo `char`, éste es un detalle de implementación que no deberá preocupar a un programador que use el tipo `DiaDelAnio`. Desde luego, si modificamos la forma en que la clase `DiaDelAnio` representa el mes, también deberemos modificar la implementación de la función miembro `salida`, pero eso es todo lo que habrá que cambiar. No deberá ser necesario modificar ninguna otra parte de un programa que use nuestra definición de la clase `DiaDelAnio`. Lamentablemente, el programa del cuadro 6.3 no cumple con este ideal. Por ejemplo, si sustituimos la variable miembro llamada `mes` por tres variables miembro de tipo `char`, no habrá ninguna variable miembro llamada `mes`, y será necesario cambiar las partes del programa que obtienen entradas, así como la instrucción `if-else`.

Con una definición de clase ideal, deberemos poder modificar los detalles de implementación de la clase, y lo único que deberemos tener que cambiar en cualquier programa que use la clase serán las definiciones de las funciones miembro. A fin de alcanzar este ideal es necesario tener suficientes funciones miembro como para nunca tener que acceder a las variables miembro directamente, sino sólo a través de las funciones miembro. Entonces, si modificamos las variables miembro, sólo tendremos que alterar las definiciones de las funciones miembro de modo que concuerden con los cambios a las variables miembro, y no tendremos que modificar ninguna otra parte de nuestros programas. En el cuadro 6.4 hemos redefinido la clase `DiaDelAnio` de modo que tenga suficientes funciones miembro para hacer todo lo que queremos que hagan nuestros programas. Así, el programa no tiene que hacer referencia directamente a ninguna variable miembro. Si examinamos con detenimiento el programa del cuadro 6.4 veremos que el único lugar en el que se usan los nombres de variables miembro `mes` y `dia` es en las definiciones de las funciones miembro. No hay referencias a `hoy.mes`, `hoy.dia`, `cumpleanios_bach.mes` ni `cumpleanios_bach.dia` en ningún lugar fuera de las definiciones de las funciones miembro.

El programa del cuadro 6.4 tiene una característica nueva diseñada para asegurar que ningún programador que use la clase `DiaDelAnio` hará alguna vez referencia directa a cualquiera de sus variables miembro. Observe la línea en la definición de la clase `DiaDelAnio` que contiene la palabra clave `private`. Todos los nombres de variables miembro que se enumeran después de esta línea son **miembros privados**, lo que significa que no se puede acceder directamente a ellos en el programa, como no sea dentro de la definición de una función miembro. Si tratamos de acceder a una de estas variables miembro en la parte `main` del programa o en la definición de alguna función que no sea función miembro de esta clase en particular, el compilador generará un mensaje de error. Si insertamos la palabra clave `private` y un signo de dos puntos en la lista de variables miembro y funciones miembro, todos los miembros que sigan a la etiqueta `private`: serán **miembros privados**. Las variables que sigan a la etiqueta `private`: serán **variables miembro privadas**, y las funciones que lo sigan serán **funciones miembro privadas**.

private:

variables miembro privadas

CUADRO 6.4 *Clase con miembros privados (parte 1 de 3)*

Ésta es una versión mejorada de la clase `DiaDelAnio` que dimos en el cuadro 6.3.

```
//Programa para demostrar la clase DiaDelAnio.
#include<iostream>
using namespace std;

class DiaDelAnio
{
public:
     void entrada();
     void salida();

     void fijar(int nuevo_mes, int nuevo_dia);
     //Precondición: nuevo_mes y nuevo_dia forman una fecha válida.
     //Postcondición: La fecha se fija según los argumentos.

     int obtener_mes();
     //Devuelve el mes, 1 para enero, 2 para febrero, etc.

     int obtener_dia();
     //Devuelve el día del mes.
private:
     void checar_fecha();
     int mes;
     int dia;
};
int main()
{
     DiaDelAnio hoy, cumpleanios_bach;
     cout << "Escriba la fecha de hoy:\n";
     hoy.entrada();
     cout << "La fecha de hoy es ";
     hoy.salida();

     cumpleanios_bach.fijar(3, 21);
     cout << "El cumpleanios de J. S. Bach es ";
     cumpleanios_bach.salida();

     if ( hoy.obtener_mes() == cumpleanios_bach.obtener_mes() &&
               hoy.obtener_dia() == cumpleanios_bach.obtener_dia() )
         cout << "Feliz cumpleanios, Juan Sebastian!\n";
     else
         cout << "Feliz no cumpleanios, Juan Sebastian!\n";
     return 0;
}
```

Funciones miembro privadas

Variables miembro privadas

CUADRO 6.4 *Clase con miembros privados (parte 2 de 3)*

```
//Usa iostream:
void DiaDelAnio::entrada()
{
     cout << "Escriba el mes como un numero: ";
     cin >> mes;
     cout << "Escriba el dia del mes: ";
     cin >> dia;
     checar_fecha();
}

void DiaDelAnio::salida()
```

<El resto de la definición de DiaDelAnio::salida se da en el cuadro 6.3.>

Los miembros privados se pueden usar en las definiciones de funciones miembro (pero en ningún otro lugar).

Una mejor definición de la función miembro `input` *podría ser solicitar al usuario que reescriba la fecha en caso de haberla escrito incorrecta; pero esto se analizará hasta el capítulo 7.*

```
void DiaDelAnio::fijar(int nuevo_mes, int nuevo_dia)
{
     mes = nuevo_mes;
     dia = nuevo_dia;
     checar_fecha ();
}

void DiaDelAnio::checar_fecha()
{
     if ((mes < 1) || (mes > 12) || (dia < 1) || (dia > 31) )
     {
          cout << "Datos no validos. Abortando el programa.\n";
          exit(1);
     }
}

int DiaDelAnio::obtener_mes()
{
     return mes;
}

int DiaDelAnio::obtener_dia()
{
     return dia;
}
```

La función miembro `checar_fecha` *no verifica todas las fechas incorrectas, pero sería sencillo completar la función haciéndola más larga. Analice el ejercicio de autoevaluación 14.*

La función exit se analizó en el capítulo 5. Esta función finaliza el programa.

CUADRO 6.4 *Clase con miembros privados (parte 3 de 3)*

Diálogo de ejemplo

```
Escriba la fecha de hoy:
Escriba el mes como un numero: 3
Escriba el dia del mes: 21
La fecha de hoy es mes = 3, dia = 21
El cumpleanios de J. S. Bach es mes = 3, dia = 21
Feliz cumpleanios, Juan Sebastian!
```

Todos los miembros privados de la clase `DiaDelAnio` que se definen en el cuadro 6.4 son variables. Podemos usar una variable miembro privada en la definición de cualquiera de las funciones miembro, pero en ningún otro lugar. Por ejemplo, con esta definición modificada de la clase `DiaDelAnio`, ya no se permiten las dos asignaciones siguientes en la parte `main` del programa:

```
DiaDelAnio hoy; //Esta línea está bien.
hoy.mes = 12; //NO PERMITIDO
hoy.dia = 25; //NO PERMITIDO
```

No se permite ninguna referencia a estas variables privadas (excepto en la definición de funciones miembro). Puesto que esta nueva definición convierte a `mes` y `dia` en variables miembro privadas, las que siguen tampoco son válidas en la parte `main` de cualquier programa que declare `hoy` como del tipo `DiaDelAnio`:

```
cout << hoy.mes; //NO VÁLIDO
cout << hoy.dia; //NO VÁLIDO
if (hoy.mes == 1) //NO VÁLIDO
    cout << "Enero";
```

Una vez que hacemos privada a una variable miembro, no hay forma de modificar su valor (ni de hacer referencia a ella de alguna otra manera), como no sea usando una de las funciones miembro. Ésta es una restricción severa, pero generalmente es prudente imponerla. Los programadores han comprobado que su código suele ser más fácil de entender y actualizar si hacen privadas a todas las variables miembro.

Podría parecer que el programa del cuadro 6.4 no prohibe realmente el acceso directo a las variables miembro privadas, ya que pueden modificarse usando la función miembro `DiaDelAnio::fijar`, y podemos descubrir sus valores usando las funciones miembro `DiaDelAnio::obtener_mes` y `DiaDelAnio::obtener_dia`. Si bien eso es cierto en el caso del programa del cuadro 6.4, podría no ser cierto si modificamos la forma de repre-

sentar el mes y/o el día en nuestras fechas. Por ejemplo, supongamos que cambiamos la definición del tipo `DiaDelAnio` a la siguiente:

```
class DiaDelAnio
{
public:
    void entrada();
    void salida();
    void fijar(int nuevo_mes, int nuevo_dia);
    //Precondición: nuevo_mes y nuevo_dia forman una fecha válida.
    //Postcondición: La fecha se fija según los argumentos.
    int obtener_mes();
    //Devuelve el mes, 1 para enero, 2 para febrero, etcétera.
    int obtener_dia();
    //Devuelve el día del mes.
private:
    void DiaDelAnio::checar_fecha();
    char primera_letra; //del mes
    char segunda_letra; //del mes
    char tercera_letra; //del mes
    int dia;
};
```

Entonces sería un poco más difícil definir las funciones miembro, pero podrían redefinirse de modo que se comportaran *exactamente* igual que antes. Por ejemplo, la definición de la función `obtener_mes` podría comenzar de la siguiente manera:

```
int DiaDelAnio::obtener_mes()
{
    if (primera_letra == 'E' && segunda_letra == 'n'
              && tercera_letra == 'e')
        return 1;
    if (primera_letra == 'F' && segunda_letra == 'e'
              && tercera_letra == 'b')
        return 2;
        . . .
```

Esto sería muy tedioso, pero no difícil.

Además, observe que las funciones miembro `DiaDelAnio::fijar` y `DiaDelAnio::entrada` coincidan para asegurarse que las variables miembro `mes` y `dia` se establezcan como valores válidos. Esto se realiza con una llamada a la función miembro `DiaDelAnio::checar_fecha`. Si las variables miembro `mes` y `dia` fueran públicas en lugar de privadas, éstas se podrían fijar a cualquier valor incluyendo valores no válidos. Al hacer a las variables miembro privadas y manipularlas sólo a través de funciones miembro podemos asegurarles que nunca se fijarán a valores no válidos o sin sentido. (En el ejercicio de autoevaluación 14, se le pide redefinir la función miembro `DiaDelAnio::checar_fecha` de manera que coincida completamente para fechas no válidas.)

funciones miembro privadas

También es posible hacer privada a una función miembro. Al igual que las variables miembro privadas, las funciones miembro privadas se pueden usar en la definición de cualquier otra función miembro, pero en ningún otro lugar por ejemplo, no en la parte `main` de un programa que utiliza el tipo de clase.

Por ejemplo, la función miembro `DiaDelAnio::checar_fecha` del cuadro 6.4 es una función miembro privada. Por lo general, una función miembro se hace privada sólo si piensa utilizarla como función auxiliar en las definiciones de funciones miembro.

public:

Empleamos la palabra clave *public* para indicar los **miembros públicos** de la misma forma que usamos *private* para indicar miembros privados. Por ejemplo, en el caso de la clase DiaDelAnio que se define en el cuadro 6.4, todas las funciones miembro exeptuando DiaDelAnio::checar_fecha son miembros públicos (y todas las variables miembro son miembros privados). Podemos emplear un miembro público en la parte main del programa o en la definición de cualquier función, incluso una función no miembro.

Podemos tener cualquier cantidad de ocurrencias de *public* y *private* en una definición de clase. Cada vez que insertamos la etiqueta

```
public:
```

la lista de miembros cambia de privada a pública. Cada vez que insertamos la etiqueta

```
private:
```

la lista de miembros cambia otra vez a miembros privados. Por ejemplo, la función miembro hacer_otra_cosa y la variable miembro mas_cosas de la siguiente definición de estructura son miembros privados, mientras que los otros cuatro miembros son públicos:

```
class ClaseMuestra
{
public:
    void hacer_algo();
    int cosas;
private:
    void hacer_otra_cosa();
    char mas_cosas;
public:
    double hacer_algo_distinto();
    double cosas_distintas;
};
```

Si enumeramos los miembros al principio de la definición de la clase y no insertamos ni *public*: ni *private*: antes de esos primeros miembros, serán miembros privados. No obstante, siempre es recomendable etiquetar explícitamente cada grupo de miembros para indicar si son públicos o privados.

TIP DE PROGRAMACIÓN

Haga que todas las variables miembro sean privadas

Al definir una clase, lo que se acostumbra es hacer que todas las variables miembro sean privadas. Esto implica que sólo se podrá acceder a ellas, o modificarlas, usando las funciones miembro. Gran parte de este capítulo se dedica a explicar cómo y por qué conviene definir las clases de esta manera.

Clases y objetos

Una **clase** es un tipo cuyas variables son **objetos**. Estos pueden tener tanto variables miembro como funciones miembro. La sintaxis de una definición de clase es:

Sintaxis

```
class Nombre_de_Clase
{
public:
     Especificacion_de_Miembro_1
     Especificacion_de_Miembro_2
                  .
                  .
                  .
     Especificacion_de_Miembro_n
private:
     Especificacion_de_Miembro_n+1
     Especificacion_de_Miembro_n+2
                  .
                  .
                  .
};
```

Miembros públicos

Miembros privados

No olvide este punto y coma.

Cada *Especificacion_de_Miembro_i* es una declaración de variable miembro o bien una declaración de función miembro. (Se permiten secciones `public` y `private` adicionales.)

Ejemplo

```
class Bicicleta
{
public:
    char obtener_color();
    int numero_de_velocidades();
    void fijar(int las_velocidades, char el_color);
private:
    int velocidades;
    char color;
};
```

Una vez que definimos una clase, podemos declarar un `objeto` (que no es más que una variable del tipo de clase) de la misma forma que las variables de cualquier otro tipo. Por ejemplo, lo que sigue declara dos objetos de tipo `Bicicleta`:

```
Bicicleta mi_bici, tu_bici;
```

TIP DE PROGRAMACIÓN

Defina funciones de acceso y de mutación

Podemos utilizar el operador == para probar dos valores de un tipo simple y determinar si son iguales o no. Lamentablemente, el operador predefinido == no se aplica automáticamente a objetos. En el capítulo 8 explicaremos cómo hacer que el operador == se aplique a los objetos de las clases que definamos. Hasta entonces, no podremos usar el operador de igualdad == con objetos (ni con estructuras). Esto puede dar pie a algunas complicaciones. Al definir una clase, el estilo recomendado es hacer que todas las variables miembro sean privadas. Así pues, si queremos probar dos objetos para ver si representan el mismo valor, necesitamos alguna forma de acceder a los valores de las variables miembro (o algo equivalente a los valores de las variables miembro). Esto nos permitirá determinar igualdad probando los valores de cada par de variables miembro correspondientes. Para hacer esto en el cuadro 6.4 usamos las funciones miembro `obtener_mes` y `obtener_dia` en la instrucción *`if-else`*.

funciones de acceso

Las funciones miembro, como `obtener_mes` y `obtener_dia`, que proporcionan acceso a los valores de las variables miembro privadas se llaman **funciones de acceso**. Dadas las técnicas que hemos aprendido hasta ahora, es importante incluir siempre un conjunto completo de funciones de acceso en cada definición de clase, a fin de poder determinar la igualdad de objetos. Las funciones de acceso no tienen que devolver literalmente los valores de cada variable miembro, pero sí deben devolver algo equivalente a esos valores. En el capítulo 8 desarrollaremos un método más elegante para determinar la igualdad de dos objetos, pero aun después de aprender esa técnica seguirá siendo útil contar con funciones de acceso.

funciones de mutación

Las funciones miembro como `fijar` del cuadro 6.4 que le permiten cambiar los valores de las variables miembro privadas se llaman **funciones de mutación**. Es importante que siempre se incluya a las funciones de mutación con cada definición de clase para poder cambiar la información almacenada en un objeto.

Funciones de acceso y mutación

Las funciones miembro que le permiten encontrar los valores de las variables miembro privadas de una clase se denominan **funciones de acceso**. Éstas no necesitan regresar literalmente los valores de cada variable miembro, pero sí deben regresar algo equivalente a esos valores. Aunque en el lenguaje C++ esto no es requerido, los nombres de las funciones de acceso normalmente incluyen la palabra `get`.

Las funciones miembro que le permiten cambiar los valores de las variables miembro privadas de una clase se denominan funciones de mutación. Aunque en el lenguaje C++ esto no es requerido, los nombres de las funciones de mutación normalmente incluyen la palabra `set`.

Es importante que siempre se incluya a las funciones de acceso y mutación con cada definición de clase para poder cambiar la información almacenada en un objeto.

Ejercicios de AUTOEVALUACIÓN

14. La función miembro privada `DiaDelAnio::checar_fecha` del cuadro 6.4 permite que pasen algunas fechas no válidas como febrero 30. Redefina la función miembro `DiaDelAnio::checar_fecha` de manera que finalice el programa en donde encuentre una fecha no válida. Permita que febrero contenga 29 días para los años bisiestos. (*Sugerencia*: Esto es un poco tedioso y la definción de función un poco larga pero no es muy difícil.)

15. Supongamos que un programa contiene esta definición de clase:

```
class Automovil
{
public:
    void fijar_precio(double precio_nuevo);
    void fijar_margen(double nuevo_margen);
    double obtener_precio();
private:
    double precio;
    double margen;
    double obtener_margen();
};
```

y suponga que la parte `main` del programa contiene la siguiente declaración y que de alguna manera el programa asigna ciertos valores a todas las variables miembro:

```
Automovil hyundai, jaguar;
```

¿Cuáles de las siguientes instrucciones son válidas en la parte `main` del programa?

```
hyundai.precio = 4999.99;
jaguar.fijar_precio(30000.97);
double un_precio, un_margen;
un_precio = jaguar.obtener_precio();
un_margen = jaguar.obtener_margen();
un_margen = hyundai.obtener_margen();
if (hyundai == jaguar)
    cout << "¿Cambiamos de coche?";
hyundai = jaguar;
```

16. Suponga que modifica el ejercicio 15 de modo que la definición de la clase `Automovil` no incluya la línea que contiene la palabra clave `private`. ¿Cómo cambiaría esto su respuesta al ejercicio 15?

17. Explique qué hacen `public:` y `private:` en una definición de clase. En particular, explique por qué no hacemos que todo sea `public:` y nos ahorramos problemas en el acceso.

18. a) ¿Cuántas secciones `public:` debe tener una clase para ser útil?

    b) ¿Cuántas secciones `private:` debe tener una clase?

    c) ¿Qué tipo de sección hay entre la `{` inicial y la primera etiqueta de sección `public:` o `private:` de una clase?

    d) ¿Qué tipo de sección hay entre la `{` inicial y la primera etiqueta de sección `public:` o `private:` de una estructura?

TIP DE PROGRAMACIÓN

Emplee el operador de asignación con objetos

Es perfectamente válido emplear el operador de asignación = con objetos o con estructuras. Por ejemplo, supongamos que la clase `DiaDelAnio` se define como se muestra en el cuadro 6.4 de modo que tiene dos variables miembro privadas llamadas `mes` y `dia`, y supongamos que los objetos `vencimiento` y `manana` se declaran como sigue:

```
DiaDelAnio vencimiento, manana;
```

Entonces, lo que sigue es perfectamente válido (siempre que las variables miembro del objeto `manana` ya hayan recibido valores):

```
vencimiento = manana;
```

La asignación anterior equivale a lo siguiente:

```
vencimiento.mes = manana.mes;
vencimiento.dia = manana.dia;
```

Esto se cumple aunque las variables miembro `mes` y `dia` sean miembros privados de la clase `DiaDelAnio`.[3]

EJEMPLO DE PROGRAMACIÓN

Clase* `CuentaBancaria`*, versión 1

El cuadro 6.5 contiene una definición de clase para una cuenta bancaria que ilustra todos los puntos que hemos visto hasta ahora acerca de las definiciones de clase. Este tipo de cuenta bancaria permite retirar dinero en cualquier momento, de modo que no tiene un plazo como en el caso del tipo `CuentaCD` que vimos antes. Una diferencia más importante es el hecho de que la clase `CuentaBancaria` tiene funciones miembro para todas las operaciones que esperaríamos emplear en un programa. Los objetos de la clase `CuentaBancaria` tienen dos variables miembro privadas: una para registrar el saldo de la cuenta y una para registrar la tasa de interés. Analicemos algunas de las funciones de la clase `CuentaBancaria`.

función miembro privada

En primer lugar, observe que la clase `CuentaBancaria` tiene una función miembro privada llamada `fraccion`. Puesto que se trata de una función miembro privada, no se puede invocar en el cuerpo de `main` ni en el cuerpo de ninguna función que no sea función miembro de la clase `CuentaBancaria`. La función `fraccion` sólo puede invocarse en las definiciones de otras funciones miembro de la clase `CuentaBancaria`. La única razón por la que tenemos esta (o cualquier otra) función miembro privada es ayudarnos a definir otras funciones miembro para la misma clase. En nuestra definición de la clase `CuentaBancaria` incluimos la función miembro `fraccion` para poder usarla en la definición de la función `actualizar`. La función miembro `fraccion` recibe un argumento que es un porcentaje, digamos `10.0` para 10.0%, y la convierte en una fracción, en este caso `0.10`. Esto nos permite calcular el monto de los intereses de la cuenta con un porcentaje dado. Si la cuenta contiene $100.00 y la tasa de interés es del 10%, los intereses serán $100 $\times$ 0.10 = $10.00.

[3] En el capítulo 12 veremos situaciones en las que una clase debe sobrecargar el operador de asignación =, pero por ahora no debemos preocuparnos por ello.

CUADRO 6.5 *La* CuentaBancaria *(parte 1 de 4)*

```
//Programa para demostrar la clase CuentaBancaria.
#include<iostream>
using namespace std;

//Clase para una cuenta bancaria:
class CuentaBancaria
{
public:
    void fijar(int pesos, int centavos, double tasa);
    //Postcondición: El saldo de la cuenta se fijó en $pesos.centavos;
    //La tasa de interés se fijó en tasa porciento.

    void fijar(int pesos, double tasa);
    //Postcondición: El saldo de la cuenta se fijó en $pesos.00;
    //La tasa de interés se fijó en tasa porciento.

    void actualizar();
    //Postcondición: Se ha sumado un año de interés simple
    //al saldo de la cuenta.

    double obtener_saldo();
    //Devuelve el saldo actual de la cuenta.

    double obtener_tasa();
    //Devuelve la tasa de interés vigente como un porcentaje.

    void salida(ostream& sale);
    //Precondición: Si sale es un flujo de archivo de salida, ya
    //se ha conectado a un archivo.
    //Postcondición: El saldo de la cuenta y la tasa de interés se
    //han escrito en el flujo sale.
private:
    double saldo;
    double tasa_interes;

    double fraccion(double porciento);
    //Convierte un porcentaje en fracción. Por ejemplo, fraccion(50.3) devuelve 0.503.
};
int main()
{
    CuentaBancaria cuenta1, cuenta2;
    cout << "Inicio de la prueba:\n";
```

La función miembro `fijar` *está sobrecargada.*

CUADRO 6.5 *La* `CuentaBancaria` *(parte 2 de 4)*

```
    cuenta1.fijar(123, 99, 3.0);
    cout << "estado inicial de cuenta1:\n";
    cuenta1.salida(cout);

    cuenta1.fijar(100, 5.0);
    cout << "cuenta1 con nuevos valores:\n";
    cuenta1.salida(cout);

    cuenta1.actualizar();
    cout << "cuenta1 despues de actualizar:\n";
    cuenta1.salida(cout);

    cuenta2 = cuenta1;
    cout << "cuenta2:\n";
    cuenta2.salida(cout);
    return 0;
}

void CuentaBancaria::fijar(int pesos, int centavos, double tasa)
{
    if ((pesos < 0) || (centavos < 0) || (tasa < 0))
    {
      cout <<"Valores no validos para dinero o tasa de interes .\n";
      exit(1);
    }
    saldo = pesos + 0.01*centavos;
    tasa_interes = tasa;
}

void CuentaBancaria::fijar(int pesos, double tasa)
{
    if ((pesos < 0) || (tasa < 0))
    {
      cout <<"Valores no validos para dinero o tasa de interes .\n";
      exit(1);
    }
    saldo = pesos;
    tasa_interes = tasa;
}
```

Llamadas a la función miembro sobrecargada `fijar`.

Definiciones de la función miembro sobrecargada `fijar`.

CUADRO 6.5 *La* `CuentaBancaria` *(parte 3 de 4)*

```
void CuentaBancaria::actualizar()
{
    saldo = saldo + fraccion(tasa_interes)*saldo;
}

double CuentaBancaria::fraccion(double porciento)
{
    return (porciento/100.00);
}

double CuentaBancaria::obtener_saldo()
{
    return saldo;
}

double CuentaBancaria::obtener_tasa()
{
    return tasa_interes;
}

//Usa iostream:
void CuentaBancaria::salida(ostream& sale)
{
    sale.setf(ios::fixed);
    sale.setf(ios::showpoint);
    sale.precision(2);
    sale << "Saldo de la cuenta $" << saldo << endl;
    sale << "Tasa de interes " << tasa_interes << "%" << endl;
}
```

En la definición de una función miembro invocamos otra función miembro así.

Parámetro de flujo que se puede sustituir por `cout` *o por un flujo de archivo de salida.*

CUADRO 6.5 *La* `CuentaBancaria` *(parte 4 de 4)*

Diálogo de ejemplo

```
Inicio de la prueba:
Estado inicial de cuenta1:
Saldo de la cuenta $123.99
Tasa de interes 3.00%
cuenta1 con nuevos valores:
Saldo de la cuenta $100.00
Tasa de interes 5.00%
cuenta1 despues de actualizar:
Saldo de la cuenta $105.00
Tasa de interes 5.00%
cuenta2:
Saldo de la cuenta $105.00
Tasa de interes 5.00%
```

Cuando invocamos, en la parte `main` del programa, una función miembro pública, como `actualizar`, debemos incluir un nombre de objeto y un punto, como en la siguiente línea del cuadro 6.5:

```
cuenta1.actualizar();
```

una función miembro llama a otra

En cambio, cuando invocamos una función miembro privada (o cualesquier otras funciones miembro) dentro de la definición de otra función miembro, sólo usamos el nombre de la función, sin objeto invocador ni operador punto. Por ejemplo, la siguiente definición de la función miembro `CuentaBancaria::actualizar` incluye una llamada a `CuentaBancaria::fraccion` (como se muestra en el cuadro 6.5):

```
void CuentaBancaria::actualizar()
{
     saldo = saldo + fraccion(tasa_de_interes)*saldo;
}
```

El objeto invocador de la función miembro `fraccion` y de las variables miembro `saldo` y `tasa_de_interes` se determinan cuando se invoca la función `actualizar`. Por ejemplo, el significado de

```
cuenta1.actualizar();
```

es el siguiente:

```
{
    cuenta1.saldo = cuenta1.saldo +
        cuenta1.fraccion(cuenta1.tasa_de_interes)*cuenta1.saldo;
}
```

Observe que la llamada a la función miembro `fraccion` se maneja igual que las referencias a las variables miembro en este sentido.

argumentos que son flujos de entrada/salida

Al igual que las clases que vimos antes, la clase `CuentaBancaria` tiene una función miembro que envía a la salida los datos almacenados en el objeto. En este programa estamos enviando las salidas a la pantalla, pero quisimos escribir la definición de clase de modo que se pueda copiar y utilizar sin modificación en otros programas. Puesto que algún otro programa podría querer enviar salidas a un archivo, hemos dado a la función miembro `salida` un parámetro formal de tipo `ostream` para que la función `salida` se pueda invocar con un argumento que sea el flujo `cout` o bien un flujo de archivo de salida. En el programa de muestra queremos que las salidas vayan a la pantalla, así que la primera llamada a la función miembro `salida` tiene la forma

```
cuenta1.salida(cout);
```

Otras llamadas a `salida` también emplean `cout` como argumento, así que todas las salidas se envían a la pantalla. Si queremos que en vez de ello las salidas se envíen a un archivo, primero deberemos conectar el archivo a un flujo de salida como vimos en el capítulo 5. Si el flujo de archivo de salida se llama `fout` y está conectado a un archivo, lo que se muestra a continuación escribirá en ese archivo, no en la pantalla, los datos contenidos en el objeto `cuenta1`:

```
cuenta1.salida(fout);
```

sobrecarga de funciones miembro

El valor de un objeto de tipo `CuentaBancaria` representa una cuenta bancaria que tiene cierto saldo y paga cierta tasa de interés. El saldo y la tasa de interés se pueden establecer mediante la función miembro `fijar`. Observe que hemos sobrecargado la función miembro `fijar` de modo que haya dos versiones. Una versión tiene tres parámetros formales y la otra sólo tiene dos. Ambas versiones tienen un parámetro formal de tipo *`double`* para la tasa de interés, pero las dos versiones usan diferentes parámetros formales para fijar el saldo de la cuenta. Una versión tiene dos parámetros formales para hacerlo, uno para los pesos y el otro para los centavos del saldo de la cuenta. La otra versión sólo tiene un parámetro formal que da los pesos que hay en la cuenta y supone que el número de centavos es cero. Esta segunda versión de `fijar` es práctica porque casi todos abrimos una cuenta con una cantidad "cerrada" de dinero, digamos $1000 sin centavos. Cabe señalar que esta sobrecarga no es nada nuevo. Las funciones miembro se sobrecargan igual que las funciones ordinarias.

Resumen de algunas propiedades de las clases

Las clases tienen todas las propiedades que describimos para las estructuras y además todas las propiedades asociadas a las funciones miembro. A continuación se muestra una lista de lo que es conveniente tener presente al usar clases:

- Las clases tienen tanto variables miembro como funciones miembro.
- Un miembro (sea una variable miembro o una función miembro) puede ser público o privado.

- Normalmente, todas las variables miembro de una clase se etiquetan como miembros privados.
- Un miembro privado de una clase sólo se puede utilizar dentro de la definición de una función miembro de la misma clase.
- El nombre de una función miembro de una clase se puede sobrecargar igual que el nombre de una función ordinaria.
- Una clase puede usar otra clase como tipo de una variable miembro.
- Una función puede tener parámetros formales cuyos tipos sean clases. (Vea los ejercicios de autoevaluación 19 y 20.)
- Una función puede devolver un objeto; es decir, el tipo del valor devuelto por una función puede ser una clase. (Vea el ejercicio de autoevaluación 21.)

Comparación de estructuras y clases

Por lo general, las estructuras tienen sólo variables miembro públicas y no tienen funciones miembro. Sin embargo, en C++ una estructura puede tener variables miembro privadas y funciones miembro tanto públicas como privadas. Fuera de algunas diferencias de notación, una estructura C++ puede hacer lo mismo que una clase. Habiendo dicho esto y satisfecho los requisitos legales de "verdad en la publicidad", le recomendamos que se olvide de este detalle técnico respecto a las estructuras. Si toma en serio este detalle técnico y emplea las estructuras igual que utiliza las clases, tendrá dos nombres (con diferentes reglas de sintaxis) para el mismo concepto. En cambio, si usa las estructuras como las hemos descrito, tendrá una diferencia significativa entre las estructuras (tal como las usa) y las clases; y su forma de usarlas coincidirá con la de la mayoría de los programadores.

Ejercicios de AUTOEVALUACIÓN

19. Dé una definición de la función que tiene la siguiente declaración de función. En el cuadro 6.5 se define la clase `CuentaBancaria`.

```
double diferencia(CuentaBancaria cuenta1, CuentaBancaria cuenta2);
//Precondición: cuenta1 y cuenta2 han recibido valores (es decir,
//sus variables miembro han recibido valores).
//Devuelve el saldo de cuenta1 menos el saldo de cuenta2.
```

20. Dé una definición de la función que tiene la siguiente declaración de función. En el cuadro 6.5 se define la clase `CuentaBancaria`. (*Sugerencia*: Es fácil si se usa una función miembro.)

```
void double_actualizar(CuentaBancaria& la_cuenta);
//Precondición: la_cuenta ya tiene un valor (es decir,
//sus variables miembro ya han recibido valores).
//Postcondición: El saldo de la cuenta se ha modificado, sumándole
//los intereses correspondientes a dos años.
```

21. Dé una definición de la función que tiene la siguiente declaración de función. En el cuadro 6.5 se define la clase `CuentaBancaria`.

```
CuentaBancaria cuenta_nueva(CuentaBancaria cuenta_vieja);
//Precondición: cuenta_vieja ya tiene un valor (es decir,
//sus variables miembro ya han recibido valores).
//Devuelve el valor de una cuenta nueva que tiene un saldo de cero
//y la misma tasa de interés que cuenta_vieja.
```

Por ejemplo, después de definirse esta función, un programa podría contener lo siguiente:

```
CuentaBancaria cuenta3, cuenta4;
cuenta3.fijar(999, 99, 5.5);
cuenta4 = cuenta_nueva(cuenta3);
cuenta4.salida(cout);
```

Esto produciría las siguientes salidas:

```
Saldo de la cuenta $0.00
Tasa de interes 5.50%
```

Constructores para inicialización

Es común querer inicializar algunas de las variables miembro de un objeto, o todas, cuando se declara el objeto. Como veremos más adelante, hay otras acciones de inicialización que podríamos querer realizar, pero la inicialización de variables miembro es el tipo de inicialización más común. C++ cuenta con recursos especiales para tales inicializaciones. Al definir una clase podemos definir un tipo especial de función miembro llamada **constructor**. Un constructor es una función miembro que se invoca automáticamente cuando se declara un objeto de esa clase. Empleamos un constructor para inicializar los valores de algunas variables miembro, o de todas, y realizar cualquier otro tipo de inicialización que pudiera necesitarse. Definimos un constructor de la misma forma que definimos cualquier otra función miembro, excepto por dos puntos:

constructor

1. El constructor debe tener el mismo nombre que la clase. Por ejemplo, si la clase se llama `CuentaBancaria`, cualquier constructor para esta clase deberá llamarse `CuentaBancaria`.
2. Una definición de constructor no puede devolver un valor. Es más, no puede darse ningún tipo, ni siquiera *void*, al principio de la declaración de la función ni del encabezado de la función.

Por ejemplo, supongamos que queremos añadir un constructor para inicializar el saldo y la tasa de interés de los objetos del tipo `CuentaBancaria` que se muestra en el cuadro 6.5. La definición de clase debe ser como sigue. (Hemos omitido algunos de los comentarios para ahorrar espacio, pero conviene incluirlos.)

```
class CuentaBancaria
{
public:
    CuentaBancaria(int pesos, int centavos, double tasa);
    //Inicializa el saldo de la cuenta con $pesos.centavos y
    //la tasa de interés a tasa porciento.
```

```
    void fijar(int pesos, int centavos, double tasa);
    void fijar(int pesos, double tasa);
    void actualizar();

    double obtener_saldo();
    double obtener_tasa();
    void salida(ostream& sale);
private:
    double saldo;
    double tasa_de_interes;
    double fraccion(double porciento);
};
```

Observe que el constructor se llama `CuentaBancaria`, que es el nombre de la clase. Observe también que el prototipo del constructor `CuentaBancaria` no comienza con *`void`* ni con algún otro nombre de tipo. Por último, observe que el constructor se coloca en la sección pública de la definición de clase. Normalmente, hacemos que nuestros constructores sean funciones miembro públicas. Si nuestros constructores fueran miembros privados, no podríamos declarar objetos de este tipo de clase, y la clase no serviría de nada.

Con la clase `CuentaBancaria` redefinida, podemos declarar e inicializar dos objetos de tipo `CuentaBancaria` como sigue:

```
CuentaBancaria cuenta1(10, 50, 2.0), cuenta2(500, 0, 4.5);
```

Suponiendo que la definición del constructor realiza la acción de inicialización que prometimos, la declaración anterior declarará el objeto `cuenta1`, establecerá el valor de `cuenta1.saldo` a `10.50`, y establecerá el valor de `cuenta1.tasa_de_interes` a `2.0`. Así pues, el objeto `cuenta1` se inicializa de modo que represente una cuenta bancaria con un saldo de $10.50 y una tasa de interés del 2.0%. Así mismo, `cuenta2` se inicializa de modo que represente una cuenta bancaria con un saldo de $500.00 y una tasa de interés del 4.5%. Lo que sucede es que se declara el objeto `cuenta1` y luego se invoca el constructor `CuentaBancaria` con los tres argumentos `10`, `50` y `2.0`. Así mismo, se declara `cuenta2` y luego se invoca el constructor `CuentaBancaria` con los argumentos `500`, `0` y `4.5`. El resultado es conceptualmente equivalente a lo que sigue (aunque no se puede escribir así en C++):

```
CuentaBancaria cuenta1, cuenta2; //PROBLEMAS—PERO CORREGIBLES
cuenta1.CuentaBancaria(10, 50, 2.0); //ABSOLUTAMENTE NO VÁLIDO
cuenta2.CuentaBancaria(500, 0, 4.5); //ABSOLUTAMENTE NO VÁLIDO
```

Como indican los comentarios, no podemos colocar las tres líneas anteriores en un programa. La primera línea puede hacerse aceptable, pero las dos llamadas al constructor `CuentaBancaria` no están permitidas. Los constructores no se pueden invocar igual que las funciones miembro ordinarias. No obstante, es obvio lo que queremos que suceda cuando escribimos las tres líneas anteriores, y eso sucede automáticamente al declarar los objetos `cuenta1` y `cuenta2` como se muestra a continuación:

```
CuentaBancaria cuenta1(10, 50, 2.0), cuenta2(500, 0, 4.5);
```

La definición de un constructor se da de la misma manera que la de cualquier otra función miembro. Por ejemplo, si modificamos la definición de la clase `CuentaBancaria` añadiendo la siguiente definición del constructor:

```
CuentaBancaria::CuentaBancaria(int pesos, int centavos, double tasa)
{
    if ((pesos < 0) || (centavos < 0) || (tasa < 0 ))
    {
       cout << "Valores no válidos para dinero o tasa de interes.\n";
       exit(1);
    }
    saldo = pesos + 0.01*centavos;
    tasa_de_interes = tasa;
}
```

Puesto que la clase y la función constructora tienen el mismo nombre, el nombre `CuentaBancaria` aparece dos veces en el encabezado de la función; el nombre `CuentaBancaria` que está antes del operador de resolución de alcance `::` es el nombre de la clase, y el nombre `CuentaBancaria` que está después del operador es el nombre de la función constructora. Observe también que no se especifica tipo devuelto en el encabezado de la definición del constructor, ni siquiera el tipo *void*. Excepto por estos detalles, los constructores se definen de la misma manera que las funciones miembro ordinarias.

Podemos sobrecargar un nombre de constructor como `CuentaBancaria::CuentaBancaria`, igual que podemos sobrecargar cualquier otro nombre de función miembro, tal como hicimos con `CuentaBancaria::fijar` en el cuadro 6.5. De hecho, es común sobrecargar los constructores para poder inicializar los objetos de más de una manera. Por ejemplo, en el cuadro 6.6 redefinimos la clase `CuentaBancaria` de modo que tenga tres versiones de su constructor. Esta redefinición sobrecarga el nombre de constructor `CuentaBancaria` para que pueda tener tres argumentos (como acabamos de ver), dos argumentos, o ninguno.

Por ejemplo, supongamos que sólo damos dos argumentos cuando declaramos un objeto de tipo `CuentaBancaria`, como en el siguiente ejemplo:

```
CuentaBancaria cuenta1(100, 2.3);
```

Entonces el objeto `cuenta1` se inicializará de modo que represente una cuenta con un saldo de $100.00 y una tasa de interés del 2.3%.

Por otra parte, si no se dan argumentos, como en el siguiente ejemplo,

```
CuentaBancaria cuenta2;
```

el objeto se inicializará de modo que represente una cuenta con un saldo de $0.00 y una tasa de interés del 0.0%. Observe que cuando el constructor no tiene argumentos, no incluimos paréntesis en la declaración del objeto. Lo que sigue es incorrecto:

```
CuentaBancaria cuenta2(); //¡ERROR! ¡NO HAGA ESTO!
```

Hemos omitido la función miembro (sobrecargada) `fijar` en esta definición modificada de la clase `CuentaBancaria` (que se da en el cuadro 6.6). Si tenemos un buen conjunto de definiciones de constructores, ya no es necesario tener otras funciones miembro que fijen las variables miembro de la clase. Podemos usar el constructor sobrecargado

CUADRO 6.6 *Clase con constructores (parte 1 de 3)*

```
//Programa para demostrar la clase CuentaBancaria.
#include<iostream>
using namespace std;

//Clase para una cuenta bancaria:
class CuentaBancaria
{
public:
    CuentaBancaria(int pesos, int centavos, double tasa);
    //Inicializa el saldo de la cuenta con $pesos.centavos e
    //Inicializa la tasa de interés con tasa porciento.

    CuentaBancaria(int pesos, double tasa);
    //Inicializa el saldo de la cuenta con $pesos.00 e
    //inicializa la tasa de interés con tasa porciento.

    CuentaBancaria();
    //Inicializa el saldo con $0.00 y la tasa de interés con 0.0%.

    void actualizar();
    //Postcondición: Se ha sumado un año de interés simple
    //al saldo de la cuenta.

    double obtener_saldo();
    //Devuelve el saldo actual de la cuenta.

    double obtener_tasa();
    //Devuelve la tasa de interés vigente como un porcentaje.

    void salida(ostream& sale);
    //Precondición: Si sale es un flujo de archivo de salida, ya
    //se ha conectado a un archivo.
    //Postcondición: El saldo de la cuenta y la tasa de interés se
    //han escrito en el flujo sale.
private:
    double saldo;
    double tasa_interes;

    double fraccion(double porciento);
    //Convierte un porcentaje en fracción. P. ej., fraccion(50.3)
    //devuelve 0.503.
};

int main()
{
    CuentaBancaria cuenta1(100, 2.3), cuenta2;
```

Esta definición de CuentaBancaria es una versión mejorada de la clase `CuentaBancaria` *dada en el cuadro 6.5.*

constructor predeterminado

Esta declaración causa una llamada al constructor predeterminado. Observe que no hay paréntesis.

CUADRO 6.6 *Clase con constructores (parte 2 de 3)*

```
    cout << "cuenta1 se inicializo asi:\n";
    cuenta1.salida(cout);
    cout << "cuenta2 se inicializo asi:\n";
    cuenta2.salida(cout);

    cuenta1 = CuentaBancaria(999, 99, 5.5);
    cout << "cuenta1 se cambió a lo siguiente:\n";
    cuenta1.salida(cout);
    return 0;
}

CuentaBancaria::CuentaBancaria(int pesos, int centavos, double tasa)
{
    if ((pesos < 0) || (centavos < 0) || (tasa < 0))
    {
        cout << "Valores no validos para dinero o tasa de interes.\n";
        exit(1);
    }

    saldo = pesos + 0.01*centavos;
    tasa_interes = tasa;
}

CuentaBancaria::CuentaBancaria(int pesos, double tasa)
{
    if ((pesos < 0) || (tasa < 0))
    {
        cout << "Valores no validos para dinero o tasa de interes.\n";
        exit(1);
    }
    saldo = pesos;
    tasa_interes = tasa;
}

CuentaBancaria::CuentaBancaria() : saldo(0), tasa_interes (0.0)
{
    //Cuerpo intencionalmente vacío
}
```

Llamada explícita al constructor
`CuentaBancaria::CuentaBancaria`

<Las definiciones de las demás funciones miembro son iguales que en el cuadro 6.5.>

CUADRO 6.6 *Clase con constructores (parte 3 de 3)*

Salida

```
cuenta1 se inicializo asi:
Saldo de la cuenta $100.00
Tasa de interes 2.30%
cuenta2 se inicializo asi:
Saldo de la cuenta $0.00
Tasa de interes 0.00%
cuenta1 se cambio a lo siguiente:
Saldo de la cuenta $999.99
Tasa de interes 5.50%
```

`CuentaBancaria` de el cuadro 6.6 para lo mismo que usaríamos la función miembro sobrecargada `fijar` (que incluimos en la versión vieja de la clase que se muestra en el cuadro 6.5).

Constructor

Un **constructor** es una función miembro de una clase que tiene el mismo nombre que la clase. Los constructores se invocan automáticamente cuando se declaran objetos de la clase. Además, sirven para inicializar objetos. Un constructor debe tener el mismo nombre que la clase de la cual es miembro.

El constructor sin parámetros que se muestra en el cuadro 6.6 necesita una segunda revisión ya que contiene algo que no hemos visto anteriormente. Para referencia reproducimos la definición del constructor sin parámetros:

```
CuentaBancaria::CuentaBancaria() : saldo(0), tasa_de_interes(0.0)
{
    //Cuerpo intencionalmente vacío
}
```

sección de inicialización

El nuevo elemento, el cual se encuentra en la primera línea, es la parte que comienza con los dos puntos. Esta parte se llama **sección de inicialización**. Como se ve en el ejemplo, esta sección va después del paréntesis que cierra la lista de parámetro y antes de la llave que abre para el cuerpo de función. La sección de inicialización consiste de dos puntos seguidos por una lista de algunas variables miembro separadas por comas. Cada variable miembro es seguida por su valor de inicialización entre paréntesis. La defini-

ción de constructor es equivalente completamente a la siguiente forma de escribir la definición:

```
CuentaBancaria::CuentaBancaria()
{
     saldo = 0;
     tasa_de_interes = 0.0;
}
```

El cuerpo de función en una definición de constructor con una sección de inicialización, no necesariamente debe estar vacío. Por ejemplo, la siguiente definición de un constructor de 2 parámetros es equivalente a la que se encuentra en el cuadro 6.6:

```
CuentaBancaria::CuentaBancaria(int pesos, double tasa)
                              : saldo(pesos), tasa_de_interes(tasa)
{
     if ((pesos < 0) || (tasa < 0))
     {
          cout << "valores no válidos para dinero o tasa de
          interés.\n";
          exit(1);
     }
}
```

Observe que los valores de inicialización se pueden dar en los términos de los parámetros del constructor.

Sección de inicialización del constructor

Algunas o todas las variables miembro en una clase pueden inicializarse (opcionalmente) en la **sección de inicialización del constructor** de una definición del constructor. La sección de inicialización del constructor va después del paréntesis que cierra la lista de parámetro y antes de la llave que abre el cuerpo de función. La sección de inicialización consiste de dos puntos seguidos por una lista de algunas variables miembro separadas por comas. Cada variable miembro es seguida por su valor de inicialización entre paréntesis. El ejemplo que se muestra a continuación utiliza una sección de inicialización del constructor y es equivalente al constructor de tres parámetros que se encuentra en el cuadro 6.6.

Ejemplo

```
CuentaBancaria::CuentaBancaria(int pesos, int centavos, double tasa)
     : saldo(pesos + 0.01*centavos), tasa_de_interes(tasa)
{
    if ((pesos < 0) || (centavos < 0) || (tasa < 0))
    {
        cout << "Valores no válidos para dinero o tasa de interés.\n";
        exit(1);
    }
}
```

Observe que los valores de inicialización se pueden dar en los términos de los parámetros del constructor.

Cómo invocar un constructor

Los constructores se invocan automáticamente cuando se declara un objeto, pero es preciso dar los argumentos para el constructor al declarar el objeto. Además, se puede invocar explícitamente un constructor para crear un objeto nuevo para una variable de clase.

Sintaxis (de una declaración de objeto cuando tenemos constructores)

Nombre_de_Clase Nombre_de_Objeto (*Argumentos_del_Constructor*) ;

Ejemplo

```
CuentaBancaria cuenta1(100, 2.3);
```

Sintaxis (de una llamada explícita a un constructor)

Objeto = *Nombre_de_Constructor* (*Argumentos_del_Constructor*) ;

Ejemplo

```
cuenta1 = CuentaBancaria(200, 3.5);
```

El constructor debe tener el mismo nombre que la clase de la cual es miembro. Por ello, en las descripciones de sintaxis anteriores *Nombre_de_Clase* y *Nombre_de_Constructor* son el mismo identificador.

llamada explícita a constructor

Los constructores se invocan automáticamente cuando declaramos un objeto de ese tipo de clase, pero también pueden invocarse otra vez después de declarado el objeto. Esto nos permite dar nuevos valores cómodamente a todos los miembros de un objeto. Los detalles técnicos son los siguientes. La invocación del constructor crea un objeto anónimo con valores nuevos. Un objeto anónimo es un objeto que (todavía) no se nombra con una variable. El objeto anónimo se puede asignar al objeto nombrado (es decir, a una variable de clase). Por ejemplo, la siguiente es una llamada al constructor `CuentaBancaria` que crea un objeto anónimo con un saldo de `$999.99` y una tasa de interés del `5.5%`. Este objeto anónimo se asigna al objeto `cuenta1`, que a su vez representa una cuenta con un saldo de $999.99 y una tasa de interés del 5.5%:[4]

```
cuenta1 = CuentaBancaria(999, 99, 5.5);
```

(Como deja entrever la notación, un constructor se comporta como una función que devuelve un objeto de su tipo de clase.)

[4] Dado que una llamada a un constructor siempre crea un objeto nuevo y una llamada a una función miembro `fijar` cambia vagamente los valores de las variables miembro existentes; una manera más eficiente para cambiar los valores de la variable miembro sería una llamada a `fijar` en lugar de una llamada a constructor; por lo que por razones de eficiencia usted quizá desee tener ambas, tanto la función miembro fijar como los constructores en su definición de clase.

TIP DE PROGRAMACIÓN

Siempre incluya un constructor predeterminado

C++ no siempre genera un constructor predeterminado para las clases que usted defina. Si no incluimos un constructor, el compilador generará un constructor predeterminado que no hace nada. Este constructor se invocará si se declaran objetos de la clase. Si damos al menos una definición de constructor para la clase, el compilador de C++ no generará ningún otro constructor. Cada vez que declaremos un objeto de ese tipo, C++ buscará una definición de constructor apropiada para emplearla. Si declaramos un objeto sin dar argumentos para el constructor, C++ buscará un constructor predeterminado, y si no hemos definido un constructor predeterminado, el compilador no podrá encontrarlo.

Por ejemplo, supongamos que definimos una clase como sigue:

```
class ClaseMuestra
{
public:
     ClaseMuestra(int parametro1, double parametro2);
     void hacer_cosas();
private:
     int dato1;
     double dato2;
};
```

Constructor que requiere dos argumentos.

Seguramente reconocerá lo que sigue como una forma válida de declarar un objeto de tipo `ClaseMuestra` e invocar el constructor de esa clase:

```
ClaseMuestra mi_objeto(7, 7.77);
```

Sin embargo, tal vez se sorprenda al saber que lo siguiente no es válido:

```
ClaseMuestra tu_objeto;
```

El compilador interpreta lo anterior como una declaración que incluye una llamada a un constructor que no tiene argumentos; sin embargo, no existe una definición de un constructor con cero argumentos. Podemos hacer una de dos cosas: añadir dos argumentos a la declaración de `tu_objeto` o bien añadir una definición para un constructor que no recibe argumentos.

constructor predeterminado

Un constructor que se puede invocar sin argumentos es un **constructor predeterminado**, ya que se aplica en el caso predeterminado donde declara un objeto sin especificar argumentos. Dado que es probable que haya ocasiones en las que queramos declarar un objeto sin dar argumentos para el constructor, es recomendable incluir siempre un constructor predeterminado. La siguiente redefinición de `ClaseMuestra` incluye un constructor predeterminado:

```
class ClaseMuestra
{
public:
     ClaseMuestra(int parametro1, double parametro2);
     ClaseMuestra();
     void hacer_cosas();
```

Constructor predeterminado

```
private:
    int dato1;
    double dato2;
};
```

Si redefinimos la clase `ClaseMuestra` de esta manera, la declaración previa de `tu_objeto` será válida:

Si no queremos que el constructor predeterminado inicialice las variables miembro, podemos dejar el cuerpo vacío al implementar el constructor. La siguiente definición de constructor es perfectamente válida. No hace nada cuando se le invoca; sólo mantiene contento al compilador:

```
ClaseMuestra::ClaseMuestra()
{
    //No hacer nada.
}
```

RIESGO *Constructores sin argumentos*

Si un constructor de una clase llamada `CuentaBancaria` tiene dos parámetros formales, declaramos un objeto y damos los argumentos para el constructor de la siguiente manera:

```
CuentaBancaria cuenta1(100, 2.3);
```

Para invocar al constructor sin argumentos, sería natural pensar que podemos declarar el objeto así:

```
CuentaBancaria cuenta2(); //ESTO CAUSARÁ PROBLEMAS.
```

Después de todo, cuando invocamos una función que no tiene argumentos, siempre escribimos un par de paréntesis vacío. Sin embargo, en el caso de los constructores esto es incorrecto. Lo peor es que tal vez ni siquiera genere un mensaje de error, pues tiene un significado que no es el que pretendemos que tenga. El compilador pensará que lo anterior es la declaración de una función llamada `cuenta2` que no recibe argumentos y que devuelve un valor de tipo `CuentaBancaria`.

No debemos incluir paréntesis cuando declaramos un objeto y queremos que C++ use el constructor que no recibe argumentos. La forma correcta de declarar `cuenta2` usando el constructor sin argumentos es:

```
CuentaBancaria cuenta2;
```

Sin embargo, si invocamos explícitamente a un constructor en una instrucción de asignación, sí usamos los paréntesis. Si las definiciones y declaraciones son como en el cuadro 6.6, lo que sigue hará que el saldo de `cuenta1` sea $0.00 y su tasa de interés sea del 0.0%:

```
cuenta1 = CuentaBancaria();
```

Constructores sin argumentos

Cuando declaramos un objeto y queremos que se invoque un constructor con cero argumentos, no debemos incluir paréntesis. Por ejemplo, para declarar un objeto y pasar dos argumentos al constructor podríamos escribir:

```
CuentaBancaria cuenta1(100, 2.3);
```

Sin embargo, si queremos que se emplee el constructor con cero argumentos, declaramos el objeto de la siguiente manera:

```
CuentaBancaria cuenta1;
```

No declaramos el objeto así:

```
CuentaBancaria cuenta1(); //DECLARACIÓN INCORRECTA
```

(El problema es que esta sintaxis declara una función que devuelve un objeto `CuentaBancaria` y no tiene parámetros.)

Ejercicios de AUTOEVALUACIÓN

22. Suponga que un programa contiene la siguiente definición de clase (junto con definiciones de las funciones miembro):

```
class SuClase
{
public:
    SuClase(int info_nueva, char mas_info_nueva);
    SuClase();
    void hacer_cosas();
private:
    int informacion;
    char mas_informacion;
};
```

¿Cuáles de las siguientes instrucciones y declaraciones son válidas?

```
SuClase un_objeto(42, 'A');
SuClase otro_objeto;
SuClase un_objeto_mas();
un_objeto = SuClase(99, 'B')
un_objeto = SuClase();
un_objeto = SuClase;
```

23. ¿Cómo modificaría la definición de la clase `DiaDelAnio` del cuadro 6.4 de modo que tenga dos versiones de un constructor (sobrecargado)? Una versión deberá tener dos parámetros formales `int` (uno para el mes

y otro para el día) y establecerá las variables miembro privadas de modo que representen ese mes y ese día. El otro no tendrá parámetros formales y establecerá la fecha representada a 1o. de enero. Haga esto sin usar la sección de inicialización del constructor en ambos constructores.

24. Haga nuevamente el ejercicio anterior, pero esta vez para inicializar todas las funciones miembro en cada constructor utilice la sección de inicialización.

6.3 Tipos de datos abstractos

Todos lo sabemos —el Times lo sabe—
pero hacemos como si no lo supiéramos.

Virginia Woolf, *Monday or Tuesday*

Un tipo de datos, como el tipo `int`, tiene ciertos valores que se especifican, como `0`, `1`, `-1`, `2`, etcétera. Tendemos a pensar que el tipo de datos son estos valores, pero las operaciones con estos valores son tan importantes como los valores mismos. Sin dichas operaciones, no podríamos hacer nada interesante con los valores. Las operaciones para el tipo `int` son `+`, `-`, `*`, `/`, `%` y unos cuantos operadores y funciones de biblioteca predefinidas más. No debemos ver un tipo de datos como una simple colección de valores. Un **tipo de datos** consiste en una colección de valores junto con un conjunto de operaciones básicas definidas para esos valores.

tipos de datos

tipos de datos abstractos (ADT)

Decimos que un tipo de datos es un **ADT (tipo de datos abstracto)** si los programadores que emplean el tipo no tienen acceso a los detalles de implementación de los valores y las operaciones. Los tipos predefinidos, como `int`, son tipos de datos abstractos (ADTs). No sabemos cómo están implementadas las operaciones, digamos `+` o `*`, para el tipo `int`. Incluso si lo supiéramos, no utilizaríamos tal información en ningún programa en C++. Los tipos definidos por el programador, como los tipos de estructura y los tipos de clase, no son automáticamente ADTs. A menos que se definan y usen con cuidado, los tipos definidos por el programador se pueden emplear de formas no intuitivas que hacen al programa difícil de entender y modificar. La mejor manera de evitar estos problemas es asegurarnos de que todos los tipos de datos que definamos sean ADTs. La forma de hacer esto en C++ es usar clases, pero no toda clase es un ADT. Para convertir una clase en un ADT hay que definirla de cierta manera, y éste es el tema de la siguiente subsección.

Clases para producir tipos de datos abstractos

Una clase es un tipo que definimos, en contraposición a los tipos como `int` y `char` que ya están predefinidos. Un valor de un tipo de clase es el conjunto de valores de las variables miembro. Por ejemplo, un valor del tipo `CuentaBancaria` del cuadro 6.6 consiste en dos números de tipo `double`. Para facilitar la explicación, repetiremos la definición de la clase (omitiendo sólo los comentarios):

```
class CuentaBancaria
{
public:
    CuentaBancaria(int pesos, int centavos, double tasa);
    CuentaBancaria(int pesos, double tasa);
    CuentaBancaria();
    void actualizar();
```

```
    double obtener_saldo();
    double obtener_tasa();
    void salida(ostream& sale);
private:
    double saldo;
    double tasa_de_interes;
    double fraccion(double porciento);
};
```

El programador que utilice el tipo `CuentaBancaria` no necesita saber cómo implementamos la definición de `CuentaBancaria::actualizar` ni ninguna de las demás funciones miembro. La definición de la función miembro `CuentaBancaria::actualizar` que usamos es la siguiente:

```
void CuentaBancaria::actualizar()
{
    saldo = saldo + fraccion(tasa_de_interes)*saldo;
}
```

Sin embargo, podíamos habernos ahorrado la función privada `fraccion` implementando la función miembro `actualizar` con la siguiente fórmula, que es un poco más complicada:

```
void CuentaBancaria::actualizar()
{
   saldo = saldo + (tasa_de_interes/100.00)*saldo;
}
```

El programador que use la clase `CuentaBancaria` no debe tener que preocuparse por cuál implementación de `actualizar` hayamos usado, ya que ambas implementaciones tienen exactamente el mismo efecto.

De forma similar, el programador que utilice la clase `CuentaBancaria` no debe preocuparse por la forma de implementar los valores de la clase. Optamos por implementar los valores como dos valores de tipo *`double`*. Si `ahorro_vacacional` es un objeto de tipo `CuentaBancaria`, el valor de `ahorro_vacacional` consistirá en dos valores de tipo *`double`* almacenados en las dos variables miembro

```
ahorro_vacacional.saldo
ahorro_vacacional.tasa_de_interes
```

Sin embargo, no queremos pensar en el valor del objeto `ahorro_vacacional` como dos números de tipo *`double`*, digamos `1.3546e+2` y `4.5`. Queremos pensar en dicho valor como los datos

```
Saldo de la cuenta $135.46
Tasa de interés 4.50%,
```

Es por ello que nuestra implementación de `CuentaBancaria::salida` escribe con este formato el valor de la clase.

El hecho de que hayamos decidido implementar este valor `CuentaBancaria` como los dos valores *`double`* `1.3546e+2` y `4.5` es un detalle de implementación. También podríamos haber implementado este valor `CuentaBancaria` como los dos valores *`int`* `135` y `46`

(para las partes de pesos y centavos del saldo) y el valor `0.045` de tipo *`double`*. El valor `0.045` es simplemente el porcentaje 4.5% convertido en fracción, la cual podría ser una forma más útil de implementar una cifra de porcentaje. Después de todo, para calcular los intereses de la cuenta convertimos el porcentaje precisamente en una fracción así. Con esta otra implementación de la clase `CuentaBancaria`, los miembros públicos no cambiarían pero los miembros privados cambiarían a lo siguiente:

```
class CuentaBancaria
{
public:
     <Esta parte es exactamente igual que antes>
private:
     int parte_pesos;
     int parte_centavos;
     double tasa_de_interes;
     double fraccion(double porciento);
};
```

Necesitaríamos modificar las definiciones de las funciones miembro para que concuerden con este cambio, pero es fácil hacerlo. Por ejemplo, las definiciones de la función `obtener_saldo` y de una versión del constructor cambiarían a lo siguiente:

```
double CuentaBancaria::obtener_saldo()
{
    return (parte_pesos + 0.01*parte_centavos);
}
CuentaBancaria::CuentaBancaria(int pesos, int centavos, double tasa)
{
     if ((pesos < 0) || (centavos < 0) || (tasa < 0))
     {
          cout << "Valores no válidos por dinero o tasa de
          interés.\n";
          exit(1);
     }
     parte_pesos = pesos;
     parte_centavos = centavos;
     tasa_de_interes = tasa;
}
```

De forma similar, todas las demás funciones miembro se podrían redefinir a modo de contemplar esta nueva forma de almacenar el saldo de la cuenta y la tasa de interés.

Observe que aunque el usuario pueda pensar en el saldo de la cuenta como una sola cifra, eso no implica que la implementación tenga que ser un solo número de tipo *`double`*. Acabamos de ver que podría ser, por ejemplo, dos números de tipo *`int`*. El programador que emplea el tipo `CuentaBancaria` no necesita conocer estos detalles de la implementación de los valores del tipo `CuentaBancaria`.

cómo escribir un ADT

Estos comentarios acerca del tipo `CuentaBancaria` ilustran la técnica básica para definir una clase de modo que sea un tipo de datos abstracto. Si queremos definir una clase de modo que sea un tipo de datos abstracto, necesitamos separar la especificación de cómo utiliza un programador el tipo, de los detalles de cómo se implementa el tipo. La separación debe ser tan completa que podamos modificar la implementación de la clase sin que

ningún programa que use el ADT de clase requiera cambios adicionales. Una forma de asegurar esta separación es:

1. Hacer miembros privados de la clase a todas las variables miembro.
2. Hacer que todas las operaciones básicas que el programador necesita sean funciones miembro públicas de la clase, y especificar plenamente la forma de usar tales funciones miembro públicas.
3. Hacer que todas las funciones auxiliares sean funciones miembro privadas.

En los capítulos 8 y 9 veremos otras estrategias para definir ADTs, pero estas tres reglas son una forma común de asegurar que una clase sea un tipo de datos abstracto.

interfaz

La **interfaz** de un ADT nos dice cómo emplear el ADT en nuestros programas. Cuando definimos un ADT como una clase C++, la interfaz consiste en las funciones miembro públicas de la clase, junto con los comentarios que explican cómo usar dichas funciones miembro. La interfaz del ADT debe ser todo lo que se necesita saber para usar el ADT en un programa.

implementación

La **implementación** del ADT indica cómo se convierte la interfaz en código C++. La implementación del ADT consiste en los miembros privados de la clase y las definiciones de las funciones miembro tanto públicas como privadas. Aunque necesitamos la implementación para ejecutar un programa que usa el ADT, no será necesario que tenga algún conocimiento previo acerca de la implementación para escribir el resto de un programa que usa el ADT; es decir, no será necesario saber acerca de la implementación para escribir la parte `main` del programa ni para escribir cualesquier funciones no miembro que se usen en la parte `main`. Esta situación es similar a la que recomendamos para las definiciones de funciones ordinarias en los capítulos 3 y 4. La implementación de un ADT, al igual que la implementación de una función ordinaria, debe visualizarse como una caja negra cuyo interior no podemos ver.

interfaz e implementación separadas

En el capítulo 9 aprenderemos a colocar la interfaz y la implementación de un ADT en archivos individuales, separados unos de otros y de los programas que emplean el ADT. Así, un programador que emplee el ADT literalmente no verá la implementación. Hasta entonces, colocaremos todos los detalles de nuestras clases ADT en el mismo archivo que la parte `main` de nuestro programa, pero seguiremos considerando la inteface (dada en la sección pública de las definiciones de clases) y la implementación (la sección privada y las definiciones de funciones miembro) como partes separadas del ADT. Procuraremos escribir nuestros ADTs de modo que el usuario del ADT sólo necesite conocer la interfaz del ADT y nada acerca de la implementación. Para asegurarnos de estar definiendo nuestros ADTs de esta manera, basta con asegurarnos de que si modificamos la implementación de nuestro ADT el programa funcionará sin que sea necesario modificar ninguna otra de sus partes. Esto se ilustra en el siguiente ejemplo de programación.

El beneficio más obvio que obtenemos al hacer que nuestras clases sean tipos de datos abstractos es que podemos modificar la implementación sin tener que modificar ninguna otra parte del programa. Sin embargo, los ADTs ofrecen otros beneficios. Si hacemos que nuestras clases sean ADTs, podremos dividir el trabajo entre diferentes programadores: un programador diseñando y escribiendo el ADT y otros programadores uzándolo. Aunque sólo haya un programador trabajando en un proyecto, habrá logrado dividir una tarea mayor en dos tareas más pequeñas, y ello hará al programa más fácil de diseñar y de depurar.

EJEMPLO DE PROGRAMACIÓN
Otra implementación de una clase

variables miembro diferentes

El cuadro 6.7 contiene la implementación alternativa del ADT `CuentaBancaria` que vimos en la subsección anterior. En esta versión, los datos de una cuenta bancaria se implementan como tres valores miembro: uno para la parte de pesos del saldo de la cuenta, uno para la parte de centavos del saldo de la cuenta y uno para la tasa de interés.

Observe que, aunque tanto la implementación del cuadro 6.6 como la del cuadro 6.7 tienen una variable miembro llamada `tasa_de_interes`, el valor que se almacena es un poco distinto en las dos implementaciones. Si la cuenta paga intereses con una tasa del 4.7%, en la implementación del cuadro 6.6 (que es básicamente la misma del cuadro 6.5) el valor de `tasa_de_interes` sería `4.7`. En cambio, en la implementación del cuadro 6.7 el valor de `tasa_de_interes` sería `0.047`. Esta implementación alternativa (que se muestra en el cuadro 6.7) almacena la tasa de interés como una fracción, no como un porcentaje. La diferencia básica en esta nueva implementación es que cuando se fija una tasa de interés se usa la función `fraccion` para convertir de inmediato la tasa de interés en una fracción. Por lo tanto, en esta nueva implementación la función miembro privada `fraccion` se usa en las definiciones de constructores, pero no se necesita en la definición de la función miembro `actualizar` porque el valor que está en la variable miembro `tasa_de_interes` ya se convirtió en una fracción. En la antigua implementación (cuadros 6.5 y 6.6) la situación era la opuesta. En esa implementación la función miembro privada `fraccion` no se usaba en la definición de constructores, pero sí en la de `actualizar`.

La interfaz pública no cambia

Aunque hemos modificado los miembros privados de la clase `CuentaBancaria`, no hemos cambiado nada en la sección pública de la definición de la clase. Las funciones miembro públicas tienen exactamente las mismas declaraciones de funciones y se comportan exactamente igual que en la antigua versión del ADT dada en el cuadro 6.6. Por ejemplo, aunque esta nueva implementación almacena un porcentaje como 4.7% en forma de la fracción `0.047`, la función miembro `obtener_tasa` sigue devolviendo el valor `4.7`, igual que hacía en la antigua implementación del cuadro 6.5. De forma similar, la función miembro `obtener_saldo` devuelve un solo valor de tipo *`double`*, que proporciona el saldo como un número con punto decimal, igual que hacía en la antigua implementación del cuadro 6.5. Esto es cierto aunque ahora el saldo se almacena en dos variables miembro de tipo *`int`*, no en una sola variable miembro de tipo *`double`* (como en las otras versiones).

modificación de funciones miembro privadas

Cabe señalar que hay una diferencia importante entre la forma en que tratamos las funciones miembro públicas y la forma en que tratamos las funciones miembro privadas. Si queremos preservar la interfaz de un ADT de modo que ningún programa que lo use tenga que cambiar (fuera de modificar las definiciones de la clase y de sus funciones miembro), no deberemos modificar las declaraciones de las funciones miembro públicas. Sin embargo, estamos en libertad de añadir, eliminar o modificar cualquiera de las funciones miembro privadas. En este ejemplo hemos añadido una función privada llamada `porcentaje`, que es el inverso de la función `fraccion`. Esta última convierte un porcentaje en una fracción, y la función `porcentaje` convierte una fracción otra vez en un porcentaje. Por ejemplo, `fraccion(4.7)` devuelve `0.047` y `porcentaje(0.047)` devuelve `4.7`.

CUADRO 6.7 *Otra implementación de la clase* CuentaBancaria *(parte 1 de 3)*

```
//Demuestra otra implementación de la clase CuentaBancaria.
#include<iostream>
#include<cmath>
using namespace std;

//Clase para una cuenta bancaria:
class CuentaBancaria
{
public:
    CuentaBancaria(int pesos, int centavos, double tasa);
    //Inicializa el saldo de la cuenta con $pesos.centavos e
    //inicializa la tasa de interés con tasa porciento.

    CuentaBancaria(int pesos, double tasa);
    //Inicializa el saldo de la cuenta con $pesos.00 e
    //inicializa la tasa de interés con tasa porciento.

    CuentaBancaria();
    //Inicializa el saldo con $0.00 y la tasa de interés con 0.0%.

    void actualizar();
    //Postcondición: Se ha sumado un año de interés simple
    //al saldo de la cuenta.

    double obtener_saldo();
    //Devuelve el saldo actual de la cuenta.

    double obtener_tasa();
    //Devuelve la tasa de interés vigente como un porcentaje.

    void salida(ostream& sale);
    //Precondición: Si sale es un flujo de archivo de salida, ya
    //se ha conectado a un archivo.
    //Postcondición: El saldo de la cuenta y la tasa de interés se
    //han escrito en el flujo sale.
private:
    int parte_pesos;
    int parte_centavos;
    double tasa_interes; //expresada como fracción; p. ej., 0.057 para 5.7%

    double fraccion(double porciento);
    //Convierte un porcentaje en fracción; p. ej., fraccion(50.3)
    //devuelve 0.503.

    double porcentaje(double valor_fraccion);
    //Convierte una fracción en un porcentaje; p. ej., porcentaje(0.503)
    //devuelve 50.3.
};
```

Observe que los miembros públicos de CuentaBancaria *se ven y se comportan exactamente igual que en el cuadro 6.6.*

Nuevo

CUADRO 6.7 *Otra implementación de la clase* `CuentaBancaria` *(parte 2 de 3)*

```
int main()
{
    CuentaBancaria cuenta1(100, 2.3), cuenta2;

    cout << "cuenta1 se inicializo asi:\n";
    cuenta1.salida(cout);
    cout << "cuenta2 se inicializo asi:\n";
    cuenta2.salida(cout);

    cuenta1 = CuentaBancaria(999, 99, 5.5);
    cout << "cuenta1 se cambio a lo siguiente:\n";
    cuenta1.salida(cout);
    return 0;
}

CuentaBancaria::CuentaBancaria(int pesos, int centavos, double tasa)
{
    if ((pesos < 0) || (centavos < 0) || (tasa < 0))
    {
        cout<< "Valor ilegal para moneda o tasa de interes.\n";
        exit(1);
    }
    parte_pesos = pesos;
    parte_centavos = centavos;
    tasa_interes = fraccion(tasa);
}

CuentaBancaria::CuentaBancaria(int pesos, double tasa)
{
    if ((pesos < 0) || (tasa < 0))
    {
        cout<< "Valor ilegal para moneda o tasa de interes.\n";
        exit(1);
    }
    parte_pesos = pesos;
    parte_centavos = 0;
    tasa_interes = fraccion(tasa);
}
```

Puesto que el cuerpo de main es idéntico al del cuadro 6.6, las salidas a la pantalla también son idénticas a las del cuadro 6.6.

En la antigua implementación de este ADT, la función miembro privada `fraccion` *se usaba en la definición de* `actualizar`*. En esta implementación,* `fraccion` *se usa en la definición de constructores.*

CUADRO 6.7 *Otra implementación de la clase* CuentaBancaria *(parte 3 de 3)*

```
CuentaBancaria::CuentaBancaria() : parte_pesos (0), parte_centavos (0),
                                   tasa_interes (0.0)
{
    //Cuerpo intencionalmente vacío.
}

double CuentaBancaria::fraccion(double porciento)
{
    return (porciento/100.00);
}

//Usa cmath:
void CuentaBancaria::actualizar()
{
    double saldo = obtener_saldo();
    saldo = saldo + tasa_interes*saldo;
    parte_pesos = floor(saldo);
    parte_centavos = floor((saldo - parte_pesos)*100);
}

double CuentaBancaria::obtener_saldo()
{
    return (parte_pesos + 0.01*parte_centavos);
}

double CuentaBancaria::porcentaje(double valor_fraccion)
{
    return (valor_fraccion*100);
}

double CuentaBancaria::obtener_tasa()
{
    return porcentaje(tasa_interes);
}

//Usa iostream:
void CuentaBancaria::salida(ostream& sale)
{
    sale.setf(ios::fixed);
    sale.setf(ios::showpoint);
    sale.precision(2);
    sale << "Saldo de la cuenta $" << obtener_saldo() << endl;
    sale << "Tasa de interes " << obtener_tasa() << "%" << endl;
}
```

Las nuevas definiciones de `obtener_saldo` *y* `obtener_tasa` *aseguran que las salidas sigan estando en las unidades correctas.*

Ocultamiento de información

Vimos el ocultamiento de información cuando presentamos las funciones en el capítulo 3. Dijimos que el **ocultamiento de información**, aplicado a las funciones, implica que debemos escribir nuestras funciones de modo que se puedan usar como cajas negras, es decir, de modo que el programador que utiliza la función no necesite conocer los detalles de la implementación de la función. Este principio implica que todo lo que un programador que usa una función necesita conocer es la declaración de la función y el comentario que la acompaña y que explica cómo usar la función. El uso de variables miembro privadas y de funciones miembro privadas en la definición de un tipo de datos abstracto es otra forma de implementar el ocultamiento de información, pero ahora aplicamos el principio tanto a valores de datos como a funciones.

Ejercicios de AUTOEVALUACIÓN

25. Cuando definimos un ADT como una clase C++, ¿debemos hacer a las variables miembro públicas o privadas? ¿Debemos hacer a las funciones miembro públicas o privadas?
26. Cuando definimos un ADT como una clase C++, ¿qué elementos se consideran parte de la interfaz del ADT? ¿Qué elementos se consideran parte de la implementación del ADT?
27. Suponga que un amigo define un ADT como una clase C++ de la forma que describimos en la sección 6.3. Usted tiene que escribir un programa que use ese ADT. Es decir, debe escribir la parte `main` del programa y también cualesquier funciones no miembro que se usen en dicha parte. El ADT es muy largo y usted no tiene mucho tiempo para escribir el programa. ¿Qué partes del ADT necesita leer y de cuáles puede hacer caso omiso sin peligro?
28. Haga nuevamente los constructores de dos y tres parámetros en el cuadro 6.7 para que todas las variables miembro sean fijadas utilizando la sección de inicialización de constructor.

Resumen del capítulo

- Podemos usar una estructura para combinar datos de diferentes tipos en un solo valor (compuesto).
- Podemos emplear una clase para combinar datos y funciones en un solo objeto (compuesto).
- Una variable miembro o una función miembro de una clase puede ser pública o privada. Si es pública, se podrá utilizar fuera de la clase. Si es privada, sólo se podrá usar en la definición de otra función miembro.
- Una función puede tener parámetros formales de un tipo de clase o de estructura. Una función puede devolver valores de un tipo de clase o de estructura.
- Las funciones miembro de una clase se pueden sobrecargar igual que las funciones ordinarias.
- Un **constructor** es una función miembro de una clase que se invoca automáticamente cuando se declara un objeto de esa clase. El constructor debe tener el mismo nombre que la clase de la cual es miembro.

- Un **tipo de datos** consiste en una colección de valores junto con un conjunto de operaciones básicas definidas para esos valores.
- Un tipo de datos es un **tipo de datos abstracto** (**ADT**) si un programador que utiliza el tipo no necesita conocer ningún detalle de la implementación de los valores y operaciones de ese tipo.
- Una forma de implementar un tipo de datos abstracto en C++ es definir una clase con todas las variables miembro privadas y con las operaciones implementadas como funciones miembro públicas.

Respuestas a los ejercicios de autoevaluación

1 a) *double*

b) *double*

c) no válido —no podemos utilizar una etiqueta `struct` en lugar de una variable de estructura.

d) no válido —no se declaró `cuenta_ahorros`.

e) *char*

f) `CuentaCD`

2.
```
A $9.99
A $1.11
```

3. Muchos compiladores generan mensajes de error deficientes. Una sorpresa agradable es que el mensaje de error de `g++` es muy informativo.

```
g++ -fsyntax-only c6testg1.cc
prob1.cc:8: semicolon missing after declaration of
'Cosas'
prob1.cc:8: extraneous 'int' ignored
prob1.cc:8: semicolon missing after declaration of
'struct Cosas'
```

4. `A x = {1,2};`

5. a) Faltan inicializadores, no es error de sintaxis. Después de la inicialización, `vencimiento.dia==21`, `vencimiento.mes==12`, `vencimiento.anio==0`. Las variables miembro para las que no se proporciona inicializador se inicializan con un cero del tipo apropiado.

b) Correcto después de la inicialización. `21==vencimiento.dia`, `12==vencimiento.mes`, `2022==vencimiento.anio`.

c) Error: demasiados inicializadores.

d) Podría ser un error de diseño, es decir, un error con toda intención. El autor del código sólo proporciona dos dígitos para el inicializador de las fechas. Deben usarse cuatro dígitos para el año porque un programa que usa fechas con dos dígitos podría fallar de maneras que van desde divertidas hasta desastrosas al terminar el siglo.

6.
```
struct RegistroEmpleado
{
    double sueldo;
    int vacaciones;
    char tipo_sueldo;
};
```

7.
```
void leer_registro_zapato(TipoZapato& zapato_nuevo)
{
    cout << "Indique estilo de zapato (una letra): ";
    cin >> zapato_nuevo.estilo;
    cout << "Escriba el precio del zapato $";
    cin >> zapato_nuevo.precio;
}
```

8.
```
TipoZapato descuento(TipoZapato registro_viejo)
{
    TipoZapato temp;
    temp.estilo = registro_viejo.estilo;
    temp.precio = 0.90*registro_viejo.precio;
    return temp;
}
```

9.
```
struct RegistroExistencias
{
    TipoZapato info_zapato;
    Fecha fecha_llegada;
};
```

10.
```
RegistroExistencias aRegistro;
aRegistro.fecha_llegada.anio = 2004;
```

11.
```
void DiaDelAnio::entrada()
{
    cout << "Introduzca el mes como un numero: ";
    cin >> mes;
    cout << "Escriba el dia del mes: ";
    cin >> dia;
}
```

12.
```
void Temperatura::fijar(double nuevos_grados, char nueva_escala)
{
    grados = nuevos_grados;
    escala = nueva_escala;
}
```

13. Tanto el operador punto como el operador de resolución de alcance se emplean con nombres de miembros para especificar la clase o estructura de la cual es miembro el nombre de miembro. Si la clase `DiaDelAnio` se define como en el cuadro 6.3 y `hoy` es un objeto de la clase `DiaDelAnio`, podemos acceder al miembro `mes` con el operador punto: `hoy.mes`. Cuando damos la definición de una función miembro, utilizamos el operador de resolución de alcance para indicar al compilador que esta función es la que se declaró en la clase cuyo nombre se da antes del operador de resolución de alcance.

14. Si las expresiones booleanas en alguna de las instrucciones *if* no tienen sentido para usted, puede dejarlas por ahora y volver a verlas después de leer el capítulo 7.

```
void DiaDelAnio::checar_fecha()
{
    if ((mes < 1) || (mes > 12)
         || (dia < 1) || (dia > 31))
    {
        cout << "Fecha no valida. Abortando el programa.\n";
        exit(1);
    }
    if (((mes == 4) || (mes == 6) || (mes == 9) || (mes == 11))
       &&(dia == 31))
    {
       cout << "Fecha no valida. Abortando el programa.\n";
       exit(1);
    }
    if ((mes == 2) && (dia > 29))
    {
        cout << "Fecha no valida. Abortando el programa.\n";
        exit(1);
    }
}
```

15.
```
hyundai.precio = 4999.99; // NO VÁLIDO. precio es privada.
jaguar.fijar_precio(30000.97); //VÁLIDO
double un_precio, un_margen; //VÁLIDO
un_precio = jaguar.obtener_precio(); //VÁLIDO
un_margen = jaguar.obtener_margen(); //NO VÁLIDO. obtener_margen es privada.
un_margen = hyundai.obtener_margen(); //NO VÁLIDO. obtener_margen es privada.
if (hyundai == jaguar) //NO VÁLIDO. No se puede usar == con clases.
   cout << "Cambiamos de coche?";
hyundai = jaguar; //VÁLIDO
```

16. Después del cambio, todas serían válidas excepto la siguiente, que sigue siendo no válida:

```
if (hyundai == jaguar) //NO VÁLIDO. No se puede usar == con clases.
    cout << "Cambiamos de coche?";
```

17. *private* restringe el acceso a las definiciones de funciones a funciones miembro de la misma clase. Restringe la modificación de variables *private* a funciones proporcionadas por el autor de la clase. Así, el autor de la clase controla los cambios a los datos privados e impide alteraciones inadvertidas de los datos de la clase.

18. a) Sólo una. El compilador advierte si no hay miembros *public*: en una clase (o en una estructura).

    b) Ninguna; normalmente esperamos encontrar al menos una sección *private*: en una clase o estructura.

    c) Las clases son *private*: predeterminado; una sección así es una sección privada.

    d) Una estructura es *public*: predeterminado; una sección así es una sección pública.

19. Una respuesta correcta es:

```
double diferencia(CuentaBancaria cuenta1, CuentaBancaria cuenta2)
{
     return(cuenta1.obtener_saldo() − cuenta2.obtener_saldo());
}
```

Cabe señalar que lo que sigue *no es correcto*, porque `saldo` es un miembro privado:

```
double diferencia(CuentaBancaria cuenta1, CuentaBancaria cuenta2)
{
     return(cuenta1.saldo - cuenta2.saldo); //NO VÁLIDO
}
```

20.
```
void doble_actualizar(CuentaBancaria& la_cuenta)
{
    la_cuenta.actualizar();
    la_cuenta.actualizar();
}
```

Observe que como no se trata de una función miembro hay que escribir el nombre del objeto y el operador punto al invocar `actualizar`.

21.
```
CuentaBancaria cuenta_nueva(CuentaBancaria cuenta_vieja)
{
    CuentaBancaria temp;
    temp.fijar(0, cuenta_vieja.obtener_tasa());
    return temp;
}
```

22.
```
SuClase un_objeto(42, 'A'); //VÁLIDO
SuClase otro_objeto; //VÁLIDO
SuClase un_objeto_mas(); //PROBLEMA
un_objeto = SuClase(99, 'B') //VÁLIDO
un_objeto = SuClase(); //VÁLIDO
un_objeto = SuClase; //NO VÁLIDO
```

La instrucción marcada `//PROBLEMA` es válida en términos estrictos, pero no significa lo que podríamos creer que significa. Si nuestra intención es que esto sea una declaración de un objeto llamado `un_objeto_mas`, entonces es errónea. Es una declaración de función correcta de una función llamada `un_objeto_mas` que no recibe argumentos y que devuelve un valor de tipo `SuClase`, pero éste no es el significado que se quería. En la práctica, podríamos considerarlo no válido. La forma correcta de declarar un objeto llamado `un_objeto_mas`, de modo que se inicialice con el constructor predeterminado, es la siguiente:

```
SuClase un_objeto_mas;
```

23. La definición de clase modificada es la siguiente:

```
class DiaDelAnio
{
public:
    DiaDelAnio(int el_mes, int el_dia);
    //Precondición: el_mes y el_dia forman una fecha válida.
    //Inicializa la fecha según los argumentos.
```

```
    DiaDelAnio();
    //Inicializa la fecha al primero de enero.

    void entrada();

    void salida();

    int obtener_mes();
    //Devuelve el mes, 1 para enero, 2 para febrero, etcétera.

    int obtener_dia();
    //Devuelve el día del mes.
private:
    void checar_flecha
    int mes;
    int dia;
};
```

Observe que hemos omitido la función miembro `fijar`, ya que los constructores la hacen innecesaria. También es preciso añadir las siguientes definiciones de función (y eliminar la definición de función de `DiaDelAnio::fijar`):

```
DiaDelAnio::DiaDelAnio(int el_mes, int el_dia)
{
    mes = el_mes;
    dia = el_dia;
    checar_flecha();
}

DiaDelAnio::DiaDelAnio()
{
    mes = 1;
    dia = 1;
}
```

24. La definición de clase es la misma que en el ejercicio anterior. Las definiciones del constructor pueden cambiar a lo siguiente

```
DiaDelAnio::DiaDelAnio(int el_mes, int el_dia)
    : mes(el_mes), dia(el_dia)
{
    checar_fecha();
}

DiaDelAnio::DiaDelAnio() : mes(1), dia(1)
{
    //Cuerpo intencionalmente vacío.
}
```

25. Todas las variables miembro deben ser privadas. Las funciones miembro que forman parte de la interfaz del ADT (es decir, las funciones miembro que son operaciones para el ADT) deben ser públicas. También podemos tener funciones auxiliares que sólo se usen en las definiciones de otras funciones miembro. Tales funciones auxiliares deben ser privadas.

26. Todas las declaraciones de variables miembro privadas forman parte de la implementación. (No debe haber variables miembro públicas.) Todas las declaraciones de funciones miembro públicas de la clase (que se enumeran en las definiciones de clase), así como los comentarios explicativos de esas declaraciones de funciones, forman parte de la interfaz. Todas las declaraciones de funciones miembro privadas forman parte de la implementación. Todas las definiciones de funciones miembro (sean públicas o privadas) forman parte de la implementación.

27. Sólo es necesario leer las partes de la interfaz. Es decir, sólo hay que leer las declaraciones de funciones de los miembros públicos de la clase (que se enumeran en las definiciones de clase) y los comentarios explicativos de dichas declaraciones de funciones. No es necesario leer las declaraciones de las funciones miembro privadas, las declaraciones de las variables miembro privadas, las definiciones de las funciones miembro públicas ni las definiciones de las funciones miembro privadas.

28.
```
CuentaBancaria::CuentaBancaria(int pesos, int centavos,
                               double tasa) : parte_pesos(pesos),
                               parte_centavos(centavos), tasa_de_interes
                               (fraccion(tasa))
{
    if ((pesos < 0) || (centavos < 0 || (tasa < 0 ))
    {
        cout <<
           "Valores no validos para dinero o tasa de interes.\n";
        exit(1);
    }
}
CuentaBancaria::CuentaBancaria(int pesos, double tasa)
     : parte_pesos(pesos), parte_centavos(0),
                   tasa_de_interes(fraccion(tasa))
{
    if ((pesos < 0) || (tasa < 0 ))
    {
       cout <<
           "Valores no validos para dinero o tasa de interes.\n";
       exit(1);
    }
}
```

Proyectos de programación

1. Escriba un programa para calificar un grupo escolar con las siguientes políticas de calificación:

a) Hay dos cuestionarios, cada uno de los cuales se califica con base en 10 puntos.

b) Hay un examen parcial y un examen final, cada uno de los cuales se califica con base en 100 puntos.

c) El examen final representa el 50% de la calificación, el parcial, el 25% y los dos cuestionarios juntos, el 25% restante. (No olvide normalizar los puntajes de los exámenes: se deben convertir en porcentajes antes de promediarse.)

Cualquier calificación de 90 o más es una A, cualquier calificación de 80 o más (pero menor que 90) es una B, cualquier calificación de 70 o más (pero menor que 80) es una C, cualquier calificación de 60 o más (pero menor que 70) es una D y cualquier calificación menor que 60 es una F.

El programa leerá los puntajes del estudiante y desplegará su registro, que consiste en dos puntajes de cuestionario y dos de examen, así como el puntaje promedio numérico para todo el curso y la calificación

de letra final. Defina y use una estructura para el registro de estudiante. Si este proyecto se deja de tarea, pregunte a su profesor si la entrada/salida se debe efectuar con el teclado y la pantalla o con archivos. Si debe realizarse con archivos, pida a su profesor instrucciones para los nombres de los mismos.

2. Realice nuevamente la programación del proyecto 1 (o hágala por primera vez), pero esta vez haga que el estudiante registre el tipo de una clase en lugar del tipo de una estructura. La clase de registro de estudiante debe tener variables miembro para todos los datos de entrada descritos en el proyecto de programación 1 y una variable miembro para el puntaje promedio numérico para el curso completo y así también una variable miembro para la letra de la calificación final. Haga que todas las variables miembro sean privadas. Incluya funciones miembro para lo siguiente: funciones miembro para establecer cada una de las variables miembro a los valores dados como argumentos de función, funciones miembro para almacenar los datos de cada una de las variables miembro, una función `void` para calcular el puntaje promedio numérico en escala del curso completo, la función que calcule esto y que además establezca la variable miembro correspondiente, y una función `void` que calcule la letra de la calificación final y establezca la variable miembro correspondiente.

3. Redefina `CuentaCD` del cuadro 6.1 de modo que sea una clase en lugar de una estructura. Use las mismas variables miembro que en el cuadro 6.1 pero hágalas privadas. Incluya funciones miembro para lo siguiente: devolver el saldo inicial, devolver el saldo al vencimiento, devolver la tasa de interés y para devolver el plazo. Incluya un constructor que asigne a todas las variables miembro cualesquier valores que se especifiquen, y también un constructor predeterminado. Incluya además una función miembro de entrada con un parámetro formal de tipo `istream` y una función miembro de salida con un parámetro formal de tipo `ostream`. Incruste su definición de clase en un programa de prueba.

4. Redefina su clase `CuentaCD` del proyecto 3 de modo que tenga la misma interfaz pero una implementación distinta. La nueva implementación es en muchos sentidos similar a la segunda implementación de la clase `CuentaBancaria` dada en el cuadro 6.7. Su nueva implementación de la clase `CuentaCD` registrará el saldo como dos valores de tipo *`int`*, uno para los pesos y otro para los centavos. La variable miembro para la tasa de interés almacenará la tasa como una fracción en lugar de una cifra de porcentaje. Por ejemplo, una tasa de interés del 4.3% se almacenará como el valor `0.043` de tipo *`double`*. Almacene el plazo de la misma forma que en el cuadro 6.1.

5. 
Defina una clase para un tipo llamado `TipoContador`. Los objetos de este tipo sirven para contar cosas, por lo que registran una cuenta que es un número entero no negativo. Incluya un constructor predeterminado que establezca el contador en cero y un constructor con un argumento que establezca el contador con el valor especificado por su argumento. Incluya funciones miembro para incrementar y decrementar el contador en uno. Asegúrese de que ninguna función miembro permita que el valor del contador se vuelva negativo. Incluya también una función miembro que devuelva el valor actual de la cuenta y otra que envíe la cuenta a un flujo de salida. La función miembro para efectuar la salida tendrá un parámetro formal de tipo `ostream` para el flujo de salida que recibe las salidas. Incruste su definición de clase en un programa de prueba.

6. 
Defina una clase llamada `Mes` que sea un tipo de datos abstracto para un mes. Esta clase tendrá una variable miembro de tipo *`int`* para representar un mes (1 para enero, 2 para febrero, etcétera). Incluya las siguientes funciones miembro: un constructor para establecer el mes utilizando las primeras tres letras del nombre del mes como tres argumentos, un constructor para establecer el mes usando un entero como argumento (1 para enero, 2 para febrero, etcétera) un constructor predeterminado, una función de entrada que lea el mes como un entero, una función de entrada que lea el mes como las primeras tres letras del nombre del mes, una función de salida que envíe a la salida el mes como un entero, una función de salida que envíe a la salida el mes como las primeras tres letras del nombre del mes y una función miembro que devuelva el siguiente mes como un valor de tipo `Mes`. Las funciones de entrada y salida tendrán cada una un parámetro formal para el flujo. Incruste su definición de clase en un programa de prueba.

7. Redefina la implementación de la clase `Mes` que se describe en el proyecto 6 (o defínala por primera vez, pero siga la implementación que se describe aquí). Esta vez el mes se implementa como tres variables miembro de tipo *`char`* que almacenan las tres primeras letras del nombre del mes. Incruste su definición en un programa de prueba.

8. (Para hacer este proyecto deberá haber efectuado antes el proyecto 6 o 7.) Reescriba el programa del cuadro 6.4, pero use la clase `Mes` que definió en el proyecto 6 o el proyecto 7 como tipo de la variable miembro que registra el mes. (Puede definir la clase `Mes` ya sea como se describe en el proyecto 6 o 7.) Redefina la función miembro `salida` de modo que tenga un parámetro formal de tipo `ostream` para el flujo de salida. Modifique también el programa de modo que todo lo que se envíe a la pantalla se envíe *también* a un archivo. Esto implica que todas las instrucciones de salida se repetirán dos veces: una con el argumento `cout` y otra con un flujo de archivo de salida como argumento. Si está en un curso, pida el nombre de archivo a su profesor. Las entradas se obtendrán del teclado; sólo las salidas se enviarán a un archivo.

9. Mi madre siempre llevaba un pequeño contador rojo a la tienda de abarrotes. El contador servía para llevar la cuenta de la cantidad de dinero que habría gastado hasta un momento dado de su visita a la tienda si comprara todo lo que ponía en su canasta. El contador podía desplegar hasta cuatro dígitos y tenía botones para incrementar cada dígito, así como un botón para restablecer el contador. Había un indicador de desbordamiento que se ponía rojo si se introducía más dinero que los $99.99 dólares que podía registrar. (Esto fue hace mucho tiempo.)

   Escriba e implemente las funciones miembro de una clase `Contador` que simule y generalice un poco el comportamiento del contador para tienda de abarrotes descrito. El constructor deberá crear un objeto `Contador` que pueda contar hasta el argumento del constructor. Es decir, `Contador(9999)` deberá crear un contador capaz de contar hasta `9999`. Un contador recién construido muestra una lectura de `0`. La función miembro *`void`* `restablecer();` pone en cero el número del contador. Las funciones miembro `void incr1();`, *`void`* `incr10();`, *`void`* `incr100();` y *`void`* `incr1000();`, incrementan en `1` los dígitos de las unidades, decenas, centenas y millares, respectivamente. Incluir cualquier acarreo al incrementar no deberá requerir más acción que sumar un número apropiado al miembro de datos privado. Una función miembro *`bool`* `desborde();` detecta desbordamientos. (El desbordamiento es el resultado de incrementar el miembro de datos privado del contador más allá del máximo especificado al construirse el contador.)

   Use esta clase para simular el aparatito rojo de mi madre. Aunque el número que se despliega es un entero, en la simulación los dos dígitos de la derecha (de orden más bajo) siempre se consideran como centavos y decenas de centavos, el siguiente dígito es pesos y el dígito de la extrema izquierda es decenas de pesos.

   Asigne teclas para centavos, decenas de centavos, pesos y decenas de pesos. Lamentablemente, no hay un conjunto de teclas que resulte especialmente mnemónico. Una opción es usar una de las teclas `asdfo`: `a` para centavos, seguida de un dígito del `1` al `9`; `s` para decenas de centavos, seguida de un dígito del `1` al `9`; `d` para pesos, seguida de un dígito del `1` al `9`; y `f` para decenas de pesos, también seguida de un dígito del `1` al `9`. Después de cada entrada (`a`, `s`, `d` o `f`, seguida de del `1` al `9`) se oprime la tecla Intro. Si hay desbordamiento, éste se informa después de cada operación. Se puede solicitar el desbordamiento oprimiendo la tecla `o`.

10. Escriba una clase de números racionales. (Volveremos a este problema en el capítulo 8 donde la sobrecarga de operadores hará al problema mucho más interesante. Por ahora usaremos funciones miembro `suma`, `resta`, `mul`, `div`, `menor` y `copiar` que realizan las operaciones +, −, *, /, < y =.) Por ejemplo, `a + b` será escrito `a .mas(b)`, y `a < b` se escribirá `a.menor(b)`.

    Defina una clase para números racionales. Un número racional es un "quebrado", formado por dos enteros y una indicación de división. La división no se efectúa, sólo se indica, como en `1/2`, `2/3`, `15/32`, `65/4`, `16/5`. Los números racionales se representarán con dos valores *`int`*, numerador y denominador.

Un principio de la construcción de tipos de datos abstractos es que debe haber constructores para crear objetos con cualesquier valores válidos. Usted deberá incluir constructores para crear objetos con pares de valores *int*; éste es un constructor con dos parámetros *int*. Puesto que todo *int* también es un racional, como en 2/1 o 17/1, se debe proporcionar un constructor con un solo parámetro *int*.

Proporcione funciones miembro entrada y salida que reciban un argumento istream y ostream, respectivamente, y obtengan números racionales en la forma 2/3 o 37/51 del teclado (o de un archivo).

Proporcione funciones suma, resta, mul y div que devuelvan un valor racional. Proporcione una función menor que devuelva un valor *bool*. Estas funciones deberán efectuar la operación que su nombre sugiere. Proporcione una función neg que no tenga un solo parámetro y devuelva el negativo del objeto invocador.

Proporcione una función main que pruebe exhaustivamente la implementación de su clase. Las siguientes fórmulas serán útiles para definir funciones:

```
a/b + c/d = (a * d + b * c) / (b * d)
a/b − c/d = (a * d − b * c) / (b * d)
(a/b) * (c/d) = (a * c) / (b * d)
(a/b) / (c/d) = (a * d) / (c * b)
−(a/b) = (−a/b)
(a/b) < (c/d) significa (a * d) < (c * b)
(a/b) == (c/d) significa (a * d) == (c * b)
```

El signo lo lleva el numerador; el denominador siempre es positivo.

7 Más flujo de control

7 Más flujo de control

Cuando llegues a un lugar en el que el camino se divide, síguelo.

ATRIBUIDO A YOGI BERRA

Introducción

El orden en que se ejecutan las instrucciones de un programa se llama flujo de control. Las instrucciones `if-else`, `while` y `do-while` son tres formas de especificar flujo de control. En este capítulo exploraremos algunas formas nuevas de usar estas instrucciones y presentaremos dos de ellas, `switch` y `for`, que también sirven para flujo de control. Las acciones de una instrucción `if-else`, `while` o `do-while` se controlan con expresiones booleanas. Comenzaremos por estudiar las expresiones booleanas con mayor detalle.

Prerrequisitos

Este capítulo utiliza material de los capítulos 2 al 5. No utiliza ningún material del capítulo 6.

7.1 Uso de expresiones booleanas

"Por otra parte", continuó Tweedledee.
"Si fuera así, podría ser;
y si así fuera, sería;
pero como no es así, no es. Eso es lógica."
Lewis Carroll, *Alicia a través del espejo*

Evaluación de expresiones booleanas

expresión booleana

Una **expresión booleana** es una expresión que puede ser *verdadera* o *falsa* (es decir, es *verdadera* si se satisface o *falsa* si no se satisface). Hasta ahora hemos usado expresiones booleanas como condición de prueba en instrucciones `if-else` y como expresión controladora en ciclos, como en los ciclos `while`. Sin embargo, una expresión booleana tiene una identidad independiente, fuera de cualquier instrucción `if-else` o instrucción cíclica en la que pudiéramos usarla. El tipo `bool` de C++ nos permite declarar variables que pueden tener los valores `true` (verdadero) y `false` (falso).

`bool`

Las expresiones booleanas se pueden evaluar igual que las expresiones aritméticas. La única diferencia es que en una expresión aritmética usamos operaciones como +, * y /, y la expresión produce un número como resultado final, en tanto que una expresión booleana usa operaciones de relación como == y <, y operaciones booleanas como &&, || y !, y produce uno de los dos valores `true` y `false` como resultado final. Cabe señalar que ==, !=, <, <=, etc., operan con pares de cualquier tipo predefinido para producir un valor booleano `true` o `false`. Si entendemos la forma en que se evalúan las expresiones booleanas, podremos escribir y entender algunas más complejas y emplear dichas expresiones como valor devuelto por una función.

Primero repasemos la evaluación de una expresión aritmética. Se usa la misma técnica de la misma manera para evaluar expresiones booleanas. Consideremos la siguiente expresión aritmética:

```
(x + 1) * (x + 3)
```

Supongamos que la variable `x` tiene el valor `2`. Para evaluar esta expresión aritmética, evaluamos las dos sumas para obtener los números `3` y `5`, y luego combinamos estos dos números usando el operador `*` para obtener `15` como valor final. Observe que, al realizar esta evaluación, no multiplicamos las expresiones `(x + 1)` y `(x + 3)`. En vez de ello, multiplicamos los valores de estas expresiones. Usamos `3`; no usamos `(x + 1)`. Usamos `5`; no usamos `(x + 3)`.

La computadora evalúa las expresiones booleanas de la misma manera. Se evalúan subexpresiones para obtener valores, cada uno de los cuales es `true` o `false`. Estos valores individuales de `true` o `false` se combinan entonces según las reglas de las tablas que se muestran en el cuadro 7.1. Por ejemplo, consideremos la expresión booleana

```
!( ( y < 3) || (y > 7) )
```

que podría ser la expresión controladora de una instrucción `if_else` o `while`. Supongamos que el valor de `y` es `8`. En este caso, la evaluación de `(y < 3)` da `false` y la de `(y > 7)` da `true`, por lo que la expresión booleana anterior equivale a

```
!( false || true )
```

Al consultar las tablas para || (rotulado **OR** en el cuadro 7.1), la computadora ve que la evaluación de la expresión dentro de los paréntesis da *true*. Por tanto, para la computadora toda la expresión equivale a

```
!(true)
```

Al consultar otra vez las tablas, la computadora ve que la evaluación de !(*true*) da *false*, por lo que concluye que el valor de la expresión booleana original es *false*.

paréntesis

Casi todos los ejemplos que hemos construido hasta ahora han incluido paréntesis para mostrar exactamente a qué expresiones aplica cada &&, || y !. Esto no siempre es obligatorio. Si omitimos paréntesis, la precedencia predeterminada es la siguiente: primero se efectúa !, luego los operadores relacionales como <, luego && y por último ||. Sin embargo, es recomendable incluir paréntesis que hagan a la expresión más fácil de entender. Un lugar en el que podemos omitir los paréntesis sin peligro es en una serie sencilla de operadores && o || (pero no mezclas de los dos). La expresión que sigue es aceptable tanto para el compilador de C++ como en términos de legibilidad:

```
(temperatura > 90) && (humedad > 0.90) && (puerta_alberca == ABIERTA)
```

CUADRO 7.1 *Tablas de verdad*

AND

Exp_1	Exp_2	Exp_1 && Exp_2
true	*true*	*true*
true	*false*	*false*
false	*true*	*false*
false	*false*	*false*

OR

Exp_1	Exp_2	Exp_1 || Exp_2
true	*true*	*true*
true	*false*	*true*
false	*true*	*true*
false	*false*	*false*

NOT

Exp	!(Exp)
true	*false*
false	*true*

Puesto que los operadores relacionales > y == se aplican antes que el operador &&, podríamos omitir los paréntesis en la expresión anterior y tendría el mismo significado, pero la inclusión de paréntesis hace a la expresión más fácil de entender.

reglas de precedencia

Cuando omitimos los paréntesis de una expresión, la computadora agrupa los elementos siguiendo reglas llamadas **reglas de precedencia**. En el cuadro 7.2 se muestran algunas de las reglas de precedencia para C++. Si un operador se aplica antes que otro, decimos que el operador que se aplica primero tiene **precedencia más alta**. Los operadores binarios que tienen la misma precedencia se aplican en orden de izquierda a derecha. Los operadores unarios que tienen la misma precedencia se aplican en orden de derecha a izquierda. En el Apéndice 2 se da un conjunto completo de reglas de precedencia.

CUADRO 7.2 *Reglas de precedencia*

Los operadores unarios: +, −, ++, -- y !

Los operadores aritméticos binarios: *, /, %

Los operadores aritméticos binarios: +, −

Los operadores booleanos: <, >, <=, >=

Los operadores booleanos: ==, !=

El operador booleano: &&

El operador booleano: ||

Más alta precedencia (se efectúan primero)

Más baja precedencia (se efectúan al último)

Observe que las reglas de precedencia incluyen tanto operadores aritméticos como + y *, así como operadores booleanos && y ||. La razón es que en muchas expresiones se combinan operaciones aritméticas y booleanas, como en este ejemplo sencillo:

```
(x + 1) > 2 || (x + 1) < −3
```

Si revisamos las reglas de precedencia del cuadro 7.2, veremos que esta expresión equivale a:

```
((x + 1) > 2) || ((x + 1) < −3)
```

porque > y < tienen mayor precedencia que ||. De hecho, podríamos omitir todos los paréntesis en la expresión anterior y tendría el mismo significado, aunque sería más difícil de entender. Si bien no recomendamos omitir todos los paréntesis, podría ser interesante ver cómo se interpreta una expresión semejante usando las reglas de precedencia. He aquí la expresión sin paréntesis:

```
x + 1 > 2 || x + 1 < −3
```

Las reglas de precedencia dicen que primero aplicamos el operador − unario y luego los +, luego aplicamos el > y el <, y por último aplicamos el ||, que es exactamente lo que la expresión con todos los paréntesis dice que hagamos.

La descripción anterior de la evaluación de una expresión booleana es básicamente correcta, pero en C++ la computadora a veces toma atajos al evaluar una expresión booleana. Observe que en muchos casos sólo necesitamos evaluar la primera de dos subexpresiones de una expresión booleana. Por ejemplo, consideremos lo siguiente:

```
(x >= 0) && (y > 1)
```

Si x es negativo, entonces (x >= 0) es *false* y, como podemos ver en las tablas del cuadro 7.1, cuando una subexpresión de una expresión && es *false*, toda la expresión es false, sin importar si la otra subexpresión es *true* o *false*. Así pues, si sabemos que la primera subexpresión es *false*, no habrá necesidad de evaluar la segunda subexpresión. Ocurre algo similar con las expresiones ||. Si la primera de dos subexpresiones enlazadas por el operador || es *true*, sabemos que toda la expresión es *true*, sin importar si la segunda expresión es *true* o *false*. El lenguaje C++ usa este hecho para ahorrarse ocasionalmente el trabajo de evaluar la segunda subexpresión de una expresión lógica conectada mediante un && o un ||. C++ evalúa primero la subexpresión de la izquierda. Si ello le proporciona suficiente información para determinar el valor final de la expresión (independiente del valor de la segunda subexpresión), C++ no se molestará en evaluar la segunda subexpresión. Este método de evaluación se conoce como **evaluación en cortocircuito**.

evaluación en cortocircuito
evaluación completa

Otros lenguajes usan **evaluación completa**. En ésta, cuando dos subexpresiones están unidas por un && o un ||, siempre se evalúan ambas subexpresiones y luego se usan las tablas de verdad para obtener el valor de la expresión final.

Tanto la evaluación en cortocircuito como la completa dan la misma respuesta, así que ¿por qué ha de importarnos que C++ use evaluación en cortocircuito? En casi todos los casos no debe importarnos. En tanto ambas subexpresiones unidas por el && o el || tengan un valor, los dos métodos darán el mismo resultado. Sin embargo, si la segunda subexpresión no está definida, nos alegrará saber que C++ usa evaluación en cortocircuito. Veamos un ejemplo que ilustra este punto. Consideremos la instrucción

```
if ( (chicos != 0) && ((dulces/chicos) >= 2) )
cout << "¡Cada niño puede tomar dos dulces!";
```

Si el valor de chicos no es cero, eata instrucción no tiene problemas. Sin embargo, supongamos que el valor de chicos es cero y consideremos cómo la evaluación en cortocircuito maneja este caso. La evaluación de la expresión (chicos != 0) da *false*, así que no hay necesidad de evaluar la segunda expresión. Al usar evaluación en cortocircuito, C++ dice que toda la expresión es *false*, *sin molestarse en evaluar la segunda expresión*. Esto evita un error de tiempo de ejecución, ya que la evaluación de la segunda expresión implicaría dividir entre cero.

A veces C++ usa enteros como si fueran valores booleanos. En particular, C++ convierte el entero 1 en *true* y convierte el entero 0 en *false*. La situación es un poco más complicada que simplemente emplear 1 para *true* y 0 para *false*. El compilador trata cualquier número distinto de cero como si fuera el valor *true* y trata a 0 como si fuera el valor *false*. En tanto no cometamos errores al escribir expresiones booleanas, esta conversión no causará problemas. En tales casos, ni siquiera tenemos que ser conscientes del hecho de que *true* y *false* se podrían convertir en 1 y 0 o viceversa. Sin embargo, cuando estamos depurando podría ser útil saber que el compilador no tiene prejuicios para combinar enteros

Los valores booleanos (`bool`) son `true` y `false`

En C++, la evaluación de una expresión booleana produce el valor `bool true` cuando se satisface y el valor `bool false` cuando no se satisface.

RIESGO *Las expresiones booleanas se convierten en valores* `int`

Supongamos que queremos usar una expresión booleana en una instrucción `if-else`, y queremos que sea `true` si todavía no se agota el tiempo (en algún juego o proceso). En términos un poco más precisos, supongamos que queremos una expresión booleana para usarla en una instrucción `if-else` y queremos que sea `true` si el valor de una variable `tiempo` de tipo `int` no es mayor que el valor de una variable llamada `limite`. Podríamos escribir lo siguiente (donde *Algo* y *Algo_Más* son `instrucciones en C++`):

```
if (!tiempo > limite)      <-- No apropiado para lo que queremos
    Algo
else
    Algo_Mas
```

Esto suena bien si lo leemos en voz alta: "si no `tiempo` mayor que `limite`". No obstante, la expresión booleana es errónea, y lo malo es que el compilador no generará un mensaje de error. Hemos sido víctimas de las reglas de precedencia de C++. El compilador aplicará las reglas de precedencia del cuadro 7.2 e interpretará la expresión booleana de la siguiente manera:

```
(!tiempo) > limite
```

Esto parece absurdo, e intuitivamente lo es. Si el valor de `tiempo` es, digamos, `36`, ¿cuál podría ser el significado de `(!tiempo)`? Después de todo, esto equivale a "no 36". Sin embargo, en C++ cualquier entero distinto de cero se convierte en `true` y cero se convierte en `false`. Así pues, `!36` se interpreta como "*no* `true`" y por lo tanto su evaluación da `false`, que a su vez se convierte en `0` porque lo estamos comparando con un `int`.

Lo que queremos como valor de esta expresión booleana y lo que C++ nos da no son la misma cosa. Si `tiempo` tiene un valor de `36` y `limite` tiene un valor de `60`, necesitamos que la evaluación de la expresión booleana dé `true` (porque *no es verdad* que `tiempo > limite`). Lamentablemente, la expresión booleana se evalúa como sigue: la evaluación de `(!tiempo)` da `false`, lo que se convierte en `0`, así que toda la expresión booleana equivale a

```
0 > limite
```

y eso a su vez equivale a `0 > 60`, ya que `60` es el valor de `limite`, y la evaluación de eso da *`false`*. Por tanto, la evaluación de la expresión lógica anterior da *`false`*, cuando lo que queremos es que dé `true`.

Hay dos formas de corregir este problema. Una consiste en usar el operador `!` correctamente. Al usar el operador `!`, encierre en paréntesis el argumento. La forma correcta de escribir la expresión booleana anterior es:

```
if (!(tiempo > limite))
      Algo
else
   Algo_Mas
```

Otra forma de corregir este problema es evitar el uso del operador `!`. Por ejemplo, lo que sigue también es correcto y más fácil de entender:

```
if (tiempo <= limite)
      Algo
else
     Algo_Mas
```

Evite usar "no"

Casi siempre podemos evitar el uso del operador `!`, y algunos programadores recomiendan evitarlo en la medida de lo posible, pues argumentan que así como los negativos en español pueden hacer las cosas difíciles de entender, el operador `!` puede hacer a los programas en C++ difíciles de entender. No hay necesidad de evitar obsesivamente el uso de este operador, pero antes de usarlo es bueno ver si podemos expresar lo mismo con mayor claridad sin usarlo.

El tipo *`bool`* es nuevo

Las versiones más viejas de C++ no tienen el tipo *`bool`*, y en vez de ello usan los enteros `1` y `0` para *`true`* y *`false`*. Si el lector tiene una versión vieja de C++ que carece del tipo *`bool`*, debería obtener un nuevo compilador.

Ejercicios de AUTOEVALUACIÓN

usando los operadores booleanos `&&`, `||` y `!`.

1. Determine el valor, *`true`* o *`false`*, de cada una de las siguientes expresiones booleanas, suponiendo que el valor de la variable `cuenta` es `0` y que el valor de la variable `limite` es `10`. Dé como respuesta uno de los valores *`true`* o *`false`*.

   a) `(cuenta == 0) && (limite < 20)`

   b) `cuenta == 0 && limite < 20`

   c) `(limite > 20) || (cuenta < 5)`

d) `!(cuenta == 12)`

e) `(cuenta == 1) && (x < y)`
f) `(cuenta < 10) || (x < y)`
g) `!( ((cuenta < 10) || (x < y)) && (cuenta >= 0) )`
h) `((limite/cuenta) > 7) || (limite < 20)`
i) `(limite < 20) || ((limite/cuenta) > 7)`
j) `((limite/cuenta) > 7) && (limite < 0)`
k) `(limite < 0) && ((limite/cuenta) > 7)`
l) `(5 && 7) + (!6)`

2. Cite dos tipos de instrucciones de C++ que alteren el orden en que se efectúan las acciones. Dé algunos ejemplos.

3. En álgebra universitaria vemos que se dan intervalos numéricos de esta manera:

   `2 < x < 3`

   En C++ este intervalo no tiene el significado que cabría esperar. Explique y dé la expresión booleana en C++ correcta que especifique que `x` está entre `2` y `3`.

4. ¿Lo siguiente produce una división entre cero?

```
j = -1;
if ((j > 0) && (1/(j+1) > 10))
```

```
        cout << i << endl;
```

Funciones que devuelven un valor booleano

Una función puede devolver un valor *bool* igual que de cualquier otro tipo predefinido, y se puede usar en una expresión booleana para controlar una instrucción *if-else*, para controlar un ciclo, o en cualquier otro lugar en el que se permite una expresión booleana. El tipo devuelto de semejante función debe ser *bool*.

Una llamada a una función que devuelve un valor booleano de *true* o *false* se puede usar en cualquier lugar en el que se permite una expresión booleana. En muchos casos esto puede hacer más comprensible un programa. Con una declaración de función podemos asociar una expresión booleana compleja a un nombre informativo y usar el nombre como expresión booleana en una instrucción *if-else* o en cualquier otro lugar en el que se permita una expresión booleana. Por ejemplo, la instrucción

```
if (((tasa >= 10) && (tasa < 20)) || (tasa == 0))
{
    ...
}
```

puede escribirse así

```
if (correcta(tasa))
```

```
{
    ...
}
```

siempre que se haya definido la siguiente función:

```
bool correcta(int tasa)
{
     return (((tasa >= 10) && (tasa < 20)) || (tasa == 0));
}
```

Ejercicios de AUTOEVALUACIÓN

5. Escriba una definición para una función llamada `en_orden` que reciba tres argumentos de tipo *`int`*. La función devuelve *`true`* si los tres argumentos están en orden ascendente; si no, devuelve *`false`*. Por ejemplo, `en_orden(1, 2, 3)` y `en_orden(1, 2, 2)` devuelven *`true`*, pero `en_orden(1, 3, 2)` devuelve *`false`*.

6. Escriba una definición para una función llamada `par` que reciba un argumento de tipo *`int`* y devuelva un valor *`bool`*. La función devuelve *`true`* si su argumento es un número par; si no, devuelve *`false`*.

7. Escriba una definición para una función llamada `esDigito` que reciba un argumento de tipo *`char`* y devuelva un valor *`bool`*. La función devuelve *`true`* si su argumento es un dígito decimal; si no, devuelve *`false`*.

8. Escriba una definición de función para una función llamada `esRaizde` que reciba dos argumentos de tipo *`int`* y devuelva un valor *`bool`*. La función devuelve *`true`* si su primer argumento es la raíz cuadrada del segundo; si no, devuelve *`false`*.

tipo de enumeración

Tipos de enumeración (*opcional*)

Un **tipo de enumeración** es un tipo cuyos valores se definen con una lista de constantes de tipo *`int`*. Un tipo de enumeración es algo muy parecido a una lista de constantes declaradas.

Al definir un tipo de enumeración podemos usar cualesquier valores *`int`*, y podemos definir cualquier cantidad de constantes en el tipo. Por ejemplo, el siguiente tipo de enumeración define una constante para la longitud en días de cada mes:

```
enum LargoMes { LARGO_ENE = 31, LARGO_FEB = 28,
    LARGO_MAR = 31, LARGO_ABR = 30, LARGO_MAY = 31,
    LARGO_JUN = 30, LARGO_JUL = 31, LARGO_AGO = 31,
    LARGO_SEP = 30, LARGO_OCT = 31, LARGO_NOV = 30,
    LARGO_DIC = 31 };
```

Como se aprecia en este ejemplo, dos o más constantes nombradas en un tipo de enumeración pueden recibir el mismo valor *`int`*.

Si no especificamos valores numéricos, se asignan a los identificadores de la definición de un tipo de enumeración valores a partir de `0`. Por ejemplo, la definición de tipo:

```
enum Direccion { NORTE = 0, SUR = 1, ESTE = 2, OESTE = 3 };
```

es equivalente a

```
enum Direccion { NORTE, SUR, ESTE, OESTE };
```

La forma que no da explícitamente los valores `int` se usa normalmente cuando lo único que queremos es una lista de nombres y no nos importa qué valores tengan.

Si inicializamos una constante de enumeración con algún valor, digamos

```
enum MiEnum { UNO = 17, DOS, TRES, CUATRO = -3, CINCO };
```

entonces `UNO` recibe el valor `17`, `DOS` recibe el siguiente valor `int`, `18`, `TRES` recibe el siguiente valor, `19`, `CUATRO` recibe `-3` y `CINCO` recibe el siguiente valor, `-2`.

En síntesis, el valor predeterminado para la primera constante de enumeración es `0`. Las demás se incrementan en 1 a menos que establezcamos una o más de las constantes de enumeración.

7.2 Bifurcaciones multivía

"¿Podría decirme, por favor,
hacia dónde debo ir?
"Eso depende en buena parte de a dónde
quieres llegar", dijo el Gato.

Lewis Carroll, *Alicia en el País de las Maravillas*

mecanismo de bifurcación

Cualquier construcción de programación que escoge una entre varias acciones alternativas es un **mecanismo de bifurcación**. La instrucción `if-else` escoge entre dos alternativas. En esta sección veremos métodos para escoger entre más de dos alternativas.

Instrucciones anidadas

sangrado

Como hemos visto, las instrucciones `if-else` e `if` contienen instrucciones más pequeñas. Hasta ahora hemos usado como tales subinstrucciones más pequeñas instrucciones compuestas e instrucciones simples, como las de asignación, pero hay otras posibilidades. De hecho, se puede usar cualquier instrucción como subparte de un instrucción `if-else`, `if`, `while` o `do-while`. Esto se ilustra en el cuadro 7.3. La instrucción de ese cuadro tiene tres niveles de anidamiento, como indican los rectángulos. Hay dos instrucciones `cout` anidadas dentro de una instrucción `if-else`, y ésta se encuentra anidada dentro de una instrucción `if`.

Al anidar instrucciones, normalmente sangramos cada nivel de subinstrucciones anidadas. En el cuadro 7.3 hay tres niveles de anidamiento, así que hay tres niveles de sangrado. Las dos instrucciones `cout` están sangradas la misma cantidad porque los dos están en el mismo nivel de anidamiento. Más adelante veremos casos específicos en los que es lógico usar otros patrones de sangrado, pero a menos que haya alguna regla que se oponga, es recomendable sangrar cada nivel de anidamiento como se ilustra en el cuadro 7.3.

TIP DE PROGRAMACIÓN

Use llaves en instrucciones anidadas

Supongamos que queremos escribir un instrucción `if-else` que se usará en un sistema de monitoreo computarizado a bordo de un automóvil de carreras. Esta parte del programa advierte al conductor cuando el combustible anda bajo, pero le indica que no se detenga en los fosos si el tanque está casi lleno. En todas las demás situaciones el programa no produce salidas para no distraer al conductor. Diseñamos el siguiente pseudocódigo:

Si la lectura de combustible marca menos de 3/4 ll
  Verificar si la lectura marca menos de 1/4 l a de combustible bajo si así es.
Si no (es decir, si la lectura marca más de 3/4 lleno):
  Indicar al conductor que no se detenga.

Lea el texto para ver qué problema hay aquí.

Si no somos cuidadosos, podríamos implementar el pseudocódigo así:

```
if (lectura_combustible < 0.75)
    if (lectura_combustible < 0.25)
        cout << "Combustible muy bajo. Cuidado!\n";
```

CUADRO 7.3 ***Instrucción*** `if-else` ***dentro de una instrucción*** `if`

```
if (cuenta > 0)
    if (puntos > 5)
        cout << "cuenta > 0 y puntos > 5\n";
    else
        cout << "cuenta > 0 y puntos <= 5\n";
```

```
    else
        cout << "Mas de 3/4 de tanque. No se detenga ahora!\n";
```

Esta implementación se ve bien, y de hecho es una instrucción C++ correctamente formada que el compilador aceptará y que se ejecutará sin mensajes de error. Sin embargo, no implementa el pseudocódigo. Observe que la instrucción incluye dos ocurrencias de *if* y sólo una de *else*. El compilador debe decidir cuál *if* se aparea con el único *else*. Hemos sangrado esta instrucción anidada de modo que sea evidente que el else se debe aparear con el primer *if*, pero al compilador no le interesa el sangrado. Para el compilador, la instrucción anidada anterior equivale a la siguiente versión, que sólo difiere en el sangrado:

else **colgante**

```
    if (lectura_combustible < 0.75)
```

CUADRO 7.4 *La importancia de las llaves (parte 1 de 2)*

```
//Ilustra la importancia de usar llaves en enunciados if-else.
#include<iostream>
using namespace std;
int main()
{
    double lectura_combustible;

    cout << "Escriba la lectura de combustible: ";
    cin >> lectura_combustible;

    cout << "Primero con llaves:\n";
    if (lectura_combustible < 0.75)
    {
        if (lectura_combustible < 0.25)
                cout << "Combustible muy bajo. Cuidado!\n";
    }
    else
    {
        cout << "Mas de 3/4 de tanque. No se detenga ahora!\n";
    }

    cout << "Ahora sin llaves:\n";
    if (lectura_combustible < 0.75)
        if (lectura_combustible < 0.25)
            cout << "Combustible muy bajo. Cuidado!\n";
    else
        cout << "Mas de 3/4 de tanque. No se detenga ahora!\n";

    return 0;
}
```

Este sangrado es bonito pero el compilador no le hace caso.

CUADRO 7.4 *La importancia de las llaves (parte 2 de 2)*

Diálogo de ejemplo 1

```
Escriba la lectura de combustible: 0.1
Primero con llaves:
Combustible muy bajo. Cuidado!
Ahora sin llaves:
Combustible muy bajo. Cuidado!
```

Las llaves no afectan en este caso, pero vea el Diálogo 2.

Diálogo de ejemplo 2

```
Escriba la lectura de combustible: 0.5
Primero con llaves:
Ahora sin llaves:
Mas de 3/4 de tanque. No se detenga ahora!
```

No deberá haber salida aquí y, gracias a las llaves, no la hay.

Salida incorrecta de la versión sin llaves.

regla para aparear los `else` con los `if`

```
if (lectura_combustible < 0.25)
    cout << "Combustible muy bajo. Cuidado!\n";
else
    cout << "Mas de 3/4 de tanque. No se detenga ahora!\n";
```

Para nuestra desgracia, el compilador usará la segunda interpretación y apareará el `else` con el segundo `if`, no con el primero. Esto se conoce como el **problema del `else` colgante**, y se ilustra con el programa del cuadro 7.4.

El compilador siempre aparea un `else` con el `if` anterior más cercano que no está ya apareado con algún `else`. Sin embargo, no trate de ajustarse a esta regla. ¡Haga caso omiso de la regla! ¡Cambie las reglas! ¡Usted es el que manda! Siempre dígale al compilador lo que quiere que haga y entonces el compilador hará lo que usted quiera. ¿Cómo le decimos al compilador lo que queremos? Usamos llaves. Las llaves en las instrucciones anidadas son como los paréntesis en las expresiones aritméticas. Las llaves le indican al compilador cómo debe agrupar las cosas, en vez de dejar que se agrupen según las convenciones por omisión, que podrían o no ser lo que queremos. A fin de evitarnos problemas y hacer a los programas más comprensibles, debemos encerrar en llaves `{` y `}` las subinstrucciones de las instrucciones `if-else`, como hemos hecho en la primera instrucción `if-else` del cuadro 7.4.

En el caso de subinstrucciones muy sencillas, digamos una sola instrucción de asignación o una sola instrucción `cout`, podemos omitir las llaves sin peligro. En el cuadro 7.4, las llaves alrededor de la siguiente subinstrucción (que está dentro de la primera instrucción `if-else`) no son necesarias:

```
cout << "Mas de 3/4 de tanque. No se detenga ahora!\n";
```

Sin embargo, aun en estos casos tan sencillos, las llaves pueden ayudar a la comprensión. Algunos programadores recomiendan encerrar en llaves hasta la sub instrucciones más sencillas cuando ocurren dentro de instrucciones *if-else*, y eso es lo que hemos hecho en la primera instrucción *if-else* del cuadro 7.4.

Instrucciones *if-else* multivía

Una instrucción *if-else* es una bifurcación de dos vías: permite al programa elegir una de dos acciones posibles. En muchos casos querremos tener una bifurcación de tres o cuatro vías para que el programa pueda escoger entre más de dos acciones alternativas. Podemos implementar tales bifurcaciones multivía anidando instrucciones *if-else*. Por ejemplo, supongamos que estamos diseñando un juego de computadora en el que el usuario debe adivinar el valor de algún número. El número puede estar en una variable llamada `numero`, y la conjetura puede estar en una variable llamada `conjetura`. Si queremos dar una pista después de cada intento de adivinar, podríamos diseñar el siguiente pseudocódigo:

```
Desplegar "Demasiado alto." si conjetura > numero.
Desplegar "Demasiado bajo." si conjetura < numero.
Desplegar "Correcto!" si conjetura == numero.
```

Siempre que una acción de bifurcación se describe como una lista de condiciones mutuamente exclusivas y sus acciones correspondientes, como en este ejemplo, podemos implementarla usando un instrucción *if-else* anidada. Por ejemplo, el pseudocódigo anterior se traduce a lo siguiente:

sangrado

```
if (conjetura > numero)
   cout << "Demasiado alto.";
else if (conjetura < numero)
   cout << "Demasiado bajo.";
else if (conjetura == numero)
   cout << "Correcto!";
```

El patrón de sangrado que usamos aquí es un poco dife mendamos antes. Si siguiéramos nuestras reglas de sangrado, producir

Use el patrón de sangrado anterior en lugar de éste.

```
if (conjetura > numero)
    cout << "Demasiado alto.";
else
    if (conjetura < numero)
        cout << "Demasiado bajo.";

    else
       if (conjetura == numero)
          cout << "Correcto!";
```

Éste es uno de esos poco frecuentes casos en los que no debemos seguir nuestras pautas generales para sangrar instrucciones anidadas. La razón es que al alinear todos los *else* también alineamos todos los pares condición/acción y con ello hacemos que la organización del programa refleje nuestro razonamiento. Otra razón es que, incluso cuando las ins-

trucciones *if-else* no están anidadas muy profundamente, ¡es fácil que se nos acabe el espacio en la página!

Puesto que las condiciones son mutuamente exclusivas, el último *if* de la instrucción *if-else* anidada anterior es superfluo y podemos omitirlo, pero a veces es mejor incluirlo en un comentario así:

```
if (conjetura > numero)
   cout << "Demasiado alto.";
else if (conjetura < numero)
   cout << "Demasiado bajo.";
else  // (conjetura == numero)
```

Instrucción *if-else* multivía

Sintaxis

```
if (Expresion_Booleana_1)
     Instruccion_1
else if (Expresion_Booleana_2)
     Instruccion_2
        .
        .
        .
else if (Expresion_Booleana_n)
      Instruccion_n
else
      Instruccion_Para_Todas_Las_Demas_Posibilidades
```

Ejemplo

```
if ((temperatura < −10) && (dia == DOMINGO))
     cout << "Quedate en casa.";
else if (temperatura < −10) // y dia != DOMINGO
     cout << "Quedate en casa, pero llama al trabajo.";
else if (temperatura <= 0 // y temperatura >= −10
     cout << "Abrigate bien.";
else //temperatura > 0
     cout << "Trabaja y juega intensamente.";
```

Las expresiones booleanas se verifican en orden hasta que se encuentra la primera expresión booleana que es *true* y entonces se ejecuta la instrucción correspondiente. Si ninguna de las expresiones booleanas es *true*, se ejecuta *la_Instruccion_Para_Todas_Las_Demas_Posibilidades*.

EJEMPLO DE PROGRAMACIÓN
Impuesto sobre la renta estatal

```
cout << "Correcto!";
```

Podemos usar esta forma de instrucción *if-else* de bifurcación múltiple incluso si las condiciones no son mutuamente exclusivas. En todo caso, la computadora evaluará las condiciones en el orden en que aparecen hasta que encuentre la primera condición que sea *true*, y luego ejecutará la acción que corresponde a esa condición. Si ninguna condición es *true*, no se efectuará ninguna acción. Si la instrucción termina con un *else* sin *if*, se ejecutará la última instrucción si todas las condiciones son *false*.
El cuadro 7.5 contiene una definición de función que usa una instrucción *if-else* multivía. La función recibe un argumento, que es el ingreso neto del contribuyente redondeado a un número entero de pesos y calcula el impuesto sobre la renta estatal a pagar sobre estos ingresos netos. El estado en cuestión calcula los impuestos según el siguiente esquema:

1. Los primeros $15,000 de ingreso neto no pagan impuesto.
2. Se cobra un impuesto del 5% por cada peso de ingreso neto entre $15,001 y $25,000.
3. Se cobra un impuesto del 10% por cada peso de ingreso neto por arriba de $25,000.

La función definida en el cuadro 7.5 usa una instrucción *if-else* multivía con una acción para cada uno de los tres casos anteriores. La condición para el segundo caso en realidad es más complicada que lo indispensable. La computadora no llegará a la segunda

Ejercicios de AUTOEVALUACIÓN

condición si no probó antes la primera y la encontró *false*. Por tanto, sabemos que si la computadora prueba la segunda condición ya sabe que `ingreso_neto` es mayor que `15000`. Entonces, podríamos sustituir la línea

```
else if ((ingreso_neto > 15000) && (ingreso_neto <= 25000))
```

por la siguiente, y el programa funcionaría exactamente igual:

```
else if (ingreso_neto <= 25000)
```

9. ¿Qué salida producirá el siguiente código si se incrusta en un programa completo?

```
int x = 2;
cout << "Inicio\n";
if (x <= 3)
     if (x != 0)
          cout << "Saludos desde el segundo if.\n";
     else
```

CUADRO 7.5 *Instrucción* `if-else` *multivía (parte 1 de 2)*

```
//Programa para calcular un impuesto sobre la renta estatal.
#include<iostream>
using namespace std;

double impuesto(int ingreso_neto);
//Precondición: El parámetro formal ingreso_neto es el ingreso
//neto redondeado a dólares cerrados. Devuelve el
//monto del impuesto sobre la renta estatal que se calcula así:
//ningún impuesto sobre ingresos hasta $15,000; 5% sobre ingresos
//entre $15,001 y $25,000, más 10% sobre ingresos arriba de $25,000.

int main()
{
    int ingreso_neto;
    double impuesto_a_pagar;

    cout << "Escriba el ingreso neto (redondeado a pesos) $";
    cin >> ingreso_neto;

    impuesto_a_pagar = impuesto(ingreso_neto);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Ingreso neto = $" << ingreso_neto << endl
         << "Impuesto a pagar = $" << impuesto_a_pagar << endl;

    return 0;
}

double impuesto(int ingreso_neto)
{
    double impuesto_cinco_porciento, impuesto_diez_porciento;
```

CUADRO 7.5 *Instrucción* `if-else` *multivía (parte 2 de 2)*

```
        if (ingreso_neto <= 15000)
            return 0;
        else if ((ingreso_neto > 15000) && (ingreso_neto <= 25000))
            //devolver 5% del monto más allá de $15,000
            return (0.05*(ingreso_neto - 15000));
        else //ingreso_neto > $25,000
        {
            //impuesto_cinco_porciento = 5% del ingreso de $15,000
            //a $25,000.
            impuesto_cinco_porciento = 0.05*10000;
            //impuesto_diez_porciento = 10% del ingreso arriba de $25,000.
            impuesto_diez_porciento = 0.10*(ingreso_neto - 25000);
            return (impuesto_cinco_porciento + impuesto_diez_porciento);
        }
    }
```

Diálogo de ejemplo

```
Escriba el ingreso neto (redondeado a pesos) $25100
Ingreso neto = $25100.00
Impuesto a pagar = $510.00
```

```
        cout << "Saludos desde el else.\n";
cout << "Final\n";

cout << "Otra vez\n";
if (x > 3)
   if (x != 0)
      cout << "Saludos desde el segundo if.\n";
   else
      cout << "Saludos desde el else.\n";
cout << "Final de nuevo\n";
```

10. ¿Qué salida producirá el siguiente código si se incrusta en un programa completo?

```
int extra = 2;
if (extra < 0)
   cout << "pequenio";
else if (extra == 0)
   cout << "mediano";
```

```
else
   cout << "grande";
```

11. ¿Cuál sería la salida del ejercicio 10 si la asignación se cambiara a lo siguiente?

```
int extra = - 37;
```

12. ¿Cuál sería la salida del ejercicio 10 si la asignación se cambiara a lo siguiente?

```
int extra = 0;
```

13. ¿Qué salida producirá el siguiente código si se incrusta en un programa completo?

```
int x = 200;
cout << "Inicio\n";
if (x < 100)
    cout << "Primera salida.\n";
else if (x > 10)
    cout << "Segunda salida.\n";
else
    cout << "Tercera salida.\n";
cout << "Fin\n";
```

14. ¿Cuál sería la salida del ejercicio 13 si la expresión booleana (x > 10) se cambiara a (x > 100)?

15. ¿Qué salida producirá el siguiente código si se incrusta en un programa completo?

```
int x = ALGUNA_CONSTANTE;
cout << "Inicio\n";
if (x < 100)
    cout << "Primera salida.\n";
else if (x > 100)
    cout << "Segunda salida.\n";
else
    cout << x << endl;
cout << "Fin\n";
```

`ALGUNA_CONSTANTE` es una constante de tipo *`int`*. Asuma que ni "`Primera salida`" ni "`Segunda salida`" se despliegan. Por lo tanto sabemos que el valor de `x` se desplegará.

16. Escriba una instrucción *`if-else`* multivía que clasifique el valor de una variable *`int`* `n` como de una de las siguientes categorías y despliegue un mensaje apropiado:

`n < 0` **o** `0 ≤ n ≤ 100` **o** `n > 100`

17. Dados la siguiente declaración e instrucción de salida, suponga que se han incrustado en un programa correcto que entonces se ejecuta. ¿Qué salida se produce?

```
enum Direccion { N, S, E, O };
//...
cout << O << " " << E << " " << S << " "  << N << endl;
```

CUADRO 7.6 *Una instrucción* `switch` *(parte 1 de 2)*

```
//Programa para ilustrar el enunciado switch.
#include<iostream>
using namespace std;

int main()
{
    char calif;

    cout << "Escribe tu calificacion parcial y oprime Intro: ";
    cin >> calif;

    switch (calif)
    {
         case 'A':
              cout << "Excelente. "
                   << "No necesitas hacer examen final.\n";
              break;
         case 'B':
              cout << "Muy bien. ";
              calif = 'A';
              cout << "Tu calificacion parcial ahora es "
                   << calif << endl;
              break;
         case 'C':
              cout << "Aprobado.\n";
              break;
         case 'D':
         case 'F':
              cout << "Nada bien. "
                   << "Vete a estudiar.\n";
              break;
         default:
             cout << "Esa calificacion no es posible.\n";
    }
    cout << "Fin del programa.\n";
    return 0;
}
```

CUADRO 7.6 *Una instrucción* `switch` *(parte 2 de 2)*

Diálogo de ejemplo 1

```
Escribe tu calificacion parcial y oprime Intro: A
Excelente. No necesitas hacer examen final.
Fin del programa.
```

Diálogo de ejemplo 2

```
Escribe tu calificacion parcial y oprime Intro: B
Muy bien. Tu calificacion parcial ahora es A.
Fin del programa.
```

Diálogo de ejemplo 3

```
Escribe tu calificacion parcial y oprime Intro: D
Nada bien. Vete a estudiar.
Fin del programa.
```

Diálogo de ejemplo 4

```
Escribe tu calificacion parcial y oprime Intro: E
Esa calificacion no es posible.
Fin del programa.
```

18. Dados la siguiente declaración e instrucción de salida, suponga que se han incrustado en un programa correcto que entonces se ejecuta. ¿Qué salida se produce?

```
enum Direccion { N = 5, S = 7, E = 1, O };
//...
cout << O << " " << E << " " << S << " "  << N << endl;
```

La instrucción *switch*

expresión controladora

Ya vimos el uso de instrucciones *if-else* para construir bifurcaciones multivía. La **instrucción** *switch* es otro tipo de instrucción C++ que implementa bifurcaciones multivía. En el cuadro 7.6 se muestra una instrucción *switch* de ejemplo. Esta instrucción en particular tiene cuatro bifurcaciones normales y una quinta para entradas no válidas. La variable `calif` determina cuál bifurcación se ejecuta. Hay una bifurcación para cada una de las calificaciones `'A'`, `'B'` y `'C'`. Las calificaciones `'D'` y `'F'` hacen que se tome la misma bifurcación; no hay una acción individual para cada una de ellas. Si el valor de `calif` es cualquier carácter distinto de `'A'`, `'B'`, `'C'`, `'D'` o `'F'`, se ejecuta la instrucción `cout` que está después de la palabra clave `default`.

La sintaxis y el patrón de sangrado preferido para la instrucción `switch` se muestran en el ejemplo del cuadro 7.6 y en el recuadro de la página 356.

Cuando se ejecuta una instrucción *switch*, se toma una de varias bifurcaciones diferentes. La decisión de cuál bifurcación ejecutar la determina una **expresión controladora** que se da entre paréntesis después de la palabra clave *switch*. La expresión controladora de la instrucción *switch* de ejemplo del cuadro 7.6 es de tipo *char*. La expresión controladora de una instrucción *switch* siempre debe devolver un valor *bool*, una constante `enum`, uno de los tipos enteros o un carácter. Cuando se ejecuta la instrucción *switch*, esta expresión controladora se evalúa y la computadora examina los valores constantes que se dan después de las diversas ocurrencias de la palabra clave `case`. Si la computadora encuentra una constante cuyo valor es igual al valor de la expresión controladora, ejecuta el código de ese *case* (caso). Por ejemplo, si la evaluación de la expresión da `'B'`, la computadora busca la siguiente línea y ejecuta las instrucciones que le siguen:

instrucción *break*

```
case 'B':
```

Observe que la constante va seguida de un signo de dos puntos. Cabe señalar que no puede haber dos ocurrencias de *case* seguidas del mismo valor constante, pues entonces la instrucción sería ambigua.

Una **instrucción** *break* consiste en la palabra clave *break* seguida de un signo de punto y coma. Cuando la computadora ejecuta las instrucciones que siguen a un rótulo *case*, continúa hasta llegar a un instrucción *break*. Cuando la computadora encuentra una instrucción *break*, la instrucción *switch* termina. Si omitimos las instrucciones *break*, entonces después de ejecutar el código de un *case* la computadora continuará con la ejecución del código del siguiente *case*.

Cabe señalar que podemos tener dos rótulos *case* para la misma sección de código. En la instrucción *switch* del cuadro 7.6 se realiza la misma acción para los valores `'D'` y `'F'`. Esta técnica también puede servir para manejar letras tanto mayúsculas como minúsculas. Por ejemplo, si queremos permitir tanto `'a'` minúscula como `'A'` mayúscula en el programa del cuadro 7.6, podemos sustituir

```
case 'A':
     cout << "Excelente. "
          << "No necesitas hacer examen final.\n";
    break;
```

default

por lo siguiente:

```
case 'A':
case 'a':
```

Instrucción `switch`

Sintaxis

```
switch (Expresion_Controladora)
{
      case Constante_1:
           Sucesion_de_Instrucciones_1
           break;
      case Constante_2:
           Sucesion_de_Instrucciones_2
           break;
                    .
                    .
                    .
      case Constante_n:
           Sucesion_de_Instrucciones_n
           break;
      default:
            Sucesion_de_Instrucciones_Por_Omision
}
```

Ejemplo

```
int clase_vehiculo;
cout << "Indica la clase de vehiculo: ";
cin >> clase_vehiculo;

switch (clase_vehiculo)
{
     case 1;
          cout << "Automovil.";
          peaje = 0.50;
          break;
      case 2;
          cout << "Autobus.";
          peaje = 1.50;
          break;
      case 3;
          cout << "Camion.";
          peaje = 2.00;
          break;
       default:
          cout << "Clase de vehiculo desconocida!";
}
```

Si olvidamos este break, los automóviles pagarán $1.50.

RIESGO *Olvidar un* `break` *en una instrucción* `switch`

Si olvidamos un `break` en una instrucción `switch`, el compilador no generará un mensaje de error. Habremos escrito una instrucción `switch` sintácticamente correcta, pero no hará lo que queríamos que hiciera. Consideremos la instrucción `switch` del recuadro intitulado “Instrucción

switch". Si omitiéramos un instrucción *break*, como indica la flecha, entonces cuando la variable `clase_vehiculo` tenga el valor `1`, se ejecutará el caso rotulado

`case 1:`

como se desea, pero luego la computadora ejecutará también el siguiente *case*. Esto producirá una salida desconcertante que dice que el vehículo es un automóvil y luego dice que es un autobús; además, el valor final de `peaje` será `1.50`, no `0.50` como debía ser. Cuando la computadora comienza a ejecutar un *case*, no se detiene hasta que encuentre un *break* o llegue al final de la instrucción *switch*.

```
        cout << "Excelente. "
             << "No necesitas hacer examen final.\n";
        break;
```

Desde luego, podemos hacer lo mismo con todas las demás letras.

menú

Si ningún rótulo *case* tiene una constante que coincida con el valor de la expresión controladora, se ejecutarán las instrucciones que siguen al rótulo *default*. No es necesario incluir una sección *default*. Si no hay sección *default* y no se encuentra una concordancia con el valor de la expresión controladora, no sucederá nada al ejecutarse la instrucción *switch*. Sin embargo, lo más seguro es siempre incluir una sección *default*. Si cree que sus rótulos *case* abarcan todos los posibles resultados, podría poner un mensaje de error en la sección *default*. Esto es lo que hicimos en el cuadro 7.6.

Uso de instrucciones *switch* para menús

La instrucción *if-else* multivía es más versátil que la instrucción *switch*, y podemos usar una instrucción *if-else* multivía en cualquier lugar que podamos usar una instrucción *switch*. No obstante, hay casos en que la instrucción *switch* es más clara. Por ejemplo, la instrucción *switch* es perfecta para implementar *menús*.

Un *menú* de un restaurante presenta una lista de alternativas entre las que un cliente puede escoger. Un **menú** en un programa de computadora hace lo mismo: presenta en la pantalla una lista de alternativas para que el usuario escoja una. El cuadro 7.7 muestra el bosquejo de un programa diseñado para proporcionar a los estudiantes información acerca de sus tareas. El programa usa un menú para que el estudiante pueda escoger qué información desea. (Si el lector quiere ver en acción la parte de menú de este programa antes de diseñar las funciones que usa, puede usar stubs en lugar de las definiciones de funciones. Vimos los stubs en el capítulo 4.)

TIP DE PROGRAMACIÓN

Use llamadas de función en instrucciones de bifurcación

La instrucción *switch* y la instrucción multivía *if-else* nos permiten colocar varias instrucciones distintas en cada bifurcación. Sin embargo, ello puede hacer a la instrucción

CUADRO 7.7 *Un Menú (parte 1 de 2)*

```
//Programa para proporcionar información sobre tareas.
#include<iostream>
using namespace std;

void mostrar_tarea();
//Muestra en la pantalla la siguiente tarea.

void mostrar_calif();
//Pide un número de estudiante y da la calificación correspondiente.

void sugerencias();
//Da sugerencias para hacer la tarea actual.

int main()
{
    int opcion;

    do
    {
        cout << endl
             << "Escoge 1 para ver la siguiente tarea.\n"
             << "Escoge 2 para ver que sacaste en tu ultima tarea.\n"
             << "Escoge 3 para ver sugerencias de como hacer la tarea.\n"
             << "Escoge 4 para salir de este programa.\n"
             << "Escribe tu opcion y oprime Intro: ";
        cin >> opcion;

        switch (opcion)
         {
             case 1:
                mostrar_tarea();
                break;
             case 2:
                mostrar_calif();
                break;
             case 3:
                sugerencias();
                break;
```

CUADRO 7.7 *Un Menú (parte 2 de 2)*

```
            case 4:
                  cout << "Fin del programa.\n";
                  break;
            default:
                  cout << "Opcion no valida.\n"
                       << "Escoge otra vez.\n";
        }
    }while (opcion != 4);
    return 0;
}
```

<Las definiciones de las funciones `mostrar_tarea`, `mostrar_calif` y `sugerencias` se insertan aquí.>

Diálogo de muestra

```
Escoge 1 para ver la siguiente tarea.
Escoge 2 para ver que sacaste en tu ultima tarea.
Escoge 3 para ver sugerencias de como hacer la tarea.
Escoge 4 para salir de este programa.
Escribe tu opcion y oprime Intro: 3

Sugerencias para la tarea:
Analiza el problema.
Escribe un algoritmo en pseudocodigo.
Traduce el pseudocodigo a un programa C++.

Escoge 1 para ver la siguiente tarea.
Escoge 2 para ver que sacaste en tu ultima tarea.
Escoge 3 para ver sugerencias de como hacer la tarea.
Escoge 4 para salir de este programa.
Teclea tu opcion y oprime Intro: 4
Fin del programa.
```

La salida exacta dependerá de la definición de la función `sugerencias`.

`switch` y `if-else` difíciles de entender. Examinemos la instrucción `switch` del cuadro 7.7. Cada bifurcación de las opciones `1`, `2` y `3` es una sola llamada de función. Esto hace que la organización de la instrucción `switch` y la estructura general del programa sean claras. Si en vez de ello hubiéramos colocado todo el código para cada bifurcación en la instrucción `switch`, en lugar de en las definiciones de funciones, la instrucción `switch` sería un mar incomprensible de instrucciones C++. De hecho, la instrucción `switch` ni siquiera cabría en una pantalla.

Bloques

variables locales

Cada bifurcación de una instrucción `switch` o de una instrucción `if-else` es una subtarea individual. Como se indicó en el tip de programación anterior, a menudo lo mejor es hacer que la acción de cada bifurcación sea una llamada de función. De ese modo, la subtarea de cada bifurcación se podrá diseñar, escribir y probar por separado. Por otra parte, hay casos en los que la acción de una bifurcación es tan sencilla que podemos especificarla con una instrucción compuesta. Habrá ocasiones en las que queramos dar a esta instrucción compuesta sus propias variables locales. Por ejemplo, consideremos el programa del cuadro 7.8, que calcula la factura de cierto número de artículos a un precio dado. Si la venta es una transacción al mayoreo, no se cobra I.V.A. (supuestamente porque el impuesto se pagará cuando los artículos se revendan a compradores al detalle). En cambio, si la venta es una transacción al detalle, se deberá sumar el I.V.A. Se usa una instrucción `if-else` para realizar diferentes cálculos con las compras al mayoreo y al detalle. En el caso de una compra al detalle, el cálculo usa una variable temporal llamada `subtotal`, por lo que esa variable se declara dentro de la instrucción compuesta de esa bifurcación de la instrucción `if-else`.

bloque
alcance

Como se muestra en el cuadro 7.8, la variable `subtotal` se declara dentro de una instrucción compuesta. Si quisiéramos, podríamos haber usado el nombre de variable `subtotal` para alguna otra cosa fuera de la instrucción compuesta en la que la declaramos. Una variable que se declara dentro de una instrucción compuesta es local respecto a dicha instrucción compuesta. Es como si hubiéramos hecho a la instrucción compuesta el cuerpo de una definición de función de modo que la declaración de variable estuviera dentro de una definición de función. La instrucción compuesta tiene una ventaja respecto a una definición de función: dentro de una instrucción compuesta podemos usar todas las variables declaradas fuera de dicha instrucción, además de las variables locales declaradas dentro de la instrucción compuesta.

bloque de instrucciones

instrucciones anidadas

Una instrucción compuesta con declaraciones es algo más que una simple instrucción compuesta, por lo que tiene un nombre especial. Una instrucción compuesta que contiene declaraciones de variables es un **bloque**, y decimos que las variables declaradas dentro del bloque son **locales respecto al bloque** o **tienen como alcance el bloque**.

Cabe señalar que el cuerpo de una función es un bloque. No existe un nombre estándar para un bloque que no sea el cuerpo de una función, pero como vamos a querer hablar de

Bloques

Un **bloque** es una instrucción compuesta que contiene declaraciones de variables. Las variables declaradas en un bloque son locales respecto al bloque, así que los nombres de las variables se pueden usar fuera del bloque para alguna otra cosa (digamos, como nombres de otras variables distintas).

CUADRO 7.8 *Bloque con una variable local (parte 1 de 2)*

```
//Programa para calcular la factura de una compra al mayoreo o al detalle.
#include<iostream>
using namespace std;
const double TASA_IVA = 0.05; //I.V.A. del 5%.

int main()
{
    char tipo_venta;
    int cantidad;
    double precio, total;

    cout << "Escriba el precio $";
    cin >> precio;
    cout << "Indique la cantidad comprada: ";
    cin >> cantidad;
    cout << "Escriba M si es una compra al mayoreo.\n"
         << "Escriba D si es una compra al detalle.\n"
         << "Luego oprima Intro.\n";
    cin >> tipo_venta;

    if ((tipo_venta == 'M') || (tipo_venta == 'm'))
    {
        total = precio * cantidad;
    }
    else if ((tipo_venta == 'D') || (tipo_venta == 'd'))
    {
         double subtotal;
         subtotal = precio * cantidad;
         total = subtotal + subtotal * TASA_IVA;
    }
    else
    {
         cout << "Error en las entradas.\n";
    }
```

local respecto al bloque

CUADRO 7.8 *Bloque con una variable local (parte 2 de 2)*

```
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << cantidad << " articulos a $" << precio << endl;
    cout << "Total de la factura = $" << total;
    if ((tipo_venta == 'D') || (tipo_venta == 'd'))
       cout << " con IVA incluido.\n";

    return 0;
}
```

Diálogo de ejemplo

```
Escriba el precio $10.00
Indique la cantidad comprada: 2
Escriba M si es una compra al mayoreo.
Escriba D si es una compra al detalle.
Luego oprima Intro.
D
2 articulos a $10.00
Total de la factura = $21.00 con IVA incluido.
```

Regla de alcance para bloques anidados

Si un identificador se declara como variable en dos bloques, y un bloque está dentro del otro, se trata de dos variables distintas que tienen el mismo nombre. Una variable existe sólo dentro del bloque interior y no es posible acceder a ella fuera del bloque interior. La otra variable sólo existe en el bloque exterior y no es posible acceder a ella dentro del bloque interior. Las dos variables son distintas, así que los cambios efectuados a una de ellas no tendrán efecto alguno sobre la otra.

RIESGO *Variables locales inadvertidas*

Cuando declaramos una variable dentro de un par de llaves, esa variable se convierte en una variable local del bloque encerrado en el par { }. Esto se cumple aunque no queramos que la variable sea local. Si queremos que una variable esté disponible fuera de las llaves, deberemos declararla fuera de ellas.

Ejercicios de AUTOEVALUACIÓN

este tipo de bloques crearemos un nombre para ellos. Llamaremos a un bloque **bloque de instrucciones** cuando no sea el cuerpo de una función (y no sea el cuerpo de la parte `main` del programa).

Los bloques de instrucciones se pueden anidar dentro de otros bloques de instrucciones, y en estos bloques anidados aplican básicamente las mismas reglas respecto a los nombres de variables locales que ya hemos estudiado. Sin embargo, aplicar tales reglas a bloques de instrucciones anidados tiene sus bemoles. Una mejor regla es nunca anidar bloques de instrucciones. Los bloques de instrucciones anidados dificultan la comprensión de los programas. Si siente la necesidad de anidar bloques de instrucciones, opte por convertir algunos de esos bloques en definiciones de funciones y use llamadas a funciones en lugar de bloques de instrucciones anidados. De hecho, recomendamos usar poco los bloques de instrucciones de cualquier tipo. En casi todas las situaciones una llamada de función es preferible a un bloque de instrucciones. Por integridad, incluiremos una regla de alcance para bloques anidados en el recuadro de resumen de la página 362.

19. ¿Qué salida producirá el siguiente código incrustado en un programa completo?

```
int primera_opcion = 1;
switch (primera_opcion + 1)
{
     case 1:
         cout << "Carne asada\n";
         break;
     case 2:
         cout << "Gusanos asados\n";
         break;
     case 3:
         cout << "Helado de chocolate\n";
     case 4:
         cout << "Helado de cebolla\n";
         break;
    default:
         cout << "Buen provecho!\n";
}
```

20. ¿Qué salida se generaría en el ejercicio de autoevaluación 19 si la primera línea se cambiara a lo que sigue?

```
int primera_opcion = 3;
```

21. ¿Qué salida se generaría en el ejercicio de autoevaluación 19 si la primera línea se cambiara a lo que sigue?

```
int primera_opcion = 2;
```

22. ¿Qué salida se generaría en el ejercicio de autoevaluación 19 si la primera línea se cambiara a lo que sigue?

```
int primera_opcion = 4;
```

23. ¿Qué salida producirá el siguiente código incrustado en un programa completo?

```
int numero = 22;
{
    int numero = 42;
    cout << numero << " ";
}
cout << numero;
```

24. Aunque le recomendamos no seguir este estilo al programar, incluiremos un ejercicio que usa bloques anidados para entender mejor las reglas de alcance. Indique la salida que el siguiente fragmento de código produciría si estuviera incrustado en un programa por lo demás completo y correcto.

```
{
  int x = 1;
  cout << x << endl;
  {
     cout << x << endl;
     int x = 2;
     cout << x << endl;
     {
        cout << x << endl;
        int x = 3;
        cout << x << endl;
     }
     cout << x << endl;
  }
  cout << x << endl;
}
```

7.3 Más acerca de las instrucciones cíclicas de C++

ciclo

cuerpo del ciclo
iteración del ciclo

comparación de `while` y `do-while`

No es verdad que la vida es una maldita cosa tras otra—
Es una maldita cosa una y otra vez.

Edna St. Vincent Millay,

Carta a Arthur Darison Ficke, 24 de octubre de 1930

Un **ciclo** es cualquier construcción de programa que repite un instrucción o sucesión de instrucciones varias veces. Las instrucciones `while` y `do-while` sencillas que ya hemos visto son ejemplos de ciclos. La instrucción (o grupo de instrucciones) que se repite en un ciclo es el **cuerpo** del ciclo, y cada repetición del cuerpo del ciclo es una **iteración** del ciclo. Los dos aspectos de diseño más importantes que hay que considerar al construir ciclos son: ¿Cuál debe ser el cuerpo del ciclo? ¿Cuántas veces debe iterarse el cuerpo del ciclo?

CUADRO 7.9 *Sintaxis de la instrucción* `while` *y de la instrucción* `do-while`

Instrucción `while` con cuerpo de una sola instrucción

```
while (Expresion_Booleana)
    Instruccion
```

Cuerpo

Instrucción `while` con cuerpo de múltiples instrucciones

```
while (Expresion_Booleana)
{
  Instruccion_1
  Instruccion _2
        .
        .
        .
  Instruccion _Ultimo
}
```

Cuerpo

Instrucción `do-while` con cuerpo de una sola instrucción

Cuerpo

```
do
  Instruccion
while (Expresion_Booleana);
```

Instrucción `do-while` con cuerpo de múltiples instrucciones

```
do
{
      Instruccion_1
      Instruccion_2
            .
            .
            .
      Instruccion_Ultimo
}while (Expresion_Booleana);
```

Cuerpo

Repaso de las instrucciones *while*

ejecución del cuerpo cero veces

La sintaxis de la instrucción *while* y su variante, la instrucción *do-while*, se repasan en el cuadro 7.9. La diferencia importante entre los dos tipos de ciclos tiene que ver con *cuándo* se verifica la expresión booleana controladora. En una instrucción while la expresión booleana se verifica *antes* de ejecutar el cuerpo del ciclo. Si la evaluación de la expresión booleana da *false*, el cuerpo no se ejecuta. En una instrucción *do-while*, primero se ejecuta el cuerpo del ciclo y la expresión booleana se verifica *después* de ejecutar el cuerpo del ciclo. Así pues, la instrucción *do-while* siempre ejecuta el cuerpo del ciclo por lo menos una vez. Después de esta acción inicial, los ciclos *while* y *do-while* tienen un comportamiento prácticamente idéntico. Después de cada iteración del cuerpo del ciclo, se verifica otra vez la expresión booleana y, si es *true*, el ciclo se itera otra vez. Si la expresión cambió de *true* a *false*, la instrucción cíclica termina.

Lo primero que sucede cuando se ejecuta un ciclo *while* es que se evalúa la expresión booleana controladora. Si la evaluación da *false* en ese punto, el cuerpo del ciclo nunca se ejecuta. Podría parecer que no tiene caso ejecutar el cuerpo de un ciclo cero veces, pero hay ocasiones en que ésa es la acción deseada. Por ejemplo, es común usar un ciclo *while* para obtener la sumatoria de una lista de números, pero la lista podría estar vacía. En términos más específicos, un programa de balanceo de chequera podría usar un ciclo *while* para sumar los valores de todos los cheques extendidos en un mes, pero tal vez usted salió de vacaciones durante todo el mes y no extendió ningún cheque. En ese caso, habrá cero números que sumar y el ciclo se iterará cero veces.

operador de incremento en expresiones

Repaso de los operadores de incremento y decremento

Ya usamos el operador de incremento como instrucción que incrementa en uno el valor de una variable. Por ejemplo, lo que sigue desplegará 42 en la pantalla:

```
int numero = 41;
numero++;
cout << numero;
```

Hasta aquí hemos empleado sólo como instrucción al operador de incremento. Sin embargo, el operador de incremento también es un operador, igual que los operadores + y −. Una expresión como numero++ también devuelve un valor, por lo que podemos usarla en una expresión aritmética como

```
2*(numero++)
```

La expresión numero++ primero devuelve el valor de la variable numero y *después* incrementa en uno el valor de numero. Por ejemplo, consideremos el siguiente código:

```
int numero = 2;
int valor_producido = 2*(numero++);
cout << valor_producido << endl;
cout << numero << endl;
```

comparación de v++ y ++v

Este código producirá la salida:

```
4
3
```

Tome nota de la expresión 2*(numero++). Cuando C++ evalúa esta expresión, usa el

valor que numero tenía *antes* de incrementarse, no el valor que tiene después de incrementarse. Así pues, el valor producido por la expresión numero++ es 2, a pesar de que el operador de incremento cambia el valor de numero a 3. Esto podría parecer extraño, pero a veces es justo lo que queremos. Y, como veremos a continuación, si queremos una expresión que se comporte de forma diferente, podemos escribirla.

La evaluación de la expresión v++ da el valor de la variable v y luego ese valor se incrementa en uno. Si invertimos el orden y colocamos el ++ antes de la variable, el orden de esas dos acciones se invertirá. La expresión ++v primero incrementa el valor de la variable v y luego devuelve este valor incrementado de v. Por ejemplo, consideremos el siguiente código:

```
int numero = 2;
int valor_producido = 2*(++numero);
cout << valor_producido << endl;
cout << numero << endl;
```

Este fragmento de código es igual al anterior, excepto que el ++ está antes de la variable. Por ello, este código producirá la siguiente salida:

operador de decremento

```
6
3
```

Cabe señalar que los dos operadores de incremento, numero++ y ++numero, tienen exactamente el mismo efecto sobre la variable numero: ambos incrementan el valor de numero en uno. Sin embargo, la evaluación de las dos expresiones da diferentes valores. Recuerde, si el ++ está *antes* de la variable, ésta se incrementa *antes* de devolver el valor; si el ++ está *después* de la variable, ésta se incrementa *después* de devolver el valor.

El programa del cuadro 7.10 usa el operador de incremento en un ciclo *while* para contar el número de veces que se repite el cuerpo del ciclo. Uno de los usos principales del operador de incremento es controlar la iteración de los ciclos de formas similares a como se hace en el cuadro 7.10.

Todo lo que hemos dicho acerca del operador de incremento aplica también al operador de decremento, excepto que el valor de la variable se reduce en uno en lugar de aumentar. Por ejemplo, consideremos el siguiente código:

```
int numero = 8;
int valor_producido = numero--;
cout << valor_producido << endl;
cout << numero << endl;
```

Esto produce la salida:

```
8
7
```

En cambio, el código

++ y -- sólo se pueden usar con variables

```
int numero = 8;
int valor_producido = --numero;
```

CUADRO 7.10 *El operador de incremento como expresión*

```
//Programa que cuenta calorías.
#include<iostream>
using namespace std;

int main()
{
    int numero_de_cosas, cuenta,
        calorias_individuales, calorias_totales;

    cout << "Cuantas cosas comiste hoy? ";
    cin >> numero_de_cosas;

    calorias_totales = 0;
    cuenta = 1;
    cout << "Indica cuantas calorias tiene cada una de las\n"
         << numero_de_cosas << " cosas que comiste:\n";

    while (cuenta++ <= numero_de_cosas)
    {
        cin >> calorias_individuales;
        calorias_totales = calorias_totales
                           + calorias_individuales;
    }

    cout << "Total de calorias ingeridas hoy = "
         << calorias_totales << endl;
    return 0;
}
```

Diálogo de ejemplo

```
Cuantas cosas comiste hoy? 7
Indica cuantas calorias tiene cada una de las
7 cosas que comiste:
300 60 1200 600 150 1 120
Total de calorias ingeridas hoy = 2431
```

Ejercicios de AUTOEVALUACIÓN

```
cout << valor_producido << endl;
cout << numero << endl;
```

Esto produce la salida:

```
7
7
```

`numero--` devuelve el valor de `numero` y luego decrementa `numero`; en cambio, `--numero` primero decrementa `numero` y luego devuelve el valor de `numero`.

No podemos aplicar los operadores de incremento y decremento a ninguna otra cosa que no sea una variable individual. En C++, las expresiones como `(x + y)++`, `--(x + y)`, `5++`, etcétera, carecen de validez.

25. ¿Qué salida produce lo siguiente (incorporado en un programa completo)?

```
int cuenta = 3;
while (cuenta-- > 0)
  cout << cuenta << " ";
```

26. ¿Qué salida produce lo siguiente (incorporado en un programa completo)?

```
int cuenta = 3;
while (--cuenta > 0)
  cout << cuenta << " ";
```

27. ¿Qué salida produce lo siguiente (incorporado en un programa completo)?

```
int n = 1;
do
  cout << n << " ";
while (n++ <= 3);
```

28. ¿Qué salida produce lo siguiente (incorporado en un programa completo)?

```
int n = 1;
do
 cout << n << " ";
while (++n <= 3);
```

La instrucción *for*

Las instrucciones *while* y *do-while* son todos los mecanismos cíclicos que necesitamos. De hecho, la instrucción *while* por sí sola es suficiente. Sin embargo, existe un tipo de ciclo que es tan común que C++ incluye una instrucción especial para implementarlo. Al realizar cálculos numéricos, es común efectuar un cálculo con el número 1, luego con el número 2, luego con 3, y así hasta llegar a algún valor final. Por ejemplo, si queremos sumar los números del 1 al 10, la computadora deberá ejecutar la siguiente instrucción diez veces,

siendo el valor de `n 1` la primera vez e incrementando `n` en uno en cada repetición subsecuente:

instrucción `for`

```
sumatoria = sumatoria + n;
```

La que sigue es una forma de efectuar esto con una instrucción *`while`*:

```
sumatoria = 0;
n = 1;
while (n <= 10)
{
     sumatoria = sumatoria + n;
     n++;
}
```

Aunque un ciclo *`while`* es satisfactorio aquí, este tipo de situación es precisamente para el que se diseñó la **instrucción** *`for`* (también llamada **ciclo** *`for`*). La siguiente instrucción *`for`* realiza de forma elegante la misma tarea:

```
sumatoria = 0;
for (n = 1; n <= 10; n++)
    sumatoria = sumatoria + n;
```

Examinemos esta instrucción *`for`* parte por parte.

Primero, observe que la versión de ciclo *`while`* y la de ciclo *`for`* se construyen ensamblando las mismas piezas: ambas comienzan con una instrucción de asignación que hace a la variable `sumatoria` igual a `0`. En ambos casos esta instrucción de asignación a `sumatoria` se coloca antes del principio de la instrucción cíclica propiamente dicho. Las instrucciones de ciclo se componen de las piezas

```
n = 1; n <= 10; n++ y sumatoria = sumatoria + n;
```

Estas piezas tienen la misma función en la instrucción *`for`* que en la instrucción *`while`*. La instrucción *`for`* no es más que una forma más compacta de decir la misma cosa. Aunque hay otras posibilidades, sólo usaremos instrucciones *`for`* para realizar ciclos controlados por una variable. En nuestro ejemplo, esa variable es `n`. Guiados por la equivalencia de los dos ciclos anteriores, repasemos las reglas para escribir una instrucción *`for`*.

Una instrucción `for` comienza con la palabra clave `for` seguida de tres cosas encerradas en paréntesis que le indican a la computadora qué debe hacer con la variable controladora. El principio de una instrucción *`for`* tiene esta sintaxis:

```
for (Accion_de_Inicializacion; Expresion_Booleana; Accion_de_Actualizacion)
```

La primera expresión nos dice cómo se inicializa la variable, la segunda da una expresión booleana que sirve para verificar si el ciclo debe terminar o no, y la última expresión nos dice cómo se actualiza la variable de control después de cada iteración del cuerpo del ciclo. Por ejemplo, el ciclo *`for`* anterior comienza con

```
for (n = 1; n <= 10; n++)
```

El `n = 1` dice que `n` se inicializa con `1`. El `n <= 10` dice que el ciclo seguirá iterando su

CUADRO 7.11 *La instrucción* `for`

Instrucción *for*

Sintaxis

```
for (Accion_de_Inicializacion; Expresion_Booleana; Accion_de_Actualizacion)
     Instruccion_del_Cuerpo
```

Ejemplo

```
for (numero = 100; numero >= 0; numero--)
     cout << numero
          << " botellas de cerveza en la repisa.\n";
```

Ciclo *while* **equivalente**

Sintaxis equivalente

```
Accion_de_Inicializacion;
while (Expresion_Booleana)
{
    Instruccion_del_Cuerpo
    Accion_de_Actualizacion;
}
```

Ejemplo equivalente

```
numero = 100;
while (numero >= 0)

    cout << numero
         << " botellas de cerveza en la repisa.\n";
    numero--;
}
```

Salida

```
100 botellas de cerveza en la repisa.
99 botellas de cerveza en la repisa.
              .
              .
              .
0 botellas de cerveza en la repisa.
```

CUADRO 7.12 *Una instrucción* `for`

```
//Ilustra un ciclo for.
#include <iostream>
using namespace std;

int main()
{
    int sumatoria = 0;

    for (int n = 1; n <= 10; n++)  //Observe que la variable n es local
         sumatoria = sumatoria + n;  //respecto al cuerpo del ciclo for.

    cout << "La sumatoria de los numeros 1 al 10 es "
         << sumatoria << endl;
    return 0;
}
```

Acción de inicialización.

Repetir el ciclo en tanto esto sea verdad.

Se efectúa después de cada iteración del cuerpo del ciclo.

Salida

```
La sumatoria de los numeros 1 al 10 es 55
```

cuerpo en tanto `n` sea menor o igual que `10`. La última expresión, `n++`, dice que `n` se incrementa en uno después de cada ejecución del cuerpo del ciclo.

declaración de variables dentro de una instrucción `for`

Las tres expresiones que están al principio de una instrucción *`for`* se separan con dos, y sólo dos, signos de punto y coma. No sucumba a la tentación de colocar un signo de punto y coma después de la tercera expresión. (La explicación técnica es que estas tres cosas son expresiones, no instrucciones, y por tanto no requieren un punto y coma al final.)

El cuadro 7.11 muestra la sintaxis de una instrucción *`for`* y también describe la acción de la instrucción mostrando cómo se traduce a una instrucción *`while`* equivalente. Observe que en una instrucción *`for`*, igual que en la instrucción *`while`* correspondiente, la condición para detenerse se prueba antes de la primera iteración del ciclo. Por tanto, es posible tener un ciclo *`for`* cuyo cuerpo se ejecute cero veces.

El cuadro 7.12 muestra un ejemplo de una instrucción *`for`* incrustada en un programa completo (aunque muy sencillo). La instrucción *`for`* del cuadro 7.12 es similar a la que acabamos de ver, pero tiene una característica nueva. La variable `n` se declara cuando se inicializa con `1`. Por tanto, la declaración de `n` está dentro de la instrucción *`for`*. La acción de inicialización de una instrucción *`for`* puede incluir una declaración de variable. Cuando una variable sólo se usa dentro de la instrucción *`for`*, ésta podría ser el mejor lugar para declarar la variable. Sin embargo, si la variable también se usa fuera de la instrucción *`for`*, lo mejor es declararla afuera de dicha instrucción.

El estándar ANSI C++ requiere que un compilador de C++ que pretenda cumplir con el estándar trate cualquier declaración hecha en un inicializador de ciclo *`for`* como si fuera local respecto al cuerpo del ciclo. Los compiladores de C++ anteriores no hacían es-

más posibles acciones de actualización

to. Le recomendamos averiguar cómo su compilador trata las variables declaradas en un inicializador de ciclo `for`. En aras de la transportabilidad, recomendamos no escribir código que dependa de este comportamiento. El estándar ANSI C++ requiere que las variables declaradas en la expresión de inicialización de un ciclo `for` sean locales respecto al bloque del ciclo `for`. Es probable que la siguiente generación de compiladores de C++ cumpla con esta regla, pero los compiladores actuales podrían hacerlo o no.

Nuestra descripción de la instrucción `for` fue un poco menos general que lo que en realidad se permite. Las tres expresiones al principio de una instrucción `for` pueden ser cualesquier expresiones C++, y por tanto podrían incluir más (¡o incluso menos!) de una variable. Sin embargo, nuestras instrucciones `for` siempre usarán una sola variable en estas expresiones.

En la instrucción `for` del cuadro 7.12 el cuerpo es una instrucción de asignación sencilla:

```
sumatoria = sumatoria + n;
```

El cuerpo puede ser cualquier instrucción. En particular, el cuerpo puede ser una instrucción compuesta. Esto nos permite colocar varias instrucciones en el cuerpo de un ciclo `for`, como se muestra en el cuadro 7.13.

Hasta ahora hemos visto ciclos `for` que incrementan la variable de control del ciclo en uno después de cada iteración del ciclo, y también ciclos `for` que decrementan la variable de control del ciclo en uno después de cada iteración. Puede haber muchos otros tipos de actualizaciones de la variable de control. Ésta podría incrementarse o decrementarse en `2` o en `3` o en cualquier número. Si la variable es de tipo `double`, puede incrementarse o decrementarse en una cantidad fraccionaria. Todos los ciclos `for` que siguen son válidos:

```
int n;
for (n = 1; n <= 10; n = n + 2)
    cout << "n es ahora igual a " << n << endl;
```

RIESGO *Punto y coma extra en una instrucción* `for`

No coloque un signo de punto y coma después del paréntesis al principio de un ciclo `for`. Para ver lo que puede suceder, considere el siguiente ciclo `for`:

```
for (int cuenta = 1; cuenta <=10; cuenta++);
    cout << "Hola\n";
```

Punto y coma problemático.

Si no nos percatáramos del punto y coma extra, podríamos esperar que este ciclo `for` escribiera `Hola` en la pantalla diez veces. Si notamos el punto y coma, podríamos esperar que el compilador generara un mensaje de error. No sucede ninguna de las dos cosas. Si incrustamos este ciclo `for` en un programa completo, el compilador no protestará. Si ejecutamos el programa, sólo se desplegará un `Hola` en lugar de diez. ¿Qué está sucediendo? Para contestar esta pregunta, necesitamos ciertos antecedentes.

CUADRO 7.13 *Ciclo* `for` *con cuerpo de múltiples instrucciones*

Sintaxis

```
for (Accion_de_Inicializacion; Expresion_Booleana ; Accion_de_Actualizacion)
{
      Instruccion_1
      Instruccion_2
            .
            .
            .
      Instruccion_Ultimo
}
```

Cuerpo

Ejemplo

```
for (int numero = 100; numero >= 0; numero--)
{
    cout << numero
         << " botellas de cerveza en la repisa.\n";
    if (numero > 0)
        cout << "Baja una y pasala a tu compañero.\n";
}
```

Una forma de crear una instrucción en C++ es poner un punto y coma después de algo. Si ponemos un punto y coma después de `x++`, transformamos la expresión

```
x++
```

en la instrucción

```
x++;
```

Si colocamos un punto y coma después de nada, también creamos una instrucción. Así pues, un signo de punto y coma por sí solo es una instrucción, llamada **instrucción vacía** o **instrucción nula**. La instrucción vacía no realiza ninguna acción, pero no por ello deja de ser una instrucción. Por tanto, lo que sigue es un ciclo `for` completo y válido, cuyo cuerpo es la instrucción vacía:

instrucción vacía

```
for (int cuenta = 1; cuenta <= 10; cuenta++);
```

Este ciclo `for` sí se repite diez veces, pero dado que su cuerpo es la instrucción vacía, nada sucede. El ciclo no hace nada, ¡y lo hace 10 veces!

Regresemos ahora y consideremos la línea del ciclo *for* rotulada *punto y coma problemático*. A causa del punto y coma adicional, el código inicia con un ciclo *for* que tiene un cuerpo vacío y, como acabamos de ver, ese ciclo *for* no hace nada. Una vez completado el ciclo *for*, se ejecuta el instrucción cout que sigue, el cual escribe Hola en la pantalla una vez:

```
cout << "Hola\n";
```

Los ciclos *for* con cuerpos vacíos pueden llegar a tener algún uso, pero a estas alturas un ciclo semejante con toda probabilidad no es más que un error por descuido.

```
for (n = 0; n > -100; n = n - 7)
    cout << "n es ahora igual a " << n << endl;

for (double magn = 0.75; magn <= 5; magn = magn + 0.05)
    cout << "magn es ahora igual a " << magn << endl;
```

La actualización ni siquiera tiene que ser una suma o una resta. Es más, la inicialización no tiene que hacer simplemente una variable igual a una constante. Podemos inicializar y modificar una variable de control de ciclo de prácticamente cualquier forma que nos plazca. Por ejemplo, lo que sigue demuestra otra forma de iniciar un ciclo *for*:

```
for (double x = pow(y, 3.0); x > 2.0; x = sqrt(x))
    cout << "x es ahora igual a " << x << endl;
```

¿Qué tipo de ciclo debemos usar?

Al diseñar un ciclo, lo mejor es posponer hasta el final de la etapa de diseño la decisión respecto a qué instrucción cíclica de C++ conviene usar. Primero hay que diseñar el ciclo usando pseudocódigo, y luego traducimos el pseudocódigo a código C++. En ese momento se-

Ejercicios de AUTOEVALUACIÓN

rá fácil decidir qué tipo de instrucción cíclica C++ conviene usar.

Si el ciclo implica un cálculo numérico empleando una variable que se modifica en cantidades iguales en cada iteración del ciclo, lo mejor es usar un ciclo *for*. De hecho, siempre que un ciclo realice un cálculo numérico hay que considerar el uso de un ciclo *for*. No siempre será lo más apropiado, pero suele ser el ciclo más claro y fácil de usar para cálculos numéricos.

En casi todos los demás casos conviene usar un ciclo *while* o un ciclo *do-while*; es fácil decidir cuál de estos dos hay que usar. Si queremos insistir en que el cuerpo del ciclo se ejecute por lo menos una vez, podemos usar un ciclo *do-while*. Si hay circunstancias en las que el cuerpo del ciclo no deba ejecutarse ni siquiera una vez, hay que usar un ciclo *while*. Una situación común que exige un ciclo *while* es la lectura de entradas cuando existe la posibilidad de que no haya datos. Por ejemplo, si el programa lee una lista de calificaciones de examen, podría haber casos de estudiantes que no han hecho ningún examen, y el ciclo de entrada podría encontrarse con una lista

vacía. Esto exige un ciclo *while*.

29. ¿Qué salidas produce lo que sigue (incrustado en un programa completo)?

```
for (int cuenta = 1; cuenta < 5; cuenta++)
    cout << (2 * cuenta) << " ";
```

30. ¿Qué salidas produce lo que sigue (incrustado en un programa completo)?

```
for (int n = 10; n > 0; n = n - 2)
{
    cout << "Hola ";
    cout << n << endl;
}
```

31. ¿Qué salidas produce lo que sigue (incrustado en un programa completo)?

```
for (double muestra = 2; muestra > 0; muestra = muestra - 0.5)
    cout << muestra << " ";
```

32. Para cada una de las situaciones siguientes, indique qué tipo de ciclo (*while*, *do-while* o *for*) funcionaría mejor:

    a) Obtener la sumatoria de una serie, digamos 1/2 + 1/3 + 1/4 + 1/5 + . . . + 1/10.

    b) Leer la lista de calificaciones de examen de un estudiante.

    c) Leer el número de días de incapacidad tomados por los empleados de un departamento.

    d) Probar una función para ver cómo funciona con diferentes valores de sus argumentos.

33. Reescriba los siguientes ciclos como ciclos *for*:

    a)
    ```
    int i = 1;
    while (i <= 10)
    {
       if (i < 5 && i != 2)
          cout << 'X';
       i++;
    }
    ```

    b)
    ```
    int i = 1;
    while (i <=10)
    {
       cout << 'X';
       i = i + 3;
    }
    ```

    c)
    ```
    long m = 1000;
    do
    {
       cout << 'X';
       m = m + 100;
    } while (m < 1000);
    ```

34. ¿Qué salida produce este ciclo? Identifique la relación entre el valor de `n` y el valor de la variable `log`.

```
int n = 1024;
```

```
int log = 0;
for (int i = 1; i < n; i = i * 2)
    log++;
cout << n << " " << log << endl;
```

35. ¿Qué salida produce este ciclo? Comente el código.

RIESGO *Variables no inicializadas y ciclos infinitos*

Cuando presentamos inicialmente los ciclos `while` y `do-while` simples en el capítulo 2 advertimos al lector de dos riesgos asociados a los ciclos. Dijimos que hay que asegurarse de que se inicialicen todas las variables que deben tener un valor en el ciclo (es decir, que se les dé un valor) antes de ejecutarse el ciclo. Esto parece obvio dicho en forma abstracta, pero en la práctica es fácil embeberse tanto en el diseño de los ciclos que se nos olvide inicializar variables antes del ciclo. También dijimos que debemos cuidarnos de los ciclos infinitos. Ambas advertencias aplican igualmente a los ciclos `for`.

break;

```
int n = 1024;
int log = 0;
for (int i = 1; i < n; i = i * 2);
   log++;
cout << n << " " << log << endl;
```

36. ¿Qué salida produce este ciclo? Comente el código.

```
int n = 1024;
int log = 0;
for (int i = 0; i < n; i = i * 2)
```

La instrucción `break`

La instrucción `break` puede servir para salir de una instrucción cíclica. Cuando se ejecuta la instrucción `break`, la instrucción cíclica termina de inmediato y la ejecución continúa con la instrucción que sigue a la instrucción cíclica. La instrucción `break` se puede usar en cualquier forma de ciclo: en un ciclo `while`, en un ciclo `do-while` o en un ciclo `for`. Ésta es la misma instrucción `break` que usamos en las instrucciones `switch`.

RIESGO *La instrucción `break` en ciclos anidados*

Una instrucción `break` sólo termina el ciclo más interior que la contiene. Si tenemos un ciclo dentro de un ciclo y una instrucción `break` en el ciclo interior, dicha instrucción sólo terminará el ciclo interior.

CUADRO 7.14 *Una instrucción* `break` *en un ciclo*

```
//Obtiene la sumatoria de una lista de 10 números negativos.
#include<iostream>
using namespace std;

int main()
{
    int numero, suma = 0, cuenta = 0;
    cout << "Escribe 10 numeros negativos:\n";

    while (++cuenta <= 10)
    {
        cin >> numero;

        if (numero >= 0)
        {
            cout << "ERROR: Se introdujo un"
                 << " numero positivo o cero como\n"
                 << cuenta << "o. numero. La entrada termina"
                 << " con el " << cuenta << "o. numero.\n"
                 << "El " << cuenta << "o. numero no se sumo.\n";
            break;
        }

        suma = suma + numero;
    }

    cout << suma << " es la suma de los "
         << (cuenta - 1) << " primeros numeros.\n";
    return 0;
}
```

Diálogo de ejemplo

```
Escribe 10 numeros negativos:
-1 -2 -3 4 -5 -6 -7 -8 -9 -10
ERROR: Se introdujo un numero positivo o cero como
4o. numero. La entrada termina con el 4o. numero.
El 4o. numero no se sumo.
-6 es la suma de los 3 primeros numeros.
```

Ejercicios de AUTOEVALUACIÓN

```
    log++;
cout << n << " " << log << endl;
```

La instrucción *break*

Ya usamos la instrucción *break* como forma de terminar un instrucción *switch*. Esa misma instrucción *break* puede servir para salir de un ciclo. Hay ocasiones en que nos interesa salir de un ciclo antes de que termine de la forma normal. Por ejemplo, el ciclo podría contener una verificación de entradas indebidas y, si se recibe una entrada indebida, podríamos querer terminar de inmediato el ciclo. El código del cuadro 7.14 lee una lista de números negativos y calcula su sumatoria como valor de la variable `suma`. El ciclo termina normalmente si el usuario teclea diez números negativos, pero si el usuario olvida un signo de menos el cálculo se echa a perder y el ciclo termina de inmediato cuando se ejecuta la instrucción *break*.

37. ¿Qué salida produce lo siguiente (si está incrustado en un programa completo)?

```
int n = 5;
while (--n > 0)
{
    if (n == 2)
        break;
    cout << n << " ";
}
cout << "Fin del ciclo.";
```

38. ¿Qué salida produce lo siguiente (si está incrustado en un programa completo)?

```
int n = 5;
while (--n > 0)
{
    if (n == 2)
        exit(0);
    cout << n << " ";
}
cout << "Fin del ciclo.";
```

39. ¿Qué hace una instrucción *break*;? ¿Dónde es válido colocar una instrucción *break*;?

7.4 Diseño de ciclos

sumas

Alla vá, dando vueltas y vueltas, y nadie sabe dónde parará.
Grito tradicional de un anunciador de feria

Al diseñar un ciclo, necesitamos diseñar tres cosas:

1. El cuerpo del ciclo.
2. Las instrucciones inicializadoras.
3. Las condiciones para terminar el ciclo.

Comenzaremos con una sección que trata dos tareas cíclicas comunes y mostraremos cómo diseñar estos tres elementos para cada una de las dos tareas.

Ciclos para sumas y productos

Muchas tareas comunes implican leer una lista de números y calcular su sumatoria. Si sabemos cuántos números hay, podemos realizar la tarea fácilmente con el siguiente pseudocódigo. El valor de la variable `tantas` es la cantidad de números que hay que sumar. La sumatoria se acumula en la variable `suma`.

```
suma = 0;
repetir lo que sigue tantas veces:
    cin >> siguiente;
    suma = suma + siguiente;
fin del ciclo.
```

Este pseudocódigo se puede implementar fácilmente con el siguiente ciclo *for*:

```
int suma = 0;
for (int cuenta = 1; cuenta <= tantas; cuenta++)
{
     cin >> siguiente;
```

Repetir "tantas" veces

Podemos usar una instrucción *for* para crear un ciclo que repita su cuerpo un número predeterminado de veces:

Pseudocódigo

Repetir lo que sigue *Tantas* veces:
    Cuerpo_del_Ciclo

Instrucción *for* *equivalente*

```
for (int cuenta = 1; cuenta <= Tantas; cuenta++)
    Cuerpo_del_Ciclo
```

Ejemplo

```
for (int cuenta = 1; cuenta <= 3; cuenta++)
    cout << "Viva Mexico!\n";
```

productos

```
        suma = suma + siguiente;
}
```

Observe que se espera que la variable `suma` tenga un valor cuando se ejecuta la siguiente instrucción del cuerpo del ciclo:

```
suma = suma + siguiente;
```

Puesto que `suma` debe tener un valor desde la primera vez que se ejecuta esta instrucción, se debe inicializar con algún valor antes de que se ejecute el ciclo. Para determinar cuál es el valor de inicialización correcto para `suma`, pensemos en lo que queremos que suceda después de una iteración del ciclo. Después de sumar el primer número, el valor de `suma` deberá ser ese número. Es decir, la primera vez que se ejecute el ciclo el valor de `suma + siguiente` deberá ser igual a `siguiente`. Para que esto suceda, el valor inicial de `suma` debe ser `0`.

Podemos obtener el producto de una lista de números de forma similar a como obtuvimos la sumatoria de una lista de números. Ilustramos la técnica con el siguiente código:

ciclos de entrada

```
int producto = 1;
for (int cuenta = 1; cuenta <= tantas; cuenta++)
{
    cin >> siguiente;
    producto = producto * siguiente;
}
```

La variable `producto` debe recibir un valor inicial. No hay que suponer que todas las variables se deben inicializar a cero. Si inicializáramos `producto` a cero, seguiría siendo cero después de terminar el ciclo anterior. Como indica el código C++ anterior, el valor de inicialización correcto para `producto` es `1`. Para ver por qué `1` es el valor inicial correcto, observe que la primera vez que se ejecute el ciclo ese valor hace que `producto` tenga el valor del primer número leído, que es lo que buscamos.

lista encabezada por tamaño

Cómo terminar un ciclo

preguntar antes de iterar

Se usan comúnmente cuatro métodos para terminar un ciclo de entrada; los examinaremos en orden.

1. Lista encabezada por tamaño.
2. Preguntar antes de iterar.
3. Lista que termina con un valor centinela.
4. Quedarse sin entradas.

Si nuestro programa puede determinar con antelación el tamaño de una lista de entrada, sea preguntando al usuario o por algún otro método, podemos usar un ciclo de "repetir *n* veces" para leer entradas exactamente *n* veces, donde *n* es el tamaño de la lista. Este método se denomina **lista encabezada por tamaño**.

El segundo método para terminar un ciclo de entrada consiste simplemente en preguntar al usuario, después de cada iteración, si el ciclo se debe repetir otra vez o no. Por ejemplo,

```
suma = 0;
cout << "Hay numeros en la lista? (Escriba\n"
     << "S e Intro si Si, N e Intro si No): ";
char respuesta;
cin >> respuesta;
while((respuesta == 'S') || (respuesta == 's'))
{
    cout << "Escriba un numero: ";
    cin >> numero;
    suma = suma + numero;
    cout << "Hay mas números? (Escriba\n"
         << "S si Si, N si No. Termine con Intro.): ";
    cin >> respuesta;
}
```

valor centinela

Sin embargo, cuando se va a leer una lista larga esto resulta muy tedioso para el usuario. Imagine teclear una lista de 100 números de esta manera. Es probable que el usuario pase de contento a burlón y luego a molesto y sarcástico. Al leer una lista larga es preferible incluir sólo una señal para detenerse, lo cual es el método que veremos ahora.

Tal vez la forma más agradable de terminar un ciclo que lee una lista de valores del teclado es con un *valor centinela*. Un **valor centinela** es un valor distinto de cualquier posible valor de la lista que se va a leer, por lo cual puede servir para indicar el final de la lista. Por ejemplo, si el ciclo lee una lista de números positivos, puede usarse un número negativo como valor centinela para indicar el final de la lista. Podríamos usar un ciclo como el que sigue para obtener la sumatoria de una lista de números no negativos:

```
cout << "Escriba una lista de enteros no negativos.\n"
     << Escriba un entero negativo despues de la lista.\n";
suma = 0;
cin >> numero;
while (numero >= 0)
{
    suma = suma + numero;
    cin >> numero;
}
```

ya no hay más entradas

Tome nota de que el último número de la lista se lee pero no se suma a `suma`. Para sumar los números `1`, `2` y `3`, el usuario anexa un número negativo al final de la lista así:

1 2 3 −1

El `−1` final se lee pero no se incluye en la sumatoria.

Para usar un valor centinela de esta forma, necesitamos tener la seguridad de que hay por lo menos un valor del tipo de datos en cuestión que nunca aparecerá en la lista de valores de entrada, y podrá usarse como valor centinela. Si la lista consiste en enteros que podrían tener cualquier valor, no quedará ningún valor que sirva como valor centinela. En esta situación hay que usar algún otro método para terminar el ciclo.

ciclo controlado por contador

Al leer entradas de un archivo podemos usar un valor centinela, pero un método más común consiste simplemente en verificar si ya se leyeron todas las entradas del archivo y terminar el ciclo cuando no queden más entradas que leer. Este método para terminar un ciclo de entrada se vio en el capítulo 5 en la sección de Tip de programación intitulada "Verificación del fin de un archivo" y en la sección intitulada "La función miembro

preguntar antes de iterar

eof".

Todas las técnicas que presentamos para terminar un ciclo de entrada son casos generales de técnicas más generales que se pueden usar para terminar ciclos de cualquier especie. Las técnicas más generales son:

- Ciclos controlados por contador.
- Preguntar antes de iterar.
- Salir al cumplirse una condición de bandera.

salir por bandera

Un **ciclo controlado por contador** es cualquier ciclo que determina el número de iteraciones antes de que inicie el ciclo y que luego itera el cuerpo del ciclo ese número de veces. La técnica de lista encabezada por tamaño que vimos en el caso de los ciclos de entrada es un ejemplo de ciclo controlado por contador. Todos nuestros ciclos de "repetir tantas veces" son ciclos controlados por conteo.

Ya hablamos de la técnica de **preguntar antes de iterar**. Podemos usar esta técnica para ciclos que no sean de entrada, pero el uso más común de esta técnica es el procesamiento de entradas.

Ya antes en esta sección vimos ciclos de entrada que terminan cuando se lee un valor centinela. En nuestro ejemplo, el programa leía enteros no negativos y los colocaba en una variable llamada `numero`. Si `numero` recibía un valor negativo, ello indicaba el término de las entradas; el valor negativo era el valor centinela. Éste es un ejemplo de una técnica más general denominada **salir cuando se cumple una condición de bandera**. Una variable que cambia de valor para indicar que ocurrió algún suceso se conoce como **bandera** o **indicador**. En nuestro ciclo de entrada de ejemplo, la bandera era la variable `numero` y cuando se vuelve negativa ello indica que la lista de entrada ha terminado.

Terminar un ciclo de entrada al agotarse los datos es otro ejemplo de la técnica de salir por bandera. En este caso la condición de bandera la determina el sistema. El sistema detecta si la lectura de entradas llegó o no al final de un archivo.

También podemos usar una bandera para terminar ciclos que no sean de entrada. Por ejemplo, el siguiente ciclo puede servir para encontrar un tutor para un estudiante. Los estudiantes del grupo se numeran comenzando con el `1`. El ciclo verifica cada número de estudiante para ver si el estudiante recibió una calificación alta y se detiene tan pronto como encuentra un estudiante con una calificación alta. En este ejemplo, una calificación de 90 o más se considera alta. Suponemos que ya se definió la función `calcular_calif`.

ciclos desbocados

```
int n = 1;
calif = calcular_calif(n);
while (calif < 90)
{
    n++;
    calif = calcular_calif(n);
}
cout << "El estudiante numero " << n << " puede ser tutor.\n"
     << "Este estudiante tiene una calificacion de " << calif << endl;
```

En este ejemplo, la variable `calif` sirve como bandera.

El ciclo anterior pone de manifiesto un problema que puede surgir al diseñar ciclos. ¿Qué sucede si ningún estudiante tiene una calificación de 90 o superior? La respuesta depende de la definición de la función `calcular_calif`. Si `calcular_calif` está definida para todos los enteros positivos, podríamos tener un ciclo infinito. Peor aún, si por definición `calcular_calif` es, digamos, `100` para todos los argumentos `n` que no sean estu-

diantes, el programa podría tratar de convertir en tutor un estudiante que no existe. En todo caso, algo saldrá mal. Si hay peligro de que un ciclo se vuelva infinito o de que itere más veces de lo que sería razonable, es necesario incluir una verificación para cuidar que el ciclo no se repita demasiadas veces. Por ejemplo, una mejor condición para el ciclo anterior sería la siguiente, donde la variable `numero_de_estudiantes` se ha hecho igual al número de estudiantes del grupo.

```
int n = 1;
calif = calcular_calif(n);
while ((calif < 90) && (n < numero_de_estudiantes))
{
   n++;
   calif = calcular_calif(n);
}
if (calif >= 90)
   cout << "El estudiante numero " << n << " puede ser tutor.\n"
        << "Este estudiante tiene una calificacion de " << calif
        << endl;
else
   cout << "Ningun estudiante tiene calificacion alta.\n";
```

Ciclos anidados

El programa del cuadro 7.15 se diseñó para ayudar a vigilar la tasa de reproducción del buitre de cuello verde, una especie en peligro de extinción. En la región donde este buitre sobrevive, los conservacionistas efectúan un recuento anual del número de huevos que hay en los nidos de buitres de cuello verde. El programa del cuadro 7.15 toma los informes de cada uno de los conservacionistas de la región y calcula el número total de huevos que hay en todos los nidos observados.

Cada informe de un conservacionista consiste en una lista de números. Cada número es el conteo de huevos observados en un nido de buitre de cuello verde. La función `void obtener_un_total` lee un informe y calcula el total de huevos observados por ese conservacionista. Se ha añadido un número negativo al final de cada lista de números para que funcione como valor centinela. La llamada a la función `obtener_un_total` se incluye en un ciclo `for` de modo que se invoque una vez por cada informe recibido.

El cuerpo de un ciclo puede contener cualquier especie de instrucción, por lo que es posible tener ciclos anidados dentro de ciclos. El programa del cuadro 7.15 contiene un ci-

Convierta un cuerpo de ciclo en una llamada de función

Siempre que tenga un ciclo anidado dentro de otro, o cualquier cálculo complejo incluido en el cuerpo del ciclo, convierta el cuerpo del ciclo en una llamada de función. De esta forma podrá separar el diseño del cuerpo del ciclo del diseño del resto del programa, dividiendo la tarea de programación en dos subtareas más pequeñas.

CUADRO 7.15 *Ciclos correctamente anidados (parte 1 de 3)*

```
//Determina el total de huevos de buitre de cuello verde
//contados por todos los conservacionistas de la región.
#include<iostream>
using namespace std;

void instrucciones();

void obtener_un_total(int& total);
//Precondición: El usuario introducirá una lista de conteos
//de huevos seguida de un número negativo.
//Postcondición: total es igual a la suma de todos los conteos de huevos.

int main()
{
    instrucciones();

    int numero_de_informes;
    cout << "Cuantos informes de conservacionistas hay? ";
    cin >> numero_de_informes;

    int gran_total = 0, subtotal, cuenta;
    for (cuenta = 1; cuenta <= numero_de_informes; cuenta++)
    {
        cout << endl << "Capture el informe del "
             << "conservacionista numero " << cuenta << endl;
        obtener_un_total(subtotal);
        cout << "El conteo total de huevos para el conservacionista "
             << " numero " << cuenta << " es "
             << subtotal << endl;
        gran_total = gran_total + subtotal;
    }

    cout << endl << "Conteo total de huevos para todos los informes = "
         << gran_total << endl;

    return 0;
}
```

CUADRO 7.15 *Ciclos correctamente anidados (parte 2 de 3)*

```
//Usa iostream:
void instrucciones()
{
    cout << "Este programa resume las cifras de los informes de\n"
         << "conservacionistas sobre el buitre de cuello verde.\n"
         << "Cada informe consiste en una lista de numeros.\n"
         << "Cada numero es el conteo de los huevos\n"
         << "observados en el nido de un "
         << "buitre de cuello verde.\n"
         << "El programa calcula el numero total "
         << "de huevos en todos los nidos.\n";
}

//Usa iostream:
void obtener_un_total(int& total)
{
    cout << "Escriba el numero de huevos en cada nido.\n"
         << "Coloque un entero negativo "
         << "al final de su lista.\n";

    total = 0;
    int siguiente;
    cin >> siguiente;
    while (siguiente >= 0)
    {
       total = total + siguiente;
       cin >> siguiente;
    }
}
```

CUADRO 7.15 *Ciclos correctamente anidados (parte 3 de 3)*

Diálogo de ejemplo

```
Este programa resume las cifras de los informes de
conservacionistas sobre el buitre de cuello verde.
Cada informe consiste en una lista de numeros.
Cada numero es el conteo de los huevos
observados en el nido de un buitre de cuello verde.
El programa calcula el numero total de huevos en todos los nidos.
Cuantos informes de conservacionistas hay? 3

Capture el informe del conservacionista numero 1
Escriba el numero de huevos en cada nido.
Coloque un entero negativo al final de su lista.
1 0 0 2 -1
El conteo total de huevos para el conservacionista numero 1 es 3

Capture el informe del conservacionista numero 2
Escriba el numero de huevos en cada nido.
Coloque un entero negativo al final de su lista.
0 3 1 -1

El conteo total de huevos para el conservacionista numero 2 es 4

Capture el informe del conservacionista numero 3
Escriba el numero de huevos en cada nido.
Coloque un entero negativo al final de su lista.
-1

El conteo total de huevos para el conservacionista numero 3 es 0

Conteo total de huevos para todos los informes = 7
```

Ejercicios de AUTOEVALUACIÓN

clo dentro de un ciclo. Normalmente no pensamos que código como este contiene un ciclo anidado porque el ciclo interior está contenido dentro de una función y lo que está contenido en el ciclo anterior es una llamada a la función. Podemos aprender una lección de estos ciclos anidados de forma tan discreta: que los ciclos anidados no son diferentes de los demás ciclos. El programa del cuadro 7.16 es el resultado de reescribir el programa del cuadro 7.15 de modo que aparezcan los ciclos anidados explícitamente. El ciclo anidado del cuadro 7.16 se ejecuta una vez por cada valor de `cuenta` desde `1` hasta `numero_de_informes`. Por cada una de estas iteraciones del ciclo *`for`* exterior hay una ejecución completa del ciclo *`while`* interior.

CUADRO 7.16 *Ciclos anidados explícitamente*

```
//Determina el total de huevos de buitre de cuello verde
//contados por todos los conservacionistas de la región.
#include<iostream>
using namespace std;

void instrucciones();

int main()
{
    instrucciones();

    int numero_de_informes;
    cout << "Cuantos informes de conservacionistas hay? ";
    cin >> numero_de_informes;

    int gran_total = 0, subtotal, cuenta;
    for (cuenta = 1; cuenta <= numero_de_informes; cuenta++)
    {
        cout << endl << "Capture el informe del "
             << "conservacionista numero " << cuenta << endl;
        cout << "Escriba el numero de huevos en cada nido.\n"
             << "Coloque un entero negativo "
             << "al final de su lista.\n";
        subtotal = 0;
        int siguiente;
        cin >> siguiente;
        while (siguiente >= 0)
        {
            subtotal = subtotal + siguiente;
            cin >> siguiente;
        }
        cout << "El conteo total de huevos para el conservacionista "
             << " numero " << cuenta << " es "
             << subtotal << endl;
        gran_total = gran_total + subtotal;
    }
    cout << endl << "Conteo total de huevos para todos los informes = "
         << gran_total << endl;
    return 0;
}
```

<La definición de `instrucciones` es la misma que en el cuadro 7.15.>

Las dos versiones de nuestro programa para obtener el total de huevos de buitre de cuello verde son equivalentes. Ambos programas producen el mismo diálogo con el usuario. Sin embargo, para la mayoría de nosotros la versión del cuadro 7.15 es más fácil de entender porque el cuerpo del ciclo es una llamada de función. Al considerar el ciclo exterior debemos pensar en el cáclulo del subtotal del informe de un conservacionista como una sola operación, y no como un ciclo.

40. Escriba un ciclo que despliegue la palabra `Hola` en la pantalla 10 veces (incorporado en un programa completo).

41. Escriba un ciclo que lea una lista de números pares (digamos 2, 24, 8, 6) y calcule la sumatoria de los números de la lista. La lista debe terminar con un valor centinela. Entre otras cosas, deberá decidir qué valor(es) centinela podría usar.

errores por uno

42. Prediga la salida de los siguientes ciclos anidados:

```
int n, m;
for (n = 1; n <= 10; n++)
    for (m = 10; m >= 1; m--)
         cout << n << "por " << m
              << " = " << n*m << endl;
```

ciclos infinitos

Depuración de ciclos

Por más cuidado que se tenga al diseñar un programa, a veces se cometen equivocaciones. En el caso de los ciclos, los tipos de errores que los programadores cometen con mayor frecuencia siguen un patrón. Casi todos los errores tienen que ver con la primera o la última iteración del ciclo. Si un ciclo no funciona como usted esperaba, verifique si se iteró una vez menos o una vez más de lo debido. Decimos que los ciclos que se iteran una vez menos o más de lo debido tienen un **error por uno**, y estos errores se cuentan entre los más comunes que ocurren en los ciclos. Asegúrese de no estar confundiendo menor que con menor o igual que. Asegúrese de haber inicializado el ciclo correctamente. Recuerde que a veces un ciclo podría tener que repetirse cero veces y compruebe que su ciclo maneja esa posibilidad correctamente.

primero localice el problema

rastreo de variables

Los ciclos infinitos suelen ser resultado de un error en la expresión booleana que controla el punto en el que el ciclo se detiene. Verifique que no invirtió una desigualdad, confundiendo menor que con mayor que. Otra causa común de ciclos infinitos es terminar un ciclo con una prueba de igualdad, en lugar de una prueba con menor que o mayor que. Con valores de tipo *`double`*, una prueba de igualdad no da respuestas significativas, ya que las cantidades que se comparan son sólo valores aproximados. Incluso con valores de tipo *`int`* la igualdad puede ser una prueba peligrosa para terminar un ciclo, pues sólo hay una forma de satisfacerla.

Si revisa una y otra vez su ciclo y no puede encontrar ningún error, y el programa sigue sin comportarse debidamente, serán necesarias pruebas más avanzadas. Primero, asegúrese de que el error esté realmente en el ciclo. El hecho de que el programa no esté operando correctamente no implica que el error está donde creemos que está. Si nuestro programa está dividido en funciones, deberá ser fácil determinar la ubicación aproximada del error o errores.

código con un error

Una vez que hayamos decidido que el error está en un ciclo dado, deberemos observar cómo el ciclo modifica los valores de las variables durante la ejecución del programa. De este modo podemos ver qué está haciendo el ciclo y así ver qué está haciendo mal. Vigilar

los cambios en el valor de una variable mientras el programa se ejecuta se conoce como **rastrear** la variable. Muchos sistemas cuentan con utilerías de depuración que nos permiten rastrear fácilmente variables sin tener que modificar el programa. Si el sistema del lector cuenta con una utilería semejante, podría valer la pena aprender a usarla. Si el sistema no cuenta con una utilería de depuración, podemos rastrear una variable colocando uns instrucción `cout` temporal en el cuerpo del ciclo; así, el valor de la variable se escribirá en la pantalla en cada iteración del ciclo.

Por ejemplo, consideremos el siguiente fragmento de código, que requiere depuración:

```
int siguiente = 2, producto = 1;
while (siguiente < 5)
{
    siguiente++;
    producto = producto * siguiente;
}
//La variable producto contiene
//el producto de los números del 2 al 5.
```

El comentario al final del ciclo nos dice lo que se supone que debe hacer el ciclo, pero lo hemos probado y sabemos que da a la variable `producto` un valor incorrecto. Necesitamos averiguar qué pasa. Para ayudarnos a depurar este ciclo, rastreamos las variables `siguiente` y `producto`. Si el lector tiene una utilería de depuración, puede usarla; si no, puede rastrear las variables insertando una instrucción `cout` así:

segundo intento

```
int siguiente = 2, producto = 1;
while (siguiente < 5)
{
    siguiente++;
    producto = producto * siguiente;
    cout << "siguiente = " << siguiente
         << " producto = " << producto << endl;
}
```

Al rastrear las variables `producto` y `siguiente`, encontramos que después de la primera iteración del ciclo los valores de `producto` y `siguente` son ambos `3`. Entonces nos queda claro que sólo hemos multiplicado los números del `3` al `5` y no incluimos el `2` en el producto.

Hay al menos dos formas aceptables de corregir este error. La más fácil es inicializar la variable `siguiente` con `1` en lugar de `2`; así, cuando se incremente `siguiente` en la primera iteración del ciclo, recibirá el valor `2` en lugar de `3`. Otra forma de corregir el ciclo es colocar el incremento después de la suma, así:

```
int siguiente = 2, producto = 1;
while (siguiente < 5)
{
    producto = producto * siguiente;
    siguiente++;
}
```

Todo cambio requiere volver a probar

Supongamos que corregimos el error cambiando de lugar la instrucción `siguiente++` como acabamos de indicar. Después de efectuar la corrección, falta una cosa por hacer:

debemos probar el código modificado. Al hacerlo, veremos que todavía da un resultado incorrecto. Si rastreamos otra vez las variables, descubriremos que el ciclo termina después de multiplicar por 4, y nunca multiplica por 5. Esto nos dice que la expresión booleana aho-

Prueba de ciclos

Todos los ciclos deben probarse con entradas que causen todas las siguientes conductas del ciclo (o tantas como sea posible): cero iteraciones del cuerpo del ciclo, una iteración del cuerpo del ciclo, el número máximo de iteraciones del cuerpo del ciclo, y una menos que el número máximo de iteraciones del cuerpo del ciclo. (Éste únicamente es un conjunto mínimo de situaciones de prueba. También hay que realizar otras pruebas específicas para el ciclo que se está probando.)

ra debe llevar un signo de menor o igual que, no un signo de menor que. Por tanto, el código correcto es:

```
int siguiente = 2, producto = 1;
while (siguiente <= 5)
{
    producto = producto * siguiente;
    siguiente++;
}
```

Siempre que modifiquemos un programa, deberemos volver a probar el programa. Nunca debemos suponer que el cambio hará que el programa sea correcto. El hecho de que hayamos encontrado una cosa qué corregir no implica que hayamos encontrado todas las cosas que necesitan corregirse. Además, como ilustra este ejemplo, cuando modificamos una parte de

Depuración de un programa muy malo

Si un programa es muy malo, no trate de depurarlo. Tírelo a la basura y vuelva a comenzar desde el principio.

Ejercicios de AUTOEVALUACIÓN

un programa para corregirlo, el cambio podría requerir una modificación de otra parte del programa.

Las técnicas que hemos desarrollado nos ayudarán a encontrar los escasos errores que logran colarse en un programa bien diseñado. Sin embargo, no hay depuración que pueda convertir un programa mal diseñado en uno confiable y comprensible. Si un programa o algoritmo es muy difícil de entender o tiene un funcionamiento muy deficiente, no trate de arreglarlo. Más bien, deséchelo y comience otra vez desde el principio. El resultado será un programa más fácil de entender y con menos probabilidad de contener errores ocultos. Lo que tal vez no sea tan obvio es que al tirar a la basura el código mal diseñado y volver a comenzar produciremos un programa funcional más rápidamente que si tratáramos de reparar el código viejo. Podría parecer un desperdicio de esfuerzo tirar a la basura to-

do el código que tanto trabajo nos costó, pero ésa es la forma más eficiente de proceder. La labor que invertimos en el código desechado no se desperdicia. Las lecciones que aprendimos nos ayudan a diseñar un mejor programa más rápidamente que si no tuviéramos experiencia. Es poco probable que el código malo en sí sea muy útil.

43. ¿Qué significa rastrear una variable? ¿Cómo se rastrea una variable?

44. ¿Qué es un error por uno en un ciclo?

45. Tenemos una cerca que debe medir 100 metros de largo. Los postes de la cerca se deben colocar cada 10 metros. ¿Cuántos postes necesitamos? ¿Por qué no es tan absurda como podría parecer la presencia de este problema en un libro de programación? ¿ A cuál de los problemas a que se enfrentan los programadores enfrentan esta pregunta?

Resumen del capítulo

- Las expresiones booleanas se evalúan de forma similar a como se evalúan las expresiones aritméticas.
- Casi todos los compiladores modernos tienen un tipo `bool` cuyos valores son `true` y `false`.
- Podemos escribir una función que devuelva un valor `true` o `false`. Podemos usar una llamada a una función semejante como expresión booleana en una instrucción `if-else` o en cualquier otro lugar en el que se permita una expresión booleana.
- Una estrategia para resolver una tarea o subtarea consiste en anotar las condiciones y las acciones correspondientes que deben realizarse en cada condición. Esto puede implementarse en C++ con una instrucción `if-else` multivía.
- Una instrucción `switch` es una buena forma de implementar un menú para el usuario del programa.
- Use llamadas a funciones en instrucciones de bifurcación multivía, como las instrucciones `switch` y las instrucciones `if-else` multivía.
- Un **bloque** es una instrucción compuesta que contiene declaraciones de variables. Las variables declaradas en un bloque son locales respecto al bloque. Entre otros usos, los bloques pueden ser la acción de una bifurcación de una instrucción de bifurcación multivía, digamos una instrucción `if-else` multivía.
- Podemos usar un ciclo `for` para obtener el equivalente de la instrucción “repetir el cuerpo del ciclo `n` veces”.
- Se usan comúnmente cuatro métodos para terminar un ciclo de entrada: lista encabezada por tamaño, preguntar antes de iterar, lista terminada con un valor centinela y quedarse sin entradas.
- Por lo regular lo mejor es diseñar los ciclos en pseudocódigo que no especifique el mecanismo cíclico de C++ que se usará. Una vez diseñado el algoritmo, suele ser obvio cuál instrucción cíclica C++ conviene usar.
- Una forma de simplificar nuestro razonamiento al diseñar ciclos anidados es convertir el cuerpo del ciclo en una llamada de función.

Respuestas a los ejercicios de autoevaluación

- Siempre debemos revisar los ciclos para comprobar que las variables que se usan en el ciclo se inicialicen debidamente antes del inicio del ciclo.
- Siempre debemos revisar los ciclos para comprobar que no se repitan una vez más o una vez menos de lo debido.
- Al depurar ciclos, resulta útil rastrear las variables clave del cuerpo del ciclo.
- Si un programa o algoritmo es muy difícil de entender u opera de forma muy deficiente, no debemos tratar de corregirlo. Hay que tirarlo y volver a comenzar.

1. a) `true`.

   b) `true`. Observe que las expresiones a) y b) significan exactamente lo mismo. Puesto que los operadores == y < tienen mayor precedencia que `&&`, no necesitamos los paréntesis. Sin embargo, los paréntesis ayudan a la comprensión. Para casi todos nosotros la expresión a es más fácil de entender que la b, aunque significan lo mismo.

   c) `true`.

   d) `true`.

   e) `false`. Puesto que el valor de la primera subexpresión `(cuenta == 1)` es `false`, sabemos que toda la expresión es `false` sin molestarnos en evaluar la segunda subexpresión. Por tanto, no importa qué valores tengan `x` y `y`. Esto se denomina `evaluación en cortocircuito`, y es lo que C++ hace.

   f) `true`. Puesto que el valor de la primera subexpresión `(cuenta < 10)` es `true`, sabemos que toda la expresión es `true` sin molestarnos en evaluar la segunda subexpresión. Por tanto, no importa qué valores tengan `x` y `y`. Esto se denomina `evaluación en cortocircuito`, y es lo que C++ hace.

   g) `false`. Observe que la expresión de g incluye la expresión de f como subexpresión. Esta subexpresión se evalúa en cortocircuito, que describimos en el caso de f. Toda la expresión de g equivale a

   ```
   !( (true || (x < y)) && true )
   ```

   que a su vez equivale a `!( true && true )`, y eso equivale a `!(true)`, lo que equivale al valor final de `false`.

   h) Esta expresión produce un error cuando se evalúa porque la primera subexpresión, `((limite/cuenta) > 7)`, implica una división entre cero.

   i) `true`. Puesto que el valor de la primera subexpresión `(limite < 20)` es `true`, sabemos que toda la expresión es `true` sin molestarnos en evaluar la segunda subexpresión. Por tanto, la segunda subexpresión

   ```
   ((limite/cuenta) > 7)
   ```

   nunca se evalúa y por tanto la computadora nunca se da cuenta de que implica una división entre ce-

ro. Esto es `evaluación en cortocircuito`, y es lo que C++ hace.

j) Esta expresión produce un error cuando se evalúa porque la primera subexpresión, `((limite/cuenta) > 7)`, implica una división entre cero.

k) *false*. Puesto que el valor de la primera subexpresión `(limite < 0)` es *false*, sabemos que toda la expresión es *false* sin molestarnos en evaluar la segunda subexpresión. Por tanto, la segunda subexpresión

```
((limite/cuenta) > 7)
```

nunca se evalúa y por tanto la computadora nunca se da cuenta de que implica una división entre cero. Esto es `evaluación en cortocircuito`, y es lo que C++ hace.

l) Si cree que esta expresión es absurda, tiene razón. La expresión no tiene ningún significado intuitivo, pero C++ convierte los valores *int* en *bool* y luego evalúa las operaciones `&&` y `!`, así que C++ sí evalúa esta barbaridad. Recuerde que en C++ cualquier entero distinto de cero se convierte en *true* y `0` se convierte en *false*, así que C++ evaluará

```
(5 && 7) + (!6)
```

como sigue: en la expresión `(5 && 7)`, el `5` y el `7` se convierten en *true*; la evaluación de *true* `&&` *true* da *true*, que en C++ se convierte en `1`; en `(!6)` el `6` se convierte en *true*, así que la evaluación de `!(`*true*`)` da *false*, que C++ convierte en `0`; por tanto, la evaluación de toda la expresión da `1 + 0`, que es `1`. Por tanto, el valor final es `1`. C++ convierte el número `1` en *true*, pero esta respuesta no tiene mucho significado intuitivo como *true*, así que tal vez sea mejor decir que la respuesta es `1`.

No hay necesidad de volverse diestros en la evaluación de este tipo de expresiones absurdas, pero si evaluamos unas cuantas entenderemos mejor por qué el compilador no genera un mensaje de error cuando por error combinamos operadores numéricos y booleanos en una misma expresión.

2. Hasta aquí hemos estudiado instrucciones de bifurcación, instrucciones de iteración e instrucciones de llamada de función.

   Los ejemplos de instrucciones de bifurcación que hemos estudiado son las instrucciones *if* e *if-else*. Los ejemplos de instrucciones de iteración son las instrucciones *while* y *do-while*.

3. La expresión `2 < x < 3` es válida. No significa `(2 < x)&&(x < 3)`, como muchos quisieran; significa `(2 < x) < 3`. Puesto que `(2 < x)` es una expresión booleana, su valor es *true* o bien *false*, así que `2 < x < 3` siempre es *true*. El resultado es *true* sea cual sea el valor de `x`.

4. No. La expresión booleana `j > 0` es *false* (se acaba de asignar `−1` a `j`). El `&&` se evalúa en cortocircuito, lo que significa que no se evalúa la segunda subexpresión si el valor de verdad se puede determinar a partir de la primera subexpresión. La primera es *false*, así que la segunda no importa.

5.
```
bool en_orden(int n1, int n2, int n3)
{
     return ((n1 <= n2) && (n2 <= n3));
}
```

6.
```
bool par(int n)
```

```
{
     return ((n % 2) == 0);
}
```

7.
```
bool es Digito(char c)
{
     return ('0' <= c) && (c <= '9');
}
```

8.
```
bool esRaizde(int candidato_raiz, int numero)
{
     return (numero == candidato_raiz* candidato_raiz);
}
```

9.
```
Inicio
Saludos desde el segundo if.
Final
Otra vez
Final de nuevo
```

10.
```
grande
```

11.
```
pequeño
```

12.
```
mediano
```

13.
```
Inicio
Segunda salida
Fin
```

14. Las instrucciones son las mismas aun si la segunda expresión boleana es (`x > 10`) o (`x > 100`). Por lo tanto, la salida es la misma que en el ejercicio de autoevaluación 13.

15.
```
Inicio
100
Fin
```

16. Estas dos instrucciones son correctas:

```
if (n < 0)
    cout << n << " es menor que cero.\n";
else if ((0 <= n) && (n <= 100))
    cout << n << " esta entre 0 y 100 (inclusive).\n";
else if (n > 100)
    cout << n << " es mayor que 100.\n";
```

y

```
if (n < 0)
    cout << n << " es menor que cero.\n";
else if (n <= 100)
    cout << n << " esta entre 0 y 100 (inclusive).\n";
else
    cout << n << " es mayor que 100.\n";
```

17. Las constantes *enum* reciben valores predeterminados que comienzan con 0, a menos que se les asignen otros valores. Las constantes se incrementan en 1. La salida es `3 2 1 0`.

18. Las constantes *enum* reciben los valores asignados. Las constantes no asignadas incrementan el valor anterior en 1. La salida es `2 1 7 5`.

19.
```
Gusanos asados
```

20.
```
Helado de cebolla
```

21.
```
Helado de chocolate
Helado de cebolla
```

(La razón es que no hay instrucción *break* en el *case* 3.)

22.
```
¡Buen provecho!
```

23.
```
42 22
```

24. Si modificamos un poco el fragmento de código podremos entender mejor qué declaración de x aplica en cada uso de x:

```
{
  int x1 = 1;//  salidas en esta columna
  cout << x1 << endl;// 1<cr>
  {
    cout << x1 << endl;// 1<cr>
    int x2 = 2;
    cout << x2 << endl;// 2<cr>
    {
        cout << x2 << endl;// 2<cr>
        int x3 = 3;
        cout << x3 << endl;// 3<cr>
    }
    cout << x2 << endl;// 2<cr>
  }
  cout << x1 << endl;// 1<cr>
}
```

Aquí <*cr*> indica que la salida inicia una nueva línea.

25. 2 1 0

26. 2 1

27. 1 2 3 4

28. 1 2 3

29. 2 4 6 8

30.
```
Hola 10
Hola 8
Hola 6
Hola 4
Hola 2
```

31. 2,000000 1.500000 1.000000 0.500000

32. a) Un ciclo *for*.

    b) y c) ambas requieren un ciclo *while* porque la lista de entrada podría estar vacía.

    d) Se puede usar un ciclo *do-while* porque se realizará por lo menos una prueba.

33. a)
```
for (int i = 1; i <= 10; i++)
                        if (i < 5 && i != 2)
                           cout << 'X';
```

    b)
```
for (i = 1; i <= 10; i = i + 3)
                  cout << 'X';
```

    c)
```
cout << 'X'  // necesario para que la salida no cambie. Observe
                        // también el cambio en la inicialización de m
                        for (long m = 200; m < 1000; m ) m + 100)
                           cout << 'X';
```

34. La salida es: `1024 10`. El segundo número es el logaritmo base 2 del primer número.

35. La salida es: `1024 1`. El ';' después del *for* probablemente es un error por descuido.

36. Éste es un ciclo infinito. Considere la expresión de actualización `i = i * 2`; no puede modificar `i` porque el valor inicial de `i` es `0`, y la expresión no cambia nunca ese valor.

37. `4 3 Fin del ciclo.`

38. `4 3`

    Observe que como el instrucción `exit` termina el programa, la frase `Fin del ciclo.` no se envía a la salida.

39. Una instrucción *`break`* sirve para salir de un ciclo (una instrucción *`while`*, *`do-while`* o *`for`*) o para terminar un caso en una instrucción *`switch`*. No es válido colocar un *`break`* en ningún otro punto de un programa C++. Cabe señalar que si los ciclos están anidados, un *`break`* sólo termina un nivel del ciclo.

40.
```
for (int cuenta = 1; cuenta <= 10; cuenta++)
    cout << "Hola\n";
```

41. Puede usar cualquier número impar como valor centinela.

```
int suma = 0, siguiente;
cout << "Escriba una lista de numeros pares. Ponga\n"
     << "un numero impar al final de la lista.\n";
cin >> siguiente;
while ((siguiente % 2) == 0)
{
   suma = suma + siguiente;
   cin >> siguiente;
}
```

42. La salida es demasiado larga para reproducirla aquí. El patrón es el siguiente:

```
1 por 10 = 10
1 por 9 = 9
   .
   .
   .
1 por 1 = 1
2 por 10 = 20
2 por 9 = 18
   .
   .
   .
2 por 1 = 2
3 por 10 = 30
   .
   .
   .
```

43. *Rastrear una variable* significa vigilar los cambios de valor de una variable durante la ejecución del programa. Esto puede hacerse con recursos de depuración especiales o insertando instrucciones de salida temporales en el programa.

Los ciclos que iteran su cuerpo una vez más o una vez menos de lo debido tienen un error por uno.

45. Los errores por uno son comunes en la resolución de problemas, no sólo en la escritura de ciclos. Un razo-

namiento típico de quienes no meditan bien las cosas es

100 metros de cerca / 10 metros entre postes = 10 postes

Esto, desde luego, dejará los últimos 10 metros de cerca sin poste. Necesitamos 11 postes para tener una cerca de 100 metros con intervalos de 10 metros entre los postes.

Proyectos de programación

Escriba un programa que juzgue el juego de papel-roca-tijeras. Cada uno de los dos usuarios teclea **P**, **R** o bien **T** y el programa anuncia entonces el ganador: `Papel cubre a roca`, `Roca rompe tijeras`, `Tijeras cortan papel` o `Nadie gana`. No olvide permitir que los usuarios tecleen letras tanto minúsculas como mayúsculas. Su programa deberá incluir un ciclo que permita a los usuarios jugar otra vez hasta que indiquen que han terminado.

2. Escriba un programa que calcule los intereses a pagar, la cantidad total a pagar y el pago mínimo para una cuenta de crédito revolvente. El programa acepta el saldo de la cuenta como entrada y luego le suma los intereses para obtener la cantidad total a pagar. El plan de pagos es el siguiente: la tasa de interés es del 1.5% sobre los primeros $1000 y del 1% sobre cualquier excedente. El pago mínimo es la cantidad total a pagar si ésta es de $10 o menos; si no, es $10 o el 10% de la cantidad total que se debe, lo que sea mayor. El programa deberá incluir un ciclo que permita al usuario repetir este cálculo hasta que indique que ya terminó.

3. Escriba un programa de astrología. El usuario introduce un cumpleaños y el programa responde con el signo y el horóscopo para ese cumpleaños. El mes se puede introducir como un número entre uno y doce. Utilice la sección de horóscopos de un diario para obtener los horóscopos y las fechas que corresponden a cada signo. Luego mejore su programa de modo que si el cumpleaños sólo está a uno o dos días de distancia de un signo adyacente el programa anuncie que el cumpleaños está en una "cúspide" y también despliegue el horóscopo para ese signo adyacente más cercano. Este programa tendrá una bifurcación multivía larga. Almacene los horóscopos en un archivo. Si está haciendo este proyecto como tarea, pregunte a su profesor si hay instrucciones especiales respecto al nombre del archivo o su ubicación. El programa deberá incluir un ciclo que permita al usuario repetir este cálculo hasta que indique que ya terminó.

   Los signos de los horóscopos y sus fechas son:

Aries	marzo 21 - abril 19
Tauro	abril 20 – mayo 20
Géminis	mayo 21 – junio 21
Cáncer	junio 22 – julio 22
Leo	julio 23 – agosto 22
Virgo	agosto 23 – septiembre 22
Libra	septiembre 23 –octubre 22
Escorpión	octubre 23 – noviembre 21
Sagitario	noviembre 22 – diciembre 21
Capricornio	diciembre 22 – enero 19
Acuario	enero 20 – febrero 18
Piscis	febrero 19 – marzo 20

4. Los signos zodiacales del mismo elemento son muy compatibles. Hay 4 elementos en la astrología y 3 signos en cada uno: FUEGO (Aries, Leo, Sagitario), TIERRA (Tauro, Virgo, Capricornio), AIRE (Géminis, Libra, Acuario), AGUA (Cáncer, Escorpión, Piscis).

Según algunos astrólogos la compatibilidad es mayor con los del mismo signo y con los otros dos elementos del mismo elemento. Es decir, Aries es compatible con otro Aries y con los otros dos signos de Fuego: Leo y Sagitario.

Modifique el programa del proyecto de programación 3 de manera que también despliegue los nombres de los signos con los que será compatible de acuerdo con la fecha de nacimiento.

5. Escriba un programa que calcule el costo de una llamada de larga distancia. El costo de la llamada se determina según el siguiente plan de tarifas:

   a) Cualquier llamada iniciada entre las 8:00 A.M. y las 6:00 P.M., de lunes a viernes, se cobra a razón de $0.40 por minuto.

   b) Cualquier llamada iniciada antes de las 8:00 A.M. o después de las 6:00 P.M., de lunes a viernes, se cobra a razón de $0.25 por minuto.

   c) Cualquier llamada iniciada en un sábado o un domingo se cobra a razón de $0.15 por minuto.

   Las entradas consistirán en el día de la semana, la hora en que se inició la llamada y la duración de la llamada en minutos. La salida será el costo de la llamada. La hora se debe introducir en notación de 24 horas, de modo que la hora 1:30 P.M. se introduce como

   13:30

   El día de la semana se leerá como uno de los siguientes pares de valores de carácter, que se almacenan en dos variables de tipo `char`:

   Lu Ma Mi Ju Vi Sa Do

   Asegúrese de que el usuario pueda usar letras mayúsculas o minúsculas o una combinación de las dos. El número de minutos se introducirá como un valor de tipo `int`. (Puede suponer que el usuario redondea el dato a un número entero de minutos.) El programa deberá incluir un ciclo que permita al usuario repetir este cálculo hasta que indique que ya terminó. Una vez que haya depurado totalmente su programa, produzca una variación que lea de un archivo la información de todas las llamadas telefónicas hechas en una semana y luego escriba una factura telefónica en otro archivo. La factura deberá mostrar todas las llamadas y lo que se cobra por cada llamada, así como el monto total de la factura. Enumere las llamadas telefónicas en el archivo de salida en el mismo orden en que aparecen en el archivo de entrada. Si está haciendo este proyecto como tarea, pregunte a su profesor si hay instrucciones especiales respecto a los nombres de archivo.

6. (Este proyecto requiere que usted conozca las bases de número complejos, por lo que sólo será apropiado si ha estudiado números complejos en clases previas de matemáticas.)

   Escriba un programa en C++ que resuelva la ecuación cuadrática para encontrar las raíces. Las raíces de la ecuación cuadrática:

   $ax^2 + bx + c = 0$

   (donde a no es cero) son dadas por la fórmula

   $(-b \pm \text{sqrt}(b^2 - 4ac)) / 2a$

   El valor del discriminador (b^2-4ac) determina la naturaleza de las raíces. Si el valor de éste es cero, entonces la ecuación tiene una sola raíz real. Si el valor del discriminante es positivo, entonces la ecuación tiene dos raíces reales. Pero si el valor es negativo, entonces la ecuación tiene dos raíces complejas.

   El programa toma como entrada los valores de a, b y c y como salida a las raíces. Sea creativo en cuanto a la salida de las raíces complejas.

Incluya un ciclo que le permita al usuario repetir este cálculo para nuevos valores de entrada hasta que quiera finalizar el programa.

7. Escriba un programa que acepte un año escrito como una cifra de cuatro dígitos arábigos (los ordinarios) y despliegue el año escrito en números romanos. Los números romanos importantes son: V para 5, X para 10, L para 50, C para 100, D para 500 y M para 1000. Recuerde que algunos números se forman empleando una especie de resta de un "dígito" romano; por ejemplo, IV es 4 que se produce como V menos I, XL es 40, CM es 900, etcétera. He aquí algunos ejemplos de años: MCM es 1900, MCML es 1950, MCMLX es 1960, MCMXL es 1940, MCMLXXXIX es 1989. Suponga que el año está entre 1000 y 3000. El programa deberá incluir un ciclo que permita al usuario repetir este cálculo hasta que indique que ya terminó.

8. Escriba un programa que juzgue una mano de veintiuno. En este juego de naipes, un jugador recibe entre dos y cinco naipes. (El jugador decide cuántos, pero eso no afecta este ejercicio.) Los naipes del 2 al 10 valen los puntos que indican: de 2 a 10. Los naipes de "mono" —jack, reina y rey— valen 10 puntos. El objetivo es acercarse lo más posible a un puntaje de 21 sin pasarse de 21 (si el puntaje rebasa 21 se dice que el jugador ha "tronado"). El as puede contar como 1 o como 11, lo que más convenga al jugador. Por ejemplo, un as y un 10 pueden juzgarse como 11 o como 21. Puesto que 21 es un mejor puntaje, esta mano se juzga como 21. Un as y dos ochos se pueden juzgar como 17 o como 27. Puesto que con 27 el jugador "truena", esta mano se juzga como 17. El programa pregunta al usuario cuántos naipes tiene y el usuario responde con un entero entre 2 y 5. Luego se pide al usuario los valores de los naipes, que son del 2 al 10, jack, reina, rey y as. Una buena forma de manejar las entradas es usar el tipo *char* de modo que un naipe 2, por ejemplo, se lea como el carácter `'2'` y no como el número `2`. Introduzca los valores del 2 al 9 como los caracteres `'2'` a `'9'`. Introduzca los valores 10, jack, reina, rey y as como los caracteres `'d'`, `'j'`, `'q'`, `'k'` y `'a'`. (Desde luego, el usuario no teclea los apóstrofos.) Asegúrese de permitir letras tanto mayúsculas como minúsculas.

   Después de leer los valores, el programa deberá convertir los valores de carácter en puntajes numéricos, teniendo especial cuidado con los ases. La salida es un número entre 2 y 21 (inclusive) o la palabra `Tronó`. Use muchas funciones. Es probable que tenga una o más bifurcaciones multivía largas que usen una instrucción *switch* o una instrucción *if-else* anidada. El programa deberá incluir un ciclo que permita al usuario repetir este cálculo hasta que indique que ya terminó.

9. Los intereses de un préstamo se pagan sobre saldos insolutos (la parte del préstamo original que falta por saldar), así que un préstamo con una tasa de interés de, digamos, el 14% puede costar mucho menos que el 14% del préstamo original. Escriba un programa que reciba un monto de préstamo y una tasa de interés como entradas y luego `exhiba` los pagos mensuales y el saldo del préstamo hasta que el préstamo queda saldado. Suponga que los pagos mensuales son una vigésima parte del monto del préstamo original, y que cualquier excedente respecto a los intereses reduce el saldo por pagar. Por tanto, con un préstamo de $20,000, los pagos serían de $1000 al mes. Si la tasa de interés es del 10% anual, cada mes los intereses serán una doceava parte del 10% del saldo insoluto. El primer mes se pagarán intereses de (10% de $20,000)/12 = $166.67, y los $833.33 restantes reducirán el saldo a $19,166.67. El siguiente mes los intereses serán (10% de $19,166.67)/12, y así. Además, haga que el programa `exhiba` el total de intereses pagados durante la vigencia del préstamo.

   Por último, determine qué porcentaje simple anualizado del monto del préstamo original se pagó por concepto de intereses. Por ejemplo, si se pagó $1000 de intereses por un préstamo de $10,000, el cual tardó dos años en saldarse, los intereses anualizados son de $500, lo cual es el 5% del monto del préstamo de $10,000. Si hace este proyecto como tarea, pregunte a su profesor si las entradas y salidas deben efectuarse con el teclado y la pantalla o con archivos. Si las entradas y salidas usan el teclado y la pantalla, el programa deberá permitir al usuario repetir el cálculo cuantas veces desee.

10. Los números de Fibonacci F_n se definen como sigue: F_0 es 1, F_1 es 1, y

$F_{i+2} = F_i + F_{i+1}$

i = 0, 1, 2, ... En otras palabras, cada número es la suma de los dos números anteriores. Los primeros números de Fibonacci son 1, 1, 2, 3, 5 y 8. Un lugar en que ocurren estos números es en ciertas tasas de crecimiento de la población. Si una población no tiene decesos, la serie muestra el tamaño de la población después de cada periodo. Un organismo requiere dos periodos para madurar hasta la edad reproductiva, y luego el organismo se reproduce una vez en cada periodo. La aplicación más directa de la fórmula es a la reproducción asexual con una tasa de un vástago por periodo.

Por ejemplo, la población de "lama verde" crece según esta tasa y tiene un periodo de cinco días. Por tanto, si una población de lama inicia como 10 kilogramos de lama, en cinco días todavía habrá 10 kilogramos de lama; en 10 días habrá 20 kg de lama, en quince, 30 kg, en veinte días, 50 kg, etcétera. Escriba un programa que reciba el tamaño inicial de una población de lama verde (en kg) y un número de días como entradas, y que `exhiba` los kilogramos de lama verde que habrá después de ese número de días. Suponga que el tamaño de la población es el mismo durante cuatro días y en el quinto día aumenta. El programa deberá permitir al usuario repetir este cálculo cuantas veces desee.

Versión alternativa con archivos: Una versión alternativa lee sus entradas de un archivo y escribe sus salidas en otro archivo. En esta versión el programa lee el tamaño inicial de la población y el número de días del archivo de entrada y escribe el tamaño inicial de la población, el número de días y el tamaño final de la población en el archivo de salida. El archivo de entrada deberá contener dos números por línea que den el tamaño inicial de la población y el número de días, respectivamente. El programa contendrá un ciclo que procese los datos del archivo de entrada. Si este proyecto se dejó como tarea, pregunte a su profesor si hay instrucciones especiales respecto a los nombres de archivo o el formato de los archivos.

11. El valor e^x se puede aproximar con la sumatoria:

$1 + x + x^2/2! + x^3/3! + \ldots + x^n/n!$

Escriba un programa que reciba un valor x como entrada y envíe a la salida la sumatoria asignando a n sucesivamente los valores desde 1 hasta 100. El programa también deberá `exhibir` e^x calculado con la función predefinida `exp`. La función `exp(x)` devuelve una aproximación del valor e^x, y está en la biblioteca cuyo archivo de encabezado es `cmath`. El programa deberá repetir el cálculo con nuevos valores de x hasta que el usuario indique que ya terminó.

Use variables de tipo `double` para almacenar los factoriales, pues de lo contrario es probable que ocurra un desbordamiento de enteros (u organice su cálculo de modo que se evite un cálculo directo de los factoriales). El programa deberá recibir sus entradas del teclado (o de un archivo si su profesor lo estipula) y enviar sus salidas a un archivo. Si está haciendo este proyecto como tarea, pregunte a su profesor el nombre del archivo de salida, y si las entradas van a provenir de un archivo, pregunte también el nombre del archivo de entrada.

12. El valor aproximado de pi se puede calcular utilizando la serie que se presenta a continuación

$pi = 4[\ 1 - 1/3 + 1/5 - 1/7 + 1/9 \ldots + ((-1)^n)/(2n+1)\ ]$

Escriba un programa en C++ para calcular el valor aproximado de pi por medio de la serie dada. El programa toma una entrada n que determina el número de términos que se aproximan al valor de pi. Incluya un ciclo que permita al usuario repetir este cálculo para nuevos valores de n hasta que el usuario decida terminar el programa

8 Friends (amigas) y sobrecarga de operadores

8 Friends (amigas) y sobrecarga de operadores

Si nos dan las herramientas, TERMINAREMOS EL TRABAJO.

WINSTON CHURCHIL, *Transmisión de radio, Febrero 9, 1941*

Introducción

En este capítulo le explicaremos más técnicas para definir funciones y operadores de clases, que incluyen la sobrecarga de los operadores más comunes como +, *, y /, de tal suerte que se puedan utilizar con las clases que defina, de la misma manera en que se utilizan con los tipos predefinidos como `int` y `double`.

Prerrequisitos

En este capítulo utilizaremos material de los capítulos 2 a 7.

8.1 Funciones friend (amigas)

Confía en tus amigos.
Proverbio común

Hasta ahora hemos implementado las siguientes operaciones de clases: entrada, salida, funciones de acceso, etcétera, como funciones miembro de la clase, pero para algunas operaciones es más natural que se implementen como funciones ordinarias (no miembros). En esta sección explicaremos las técnicas para definir las operaciones sobre los objetos como funciones no miembros. Comenzaremos con un ejemplo sencillo.

EJEMPLO DE PROGRAMACIÓN
Función de igualdad

En el capítulo 6 desarrollamos una clase llamada `DiaDelAnio` que almacena una fecha como Enero 1 o Julio 4, la cual podría ser un día festivo, un cumpleaños o algún otro suceso anual. Progresivamente fuimos dando mejores versiones de la clase. La versión final se produjo en el ejercicio de autoevaluación 23 de dicho capítulo. En el cuadro 8.1 repetimos esta versión final de la clase `DiaDelAnio`. En dicho cuadro mejoramos una vez más la clase al agregar una función llamada `igual` que podía evaluar dos objetos de tipo `DiaDelAnio` para ver si sus valores representaban el mismo dato.

Suponga que `hoy` y `cumpleanios_de_Bach` son dos objetos de tipo `DiaDelAnio`, a los cuales les dimos valores que representan algunas fechas. Usted puede probar si representan la misma fecha mediante la siguiente expresión booleana:

```
igual(hoy, cumpleanios_de_Bach)
```

Esta llamada a la función `igual` devuelve *`verdadero`* si `hoy` y `Cumpleanios_de_Bach` representan la misma fecha. En el cuadro 8.1 se utiliza esta expresión booleana para controlar la instrucción *`if-else`*.

La definición de la función `igual` es muy sencilla. Dos fechas son iguales si representan el mismo mes y el mismo día del mes. La definición de `igual` utiliza las funciones de acceso `obtiene_mes` y `obtiene_dia` para comparar los meses y los días representados en los dos objetos.

Observe que no hicimos a la función `igual` una función miembro. Sería posible hacer a `igual` una función miembro de la clase `DiaDelAnio`, pero `igual` compara *dos* objetos de tipo `DiaDelAnio`. Si usted hace de `igual` una función miembro, entonces deberá decidir si el objeto que hace la llamada debe ser la primera o la segunda fecha. En lugar de elegir de manera arbitraria una de las dos fechas como el objeto que hace la llamada, tratamos a las dos fechas de la misma manera. Hicimos de `igual` una función ordinaria (no miembro) que toma dos fechas como sus argumentos.

CUADRO 8.1 *Función de igualdad (parte 1 de 3)*

```
//Programa para demostrar la función igual. La clase DiaDelAnio
//es la misma que la del ejercicio de autoevaluación 23-24 del
//capítulo 6.
#include <iostream>
using namespace std;

class DiaDelAnio
{
public:
    DiaDelAnio(int el_mes, int el_dia);
    //Precondición: el_mes y el_dia forman una
    //fecha posible. Inicializa la fecha de acuerdo con
    //los argumentos.

    DiaDelAnio( );
    //Inicializa primero la fecha con Enero.

    void entrada( );

    void salida( );

    int obtiene_mes( );
    //Devuelve el mes, 1 para Enero, 2 para Febrero, etc.

    int obtiene_dia( );
    //Devuelve el día del mes.
private:
    void comprueba_fecha( );
    int mes;
    int dia;
};

bool igual(DiaDelAnio fecha1, DiaDelAnio fecha2);
//Precondición: fecha1 y fecha2 deben tener valores.
//Devuelve true si fecha1 y fecha2 representan el mismo valor,
//en caso contrario devuelve false.

int main( )
{
    DiaDelAnio hoy, cumpleanios_bach(3, 21);
```

CUADRO 8.1 *Función de igualdad (parte 2 de 3)*

```
    cout << "Escriba la fecha de hoy:\n";
    hoy.entrada( );
    cout << "La fecha de hoy es ";
    hoy.salida( );

    cout << "El cumpleanios de J. S. Bach es ";
    cumpleanios_bach.salida( );

    if ( igual(hoy, cumpleanios_bach))
       cout << "Feliz cumpleanios, Johann Sebastian!\n";
   else
       cout << "Feliz no cumpleanios, Johann Sebastian!\n";
   return 0;
}

bool igual(DiaDelAnio fecha1, DiaDelAnio fecha2)
{
     return ( fecha1.obtiene_mes( ) == fecha2.obtiene_mes( ) &&
                 fecha1.obtiene_dia( ) == fecha2.obtiene_dia( ) );
}

DiaDelAnio::DiaDelAnio(int el_mes, int el_dia)
                    : mes(el_mes), dia(el_dia)
{
    comprueba_fecha();
}

DiaDelAnio::DiaDelAnio( ) : mes(1), dia(1)
{
    //Cuerpo vacío de manera intencional.
}

int DiaDelAnio::obtiene_mes( )
{
    return mes;
}

int DiaDelAnio::obtiene_dia( )
{
    return dia;
}
```

CUADRO 8.1 *Función de igualdad (parte 3 de 3)*

```
//Usa iostream:
void DiaDelAnio::entrada( )
{
   cout << "Escriba el mes como un numero: ";
   cin >> mes;
   cout << "Escriba el dia del mes: ";
   cin >> dia;
}

//Usa iostream:
void DiaDelAnio::salida( )
{
   cout << "mes = " << mes
        << ", dia = " << dia << endl;
}
void DiaDelAnio::comprueba_fecha( )
{
    if ((mes < 1) || (mes > 12)
        || (dia < 1) || (dia > 31))
    {
        cout << "Fecha ilegal. Se abortara el programa.\n";
        exit(1);
    }
    if (((mes == 4) || (mes == 6) || (mes == 9)
                                        || (mes == 11))
       && (dia == 31))
    {
        cout << "Fecha ilegal. Se abortara el programa.\n";
        exit(1);
    }
    if ((mes == 2) && (dia > 29))
    {
        cout << "Fecha ilegal. Se abortara el programa.\n";
        exit(1);
    }
}
```

Diálogo de ejemplo

```
Escriba la fecha de hoy:
Escriba el mes como un numero: 3
Escriba el dia del mes: 21
La fecha de hoy es mes = 3, dia = 21
El cumpleanios de J. S. Bach es mes = 3, dia = 21
Feliz cumpleanios, Johann Sebastian!
```

Ejercicio de AUTOEVALUACIÓN

1. Escriba la definición de una función llamada `antes` que reciba dos argumentos de tipo `DiaDelAnio`, el cual se define en el cuadro 8.1. La función debe devolver un valor *`bool`* y devolver *`true`* si el primer argumento representa una fecha previa a la fecha que representa el segundo argumento; de lo contrario, la función debe devolver *`false`*. Por ejemplo, el 5 de enero está antes que el 2 de febrero.

Funciones friend (amigas)

Si su clase contiene todo un conjunto de funciones de acceso, podemos utilizarlas para definir una función que evalúe la igualdad o que haga cualquier clase de cálculo que dependa de las variables miembro privadas. Sin embargo, aunque esto nos proporcione acceso a las variables miembro privadas, tal acceso podría no ser eficiente. Examinemos nuevamente la definición de la función `igual` dada en el cuadro 8.1. Para leer el mes, es preciso invocar a la función de acceso `obtiene_mes`. Para leer el día es preciso invocar a la función de acceso `obtiene_dia`. Esto funciona, pero el código sería más sencillo y más eficiente si pudiéramos solamente acceder a las variables miembro.

Una definición más sencilla y eficiente de la función `igual` que aparece en el cuadro 8.1 es la siguiente:

```
bool igual(DiaDelAnio fecha1, DiaDelAnio fecha2)
{
     return( fecha1.mes == fecha2.mes &&
                     fecha1.dia == fecha2.dia );
}
```

Existe solamente un problema con esta definición: Es ilegal porque las variables miembro `mes` y `dia` son miembros privados de la clase `DiaDelAnio`. Por lo general, no se puede hacer referencia a las variables miembro privadas (y las funciones miembro privadas) en el cuerpo de una función, a menos que la función sea una función miembro, e `igual` no es una función miembro de la clase `DiaDelAnio`. Pero existe una forma de dar a una función no miembro los mismos privilegios de acceso que una función miembro. Si hacemos de la función `igual` una *friend* de la clase `DiaDelAnio`, entonces la definición previa de `igual` será legal.

Las funciones amigas pueden acceder a miembros privados

Una **función amiga** de una clase no es una función miembro de la clase, pero tiene acceso a los miembros privados de esa clase, de igual forma que una función miembro. Una función amiga puede leer de manera directa el valor de una variable miembro e incluso puede modificar el valor de una variable miembro, por ejemplo, con una instrucción de asignación que tenga una variable miembro privada de un lado del operador de asignación. Para convertir a una función en una función amiga, le debe nombrar como amiga en la definición de la clase. Por ejemplo, en el cuadro 8.2 reescribimos la definición de la clase `DiaDelAnio` para que la función `igual` sea amiga de la clase. Para hacer que una función sea amiga de una clase, se lista la declaración de esa función en la definición de la clase y se coloca la palabra clave *`friend`* antes de la declaración de la función.

Una función friend no es una función miembro

Para agregar una función friend a la definición de la clase se incluye su declaración en la definición de la clase, tal como se incluye la declaración de una función miembro, sólo que se coloca la palabra reservada *`friend`* antes de la declaración de la función. Sin embargo, una función friend no es una función miembro; en realidad, es una función normal con acceso extraordinario a los datos miembro de la clase. La función friend se define y se invoca tal como una función normal. En particular, la definición de la función `igual` que mostramos en el cuadro 8.2 no incluye el calificador `DiaDelAnio::` en el encabezado de la función. Además, la función `igual` no se invoca mediante el uso del operador punto. La función `igual` toma como argumentos los objetos de tipo `DiaDelAnio`, de la misma manera en que cualquier otra función no miembro tomaría argumentos de cualquier otro tipo. Sin embargo, la definición de una función friend puede acceder a las variables miembro privadas y a las funciones miembro privadas de la clase por su nombre, así que tiene los mismos privilegios de acceso que una función miembro.

CUADRO 8.2 *La función de igualdad como función amiga (parte 1 de 2)*

```
//Demuestra el uso de la función igual.
//En esta versión, igual es función amiga de la clase DiaDelAnio.
#include <iostream>
using namespace std;

class DiaDelAnio
{
public:
    friend bool igual(DiaDelAnio fecha1, DiaDelAnio fecha2);
    //Precondición: fecha1 y fecha2 deben tener valores.
    //Devuelve true si fecha1 y fecha2 representan la misma fecha;
    //en caso contrario devuelve false.

    DiaDelAnio(int el_mes, int el_dia);
    //Precondición: el_mes y el_dia forman una
    //fecha posible. Inicializa la fecha de acuerdo con
    //los argumentos.

    DiaDelAnio( );
    //Inicializa primero la fecha con Enero.

    void entrada( );

    void salida( );

    int obtiene_mes( );
    //Devuelve el mes, 1 para Enero, 2 para Febrero, etc.

    int obtiene_dia( );
    //Devuelve el día del mes.
private:
    void comprueba_fecha( );
    int mes;
    int dia;
};
```

CUADRO 8.2 *La función de igualdad como función amiga (parte 2 de 2)*

```
int main( )
{
```

<La parte main del programa es la misma que el cuadro 8.1.>

```
}

bool igual(DiaDelAnio fecha1, DiaDelAnio fecha2)
{
   return ( fecha1.mes == fecha2.mes &&
                          fecha1.dia == fecha2.dia );
}
```

Observe que puede acceder a las variables miembro privadas `mes` *y* `dia` *por su nombre.*

<El resto del cuadro, incluso el diálogo de ejemplo, es el mismo que el cuadro 8.1.>

TIP DE PROGRAMACIÓN

Definición de funciones de acceso y funciones friend

Podría parecer que si convierte todas sus funciones en funciones `friend` de una clase, eliminaría la necesidad de incluir funciones de acceso y mutantes en la clase. Después de todo, las funciones `friend` tienen acceso a las variables miembro privadas y por lo tanto no requieren funciones de acceso ni mutantes. Esto no es totalmente erróneo. Es verdad que si usted convirtiera todas las funciones del mundo en funciones `friend` de una clase, no necesitaría funciones de acceso ni mutantes. Sin embargo, no es práctico hacer que todas las funciones sean `friend`.

Para poder ver por qué necesita de todas formas funciones de acceso, considere el ejemplo de la clase `DiaDelAnio` que se da en el cuadro 8.2. Podríamos utilizar esta clase en otro programa, y ese otro programa podría bien querer hacer algo con la parte correspondiente al mes de un objeto `DiaDelAnio`. Por ejemplo, el programa podría querer calcular cuántos meses restan en el año. En especial, la parte `main` del programa podría contener lo siguiente:

```
DiaDelAnio hoy;
cout << "Introduzca la fecha actual: \n";
hoy.entrada();
cout << "Existen " << (12 - hoy.obtiene_mes())
     << " meses restantes en el anio.\n";
```

No podemos remplazar `hoy.obtiene_mes()` por `hoy.mes` debido a que `mes` es un miembro privado de la clase; necesitamos la función de acceso `obtiene_mes()`.

Acabamos de ver que en definitiva necesitamos incluir las funciones de acceso en nuestra clase. Otros casos requieren funciones mutantes. Ahora podríamos pensar que, como por lo general necesitamos funciones de acceso y funciones mutantes, no necesitamos funciones `friend`. En cierto sentido, esto es verdad. Observe que podemos definir la función `igual` ya sea como `friend` sin el uso de funciones de acceso (cuadro 8.2) o no como `friend` y con el uso de funciones de acceso (como en el cuadro 8.1). En la mayoría de las situaciones, la única razón para que una función sea `friend` es hacer que la definición de la función sea más simple y eficiente; algunas veces esta razón es suficiente.

Funciones amigas

Una **función amiga** de una clase es una función común, sólo que tiene acceso a los miembros privados de los objetos de esa clase. Para convertir una función en amiga de una clase, debe listar la declaración de la función amiga en la definición de la clase. Debemos colocar la palabra reservada *`friend`* antes de la declaración de la función, la cual se puede colocar ya sea en la sección `private` o en la sección `public`, pero será una función pública en ambos casos, por lo cual es más claro listarla en la sección `public`.

Sintaxis (de la definición de una clase con funciones friend)

No necesitamos listar las funciones `friend` primero. Podemos mezclar el orden de la declaración de estas funciones.

```
class Nombre_Clase
{
public:
    friend Declaracion_para_funcion_friend_1
    friend Declaracion_ para_funcion_friend_2
                .
                .
                .

    Declaraciones_funciones_miembro
private:
    Declaraciones_miembros_privados
};
```

Ejemplo

```
class TanqueCombustible
{
public:
    friend double necesita_llenar(TanqueCombustible tanque);
    //Precondición: Las variables miembro de tanque deben
    //contener valores.
    //Devuelve el número de litros necesarios para llenar
    //el tanque.

    TanqueCombustible(double la_capacidad, double el_nivel);
    TanqueCombustible( );
    void entrada( );
    void salida( );
private:
    double capacidad;//en litros
    double nivel;
};
```

Una función `friend` *no* es una función miembro. Este tipo de función se define y se invoca de la misma manera que una función ordinaria. No debe utilizar el operador punto en una llamada a una función `friend` y no debe utilizar un calificador de tipo en la definición de una función `friend`.

TIP DE PROGRAMACIÓN

Utilice funciones miembro y funciones no miembro

Las funciones miembro y las funciones `friend` juegan un papel similar. De hecho, algunas veces no es claro si debemos hacer que cierta función sea función `friend` de una clase o función miembro de la clase. En la mayoría de los casos, podemos convertir a una función ya sea en una función miembro o en una función `friend` y realizará la misma tarea de la misma forma. Sin embargo, existen lugares en donde es mejor utilizar una función miembro y lugares en donde es mejor utilizar una función `friend` (o incluso una vieja función simple que no sea `friend`, como la versión de `igual` en el cuadro 8.1). A continuación se muestra una regla sencilla para decidir entre funciones miembro y funciones no miembro:

- Utilice una función miembro si la tarea a realizar por la función involucra sólo un objeto.
- Utilice una función no miembro si la tarea a realizar involucra a más de un objeto. Por ejemplo, la función `igual` del cuadro 8.1 (y del cuadro 8.2) involucra a dos objetos, por lo que creamos una función `friend` no miembro.

La elección entre convertir a una función no miembro en función `friend` o utilizar funciones de acceso y mutantes es cuestión de eficiencia y gusto personal. Mientras tenga suficientes funciones de acceso y funciones mutantes, cualquier método funcionará.

La elección de usar funciones miembro o no miembro no siempre es tan sencilla como en las dos reglas anteriores. Con más experiencia, descubrirá situaciones en las cuales vale la pena violar estas reglas. Una regla más precisa pero más difícil de comprender es que se deben usar funciones miembro cuando una tarea está muy relacionada con un solo objeto; se debe usar una función no miembro si la tarea involucra a más de un objeto y los objetos se utilizan de manera simétrica. Sin embargo, esta regla más precisa no es tan clara, por lo que las dos reglas sencillas anteriores servirán como una guía útil y confiable hasta que usted se haga más hábil en el manejo de los objetos.

EJEMPLO DE PROGRAMACIÓN

La clase Dinero (Versión 1)

El cuadro 8.3 contiene la definición de una clase llamada `Dinero`, la cual representa cantidades monetarias. El valor se implementa como un valor entero que representa el monto de dinero como si se convirtiera todo a centavos. Por ejemplo, `$9.95` se almacena como `995`. Dado que utilizamos un entero para representar el monto en dinero, este monto se representa como una cantidad exacta. No utilizamos un valor de tipo *`double`* dado que los valores de tipo *`double`* se almacenan como valores aproximados y queremos que nuestros montos en dinero sean cantidades exactas.

Este entero para el monto en dinero (expresado todo en centavos) se almacena en una variable miembro llamada `todo_centavos`. Podríamos utilizar *`int`* para el tipo de la variable miembro `todo_centavos`, pero en algunos compiladores esto limitaría de manera importante el monto de dinero que podemos representar. En algunas implementaciones de C++ sólo se utilizan dos bytes para almacenar el tipo *`int`*[1]. El resultado de esta implemen-

[1] Ver los detalles en el capítulo 2. El cuadro 2.2 da una descripción de los tipos de datos tal como los implementan la mayoría de los compiladores.

tación con dos bytes es que el valor más grande de tipo *int* es sólo un poco mayor que 32000, pero 32000 centavos tan sólo representan $320, la cual es una cantidad de dinero bastante pequeña. Dado que sería conveniente tratar con montos de dinero mucho mayores que $320, hemos optado por utilizar *long* para el tipo de la variable miembro todo_centavos. Por lo general, los compiladores de C++ que implementan el tipo *int* en dos bytes implementan el tipo *long* en cuatro bytes. Los valores de tipo *long* son enteros como los valores de tipo *int*, excepto que la implementación de cuatro bytes de *long* hace que el mayor valor permitido de tipo *long* sea mucho más grande que el mayor valor permitido de tipo *int*. En la mayoría de los sistemas, el valor más grande permitido de tipo *long* es de 2 mil millones o más. (Al tipo *long* se le llama también *long int*. Los dos nombres *long* y *long int* hacen referencia al mismo tipo).

<long>

La clase Dinero contiene dos operaciones que son funciones friend: igual y suma (la cual definimos en el cuadro 8.3). La función suma devuelve un objeto Dinero cuyo valor es la suma de los valores de sus dos argumentos. Una llamada a la función de la forma igual(monto1, monto2) devuelve *true* si los dos objetos monto1 y monto2 contienen valores que representan montos iguales de dinero.

Observe que la clase Dinero lee y escribe montos de dinero de la misma forma común en que escribimos montos de dinero como $9.95 o −$9.95. Consideremos primero la función miembro entrada (definida también en el cuadro 8.3). Esta función primero lee un carácter individual, el cual podría ser un signo de moneda ('$') o el signo negativo ('−'). Si este primer carácter es el signo negativo, entonces la función recuerda que el monto es negativo asignando *true* al valor de la variable negativo. Después lee un carácter adicional, el cual debe ser el signo de moneda. Por otro lado, si el primer símbolo no es '−', entonces se asigna el valor *false* a negativo. En este punto ya se leyeron el signo negativo (si existe) y el signo de moneda. Ahora la función entrada lee la cantidad en moneda como un valor de tipo *long* y coloca dicha cantidad en la variable local llamada pesos. Después de leer la parte de la cantidad de la entrada, la función entrada lee el resto de la entrada como valores de tipo *char*; lee tres caracteres, los cuales deben ser un punto decimal y dos dígitos.

entrada

(Podríamos sentirnos tentados a definir la función miembro entrada de tal manera que lea el punto decimal como un valor de tipo *char* y que después lea el número de centavos como un valor de tipo *int*. Esto no se debe hacer debido a la manera en que algunos compiladores de C++ tratan con los ceros a la izquierda. Como explicamos en la sección de Riesgo titulada "Ceros a la izquierda en constantes numéricas", muchos de los compiladores que se siguen utilizando hoy en día no leen los números con ceros a la izquierda como es de esperarse, de modo que un monto como $7.09 se podría leer incorrectamente si su código en C++ leyera el 09 como un valor de tipo *int*).

La siguiente instrucción de asignación convierte los dos dígitos que componen la parte de los centavos del monto de entrada en un solo entero, el cual se almacena en la variable local centavos:

```
centavos = digito_a_int(digito1)*10 + digito_a_int(digito2);
```

Una vez que se ejecuta esta instrucción de asignación, el valor en centavos es el número de centavos en el monto de entrada.

La función auxiliar digito_a_int toma un argumento que es un dígito como '3' y lo convierte en su correspondiente valor *int*, como 3. Necesitamos esta función de ayuda debido a que la función miembro entrada lee los dos dígitos para el número de centavos como dos valores de tipo *char*, los cuales se almacenan en las variables locales digito1 y digito2. Sin embargo, una vez que se leen los dígitos dentro de la computadora, queremos utilizarlos como números. Por lo tanto, utilizamos la función digito_a_int para convertir un dígito como '3' en un número como 3. En el cuadro 8.3 aparece la función

digito_a_int

CUADRO 8.3 *La clase* Dinero *– Versión 1 (parte 1 de 5)*

```
//Programa que demuestra el uso de la clase Dinero.
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;


//Clase para montos de dinero.
class Dinero
{
public:
    friend Dinero suma(Dinero cantidad1, Dinero cantidad2);
    //Precondición: cantidad1 y cantidad2 han recibido valores.
    //Devuelve la suma de los valores de cantidad1 y cantidad2.

    friend bool igual(Dinero cantidad1, Dinero cantidad2);
    //Precondición: cantidad1 y cantidad2 han recibido valores.
    //Devuelve true si cantidad1 y cantidad2 tienen el mismo valor;
    //en caso contrario devuelve false.

    Dinero(long pesos, int centavos);
    //Inicializa el objeto de manera que su valor represente un monto con los
    //pesos y los centavos que se proporcionan como argumentos. Si el monto
    //es negativo, entonces tanto la cantidad como los centavos deben ser
    //negativos.

    Dinero(long pesos);
    //Inicializa el objeto de manera que su valor represente $pesos.00.

    Dinero( );
    //Inicializa el objeto de manera que su valor represente $0.00.

    double obtiene_valor( );
    //Precondición: el objeto que hizo la llamada debe tener un valor.
    //Devuelve el monto de dinero registrado en los datos del objeto que
    //hace la llamada.

    void entrada(istream& ins);
    //Precondición: si ins es un flujo de entrada de archivo, entonces ya se ha
    //conectado a un archivo. Un monto de dinero, incluyendo el signo de pesos,
    //se ha escrito en el flujo de entrada ins. La notación para montos negativos
    //es -$100.00.
    //Postcondición: El valor del objeto que hizo la llamada se ajusta al
    //monto de dinero que se leyó del flujo de entrada ins.
```

CUADRO 8.3 *La clase* Dinero *– Versión 1 (parte 2 de 5)*

```
    void salida(ostream& outs);
    //Precondición: si outs es un flujo de salida de archivo, entonces ya se ha
    //conectado a un archivo.
    //Postcondición: se han enviado un signo de pesos y el monto de dinero
    //registrado en el objeto que hizo la llamada al flujo de salida outs.
private:
    long todo_centavos;
};

int digito_a_int(char c);
//Declaración para la función que se utiliza en la definición de Dinero::entrada:
//Precondición: c debe ser uno de los dígitos del '0' al '9'.
//Devuelve el entero para el dígito; por ejemplo, digito_a_int('3') devuelve 3.

int main( )
{
    Dinero su_monto, mi_monto(10, 9), nuestro_monto;
    cout << "Escriba un monto de dinero: ";
    su_monto.entrada(cin);
    cout << "Su monto es ";
    su_monto.salida(cout);
    cout << endl;
    cout << "Mi monto es ";
    mi_monto.salida(cout);
    cout << endl;

    if (igual(su_monto, mi_monto))
        cout << "Tenemos los mismos montos.\n";
    else
        cout << "Uno de nosotros es mas rico.\n";
    nuestro_monto = suma(su_monto, mi_monto);
    su_monto.salida(cout);
    cout << " + ";
    mi_monto.salida(cout);
    cout << " es igual a ";
    nuestro_monto.salida(cout);
    cout << endl;
    return 0;
}
```

CUADRO 8.3 *La clase* `Dinero` *– Versión 1 (parte 3 de 5)*

```
Dinero suma(Dinero monto1, Dinero monto2)
{
    Dinero temp;

    temp.todo_centavos = monto1.todo_centavos + monto2.todo_centavos;
    return temp;
}

bool igual(Dinero monto1, Dinero monto2)
{
    return (monto1.todo_centavos == monto2.todo_centavos);
}

Dinero::Dinero(long pesos, int centavos)
{
    if(pesos*centavos < 0) //Si uno es negativo y el otro es positivo
    {
        cout << "Valores ilegales para pesos y centavos.\n";
        exit(1);
    }
    todo_centavos = pesos*100 + centavos;
}

Dinero::Dinero(long pesos) : todo_centavos(pesos*100)
{
    //Cuerpo vacío de manera intencional.
}

Dinero::Dinero( ) : todo_centavos(0)
{
    //Cuerpo vacío de manera intencional.
}

double Dinero::obtiene_valor( )
{
    return (todo_centavos * 0.01);
}
```

CUADRO 8.3 *La clase* Dinero *– Versión 1 (parte 4 de 5)*

```
//Usa iostream, cctype, cstdlib:
void Dinero::entrada(istream& ins)
{
    char un_char, punto_decimal,
         digito1, digito2; //dígitos para el monto de los centavos
    long pesos;
    int centavos;
    bool negativo;//se hace true si entrada is negativo.

    ins >> un_char;
    if (un_char == '-')
    {
        negativo = true;
        ins >> un_char; //lee el '$'
    }
    else
        negativo = false;
    //si entrada es legal, entonces un_char == '$'

    ins >> pesos >> punto_decimal >> digito1 >> digito2;

    if ( un_char != '$' || punto_decimal != '.'
         || !isdigit(digito1) || !isdigit(digito2) )
    {
        cout << "Error formato ilegal para introducir dinero\n";
        exit(1);
    }
    centavos = digito_a_int(digito1)*10 + digito_a_int(digito2);

    todo_centavos = pesos*100 + centavos;
    if (negativo)
        todo_centavos = -todo_centavos;
}
```

CUADRO 8.3 *La clase* `Dinero` *– Versión 1 (parte 5 de 5)*

```
//Usa cstdlib and iostream:
void Dinero::salida(ostream& outs)
{
    long centavos_positivos, pesos, centavos;
    centavos_positivos = labs(todo_centavos);
    pesos = centavos_positivos/100;
    centavos = centavos_positivos%100;

    if (todo_centavos < 0)
        outs << "-$" << pesos << '.';
    else
        outs << "$" << pesos << '.';

    if (centavos < 10)
        outs << '0';
    outs << centavos;
}

int digito_a_int(char c)
{
    return (static_cast<int>(c) - static_cast<int>('0') );
}
```

Diálogo de ejemplo

```
Escriba un monto de dinero: $123.45
Su monto es $123.45
Mi monto es $10.09
Uno de nosotros es mas rico.
$123.45 + $10.09 es igual a $133.54
```

`digito_a_int`. Puede confiar en que esta definición hace lo que se supone que debe hacer, para tratar a esta función como una caja negra. Todo lo que necesita saber es que `digito_a_int('0')` devuelve `0`, `digito_a_int('1')` devuelve `1`, y así sucesivamente. Sin embargo, no es tan difícil ver como trabaja esta función, de manera que puede leer la sección opcional que sigue a ésta, en donde explicaremos la implementación de `digito_a_int`.

Una vez que las variables locales `pesos` y `centavos` adquieren los pesos y los centavos del monto de la entrada, es fácil establecer el valor de la variable miembro

todo_centavos. La siguiente instrucción de asignación establece todo_centavos al número correcto de centavos:

```
todo_centavos = pesos*100 + centavos;
```

Sin embargo, esto siempre da a todo_centavos un valor positivo. Si el monto del dinero es negativo, entonces el valor todo_centavos se debe modificar de positivo a negativo. Esto se hace con la siguiente instrucción:

```
if (negativo)
    todo_centavos = -todo_centavos;
```

salida

La función miembro salida (cuadro 8.3) calcula los pesos y el número de centavos del valor de la variable miembro todo_centavos. Calcula los pesos y los centavos con el uso de la división entera entre 100. Por ejemplo, si todo_centavos tiene un valor igual a 995 (centavos), entonces la cantidad es 995/100, la cual es igual a 9, y el número de centavos es 995%100, que es 95. Así, el valor de salida será $9.95 cuando el valor del todo_centavos sea de 995 (centavos).

La definición de la función miembro salida necesita tomar provisiones especiales para desplegar cantidades negativas. El resultado de la división entera con miembros negativos no tiene una definición estándar y puede variar de una implementación a otra. Para evitar este problema, tomamos el valor absoluto del número en todo_centavos antes de realizar la división. Para calcular el valor absoluto utilizamos la función predefinida labs. La función labs devuelve el valor absoluto de sus argumentos, tal como la función abs, pero labs toma un argumento de tipo long y devuelve un valor de tipo *long*. La función labs se encuentra en la biblioteca con el nombre de encabezado cstlib, tal como la función abs. (Algunas versiones de C++ no incluyen labs. Si su implementación de C++ no incluye labs, es muy sencillo definirla usted mismo).

Implementación de digito_a_int (*Opcional*)

He aquí la definición de la función digito_a_int del cuadro 8.3:

```
int digito_a_int(char c)
{
    return( static_cast<int>(c) - static_cast<int>('0') );
}
```

A primera vista, la fórmula para el valor devuelto podría parecer un poco extraña, pero los detalles no son tan complicados. Por ejemplo, el dígito a convertir '3' es el parámetro c y el valor devuelto resultará ser el valor int correspondiente, en este ejemplo, 3. Como vimos en los capítulos 2 y 5, los valores de tipo *char* se implementan como números. Por desgracia y como ejemplo, el número que implementa el dígito '3' no es el número 3. La conversión de tipo *static_cast<int>(c)* produce el número que implementa el carácter c y convierte este número al tipo *int*. Esto hace que c cambie de tipo char a un número de tipo *int*; el problema es que no es el número que deseamos. Por ejemplo, *static_cast<int>('3')* no es 3, sino algún otro número. Necesitamos convertir *static_cast<int>(c)* en el número que corresponde a c (por ejemplo, '3' en 3). Veamos ahora cómo debemos ajustar *static_cast<int>(c)* para obtener el número que deseamos.

Sabemos que los dígitos están en orden. Entonces `static_cast<int>('0') + 1` es igual a `static_cast<int>('1')`; `static_cast<int>('1') + 1` es igual a `static_cast<int>('2')`; `static_cast<int>('2') + 1` es igual a `static_cast<int>('3')`, y así sucesivamente. Saber que los dígitos están en este orden es todo lo que necesitamos para ver que `digito_a_int` devuelve el valor correcto. Si `c` es `'0'`, el valor devuelto es:

```
static_cast<int>(c) - static_cast<int>('0')
```

el cual es

```
static_cast<int>('0') - static_cast<int>('0')
```

Por lo que `digito_int('0')` devuelve `0`.

Consideremos ahora qué sucede cuando `c` toma el valor `'1'`. El valor que se devuelve entonces es `static_cast<int>(c) - static_cast<int>('0')`, el cual es `static_cast<int>('1') - static_cast<int>('0')`. Eso equivale a `c static_cast<int>('0') + 1) - static_cast<int>('0')` y eso, a su vez, equivale a `static_cast<int>('0')- static_cast<int>('0')+1`. Ya que `static_cast<int>('0') - static_cast<int>('0')` es `0`, este resultado es `0 + 1`, o `1`. Puede verificar los otros dígitos del `'2'` al `'9'` por su cuenta; cada dígito produce un número que es `1` mayor que el dígito anterior.

RIESGO *Ceros a la izquierda en constantes numéricas*

Las siguientes son declaraciones de objetos que se proporcionan en la parte `main` del programa en el cuadro 8.3:

```
Dinero su_monto, mi_monto(10, 9), nuestro_monto;
```

Los dos argumentos en `mi_monto(10, 9)` representan `$10.09`. Ya que por lo general escribimos los centavos en el formato `".09"`, podríamos sentirnos tentados a escribir la declaración del objeto como `mi_monto(10, 09)`. Sin embargo, esto provocará problemas. En matemáticas, 9 y 09 representan el mismo número. Sin embargo, algunos compiladores de C++ utilizan un cero a la izquierda para indicar un tipo diferente de número, de tal forma que en C++ las constantes 9 y 09 no tienen que ser el mismo número. En algunos compiladores, un cero a la izquierda significa que el número está escrito en base 8 y no en base 10. Dado que los números en base 8 no utilizan el dígito 9, la constante 09 no tiene sentido en C++. Las constantes 00 a 07 deben funcionar correctamente, ya que significan lo mismo en base 8 y en base 10, pero algunos sistemas tendrán problemas en ciertos contextos, incluso con los números del 00 al 07.

El estándar ANSI de C++ estipula que las entradas deben interpretarse de manera predeterminada como números decimales, sin importar el 0 a la izquierda. El compilador de C++ del proyecto GNU llamado g++ y el compilador VC++ de Microsoft cumplen con el estándar, por lo que no tienen problemas con los ceros a la izquierda. La mayoría de los fabricantes de software dan seguimiento al estándar ANSI y por ende deben cumplir con el estándar ANSI C++, de modo que este problema de los ceros a la izquierda desaparecerá en algún momento. Sería conveniente que escribiera un pequeño programa para evaluar esta situación en su compilador.

Ejercicios de AUTOEVALUACIÓN

2. ¿Cual es la diferencia entre una función `friend` de una clase y una función miembro de la clase?

3. Suponga que desea agregar una función `friend` a la clase `DiaDelAnio` que aparece en el cuadro 8.2. A esta función `friend` le llamará `despues` y recibirá dos argumentos de tipo `DiaDelAnio`. La función devolverá *`true`* si el primer argumento representa una fecha posterior a la fecha representada por el segundo argumento; de lo contrario, la función devolverá *`false`*. Por ejemplo, Febrero 2 es posterior a Enero 5. ¿Que es lo que necesita agregar a la definición de la clase `DiaDelAnio` en el cuadro 8.2?

4. Suponga que desea agregar una función `friend` para la resta a la clase `Dinero` que aparece en el cuadro 8.3. ¿Qué necesita agregar a la definición de la clase `Dinero` que aparece en el cuadro 8.3? La función de resta debe tomar dos argumentos de tipo `Dinero` y debe regresar un valor de tipo `Dinero`, cuyo valor sea el valor del primer argumento menos el valor del segundo argumento.

5. Observe la función miembro `salida` en la definición de la clase `Dinero` que aparece en el cuadro 8.3. Con el objeto de escribir un valor de tipo `Dinero` en la pantalla, hace una llamada a `salida` con `cout` como argumento. Por ejemplo, si `monedero` es un objeto de tipo `Dinero`, entonces para mostrar el monto de dinero que hay en `monedero` en la pantalla, debe escribir lo siguiente en su programa:

   ```
   monedero.salida(cout);
   ```

   Podría ser mejor no tener que listar el flujo `cout` al enviar salida a la pantalla.

   Reescriba la definición de la clase para el tipo `Dinero` que se proporciona en el cuadro 8.3. La única modificación es que esta versión reescrita sobrecarga a la función de nombre `salida`, por lo que existen dos versiones de `salida`. Una versión es justo como la versión que aparece en el cuadro 8.3; la otra versión de `salida` no toma argumentos y envía su salida a la pantalla. Con esta versión reescrita de tipo `Dinero`, las dos siguientes llamadas son equivalentes:

   ```
   monedero.salida(cout);
   ```

   y

   ```
   monedero.salida();
   ```

   pero la segunda es más simple. Observe que ya que existen dos versiones de la función `salida`, puede enviar la salida a un archivo. Si `outs` es un flujo de archivo de salida conectado a un archivo, entonces la siguiente instrucción enviará la cantidad de dinero en el objeto `monedero` hacia el archivo conectado a `outs`.

   ```
   monedero.salida(outs);
   ```

6. Observe la definición de la función miembro `entrada` de la clase `Dinero` que aparece en el cuadro 8.3. Si el usuario introduce cierta clase de entrada incorrecta, la función despliega un mensaje de error y termina el programa. Por ejemplo, si el usuario omite un signo de moneda, la función despliega un mensaje de error. Sin embargo, las verificaciones que aparecen ahí no atrapan toda clase de entrada incorrecta. Por ejemplo, se supone que los montos negativos de dinero se deben introducir en la forma **−$9.95**, pero si el usuario introduce por error un monto con la forma **$−9.95**, entonces la entrada no desplegará un mensaje de error y el valor del objeto `Dinero` se establecerá con un valor incorrecto. ¿Cual monto leerá la función miembro `entrada` si el usuario introduce **$−9.95** por error? ¿Cómo podría agregar comprobaciones adicionales para atrapar a la mayoría de los errores provocados por un signo negativo mal colocado?

7. La sección Riesgo titulada “Ceros a la izquierda en constantes numéricas” le sugiere que escriba un programa corto para evaluar si un 0 a la izquierda provocará que su compilador interprete la entrada de números como si estuvieran en base ocho. Escriba tal programa.

El modificador de parámetro `const`

Un parámetro de llamada por referencia es más eficiente que un parámetro de llamada por valor. Un parámetro de llamada por valor es una variable local que se inicializa con el valor de su argumento, por lo que cuando se hace una llamada a la función hay dos copias del argumento. Con un parámetro de llamada por referencia, el parámetro es sólo un recipiente que se sustituye por el argumento, de manera que sólo hay una copia del argumento. Para parámetros de tipos simples como *int* o *double*, la diferencia en eficiencia es inapreciable, pero para los parámetros de clases la diferencia en eficiencia puede algunas veces ser importante. Por ello, puede tener sentido utilizar un parámetro de llamada por referencia en lugar de un parámetro de llamada por valor, incluso si la función no modifica ese parámetro.

Si utilizamos un parámetro de llamada por referencia y su función no modifica el valor del parámetro, podemos marcarlo de tal modo que el compilador sepa que el parámetro no debe modificarse. Para ello debe colocar el modificador *const* antes del tipo del parámetro. De esta forma, al parámetro se le llama **parámetro constante**. Por ejemplo, considere la clase `Dinero` que se definió en el cuadro 8.3. Los parámetros de `Dinero` para la función `friend suma` pueden convertirse en parámetros constantes de la siguiente manera:

parámetro constante

```
class Dinero
{
public:
     friend Dinero suma(const Dinero& monto1, const Dinero& monto2);
   //Precondición: monto1 y monto2 deben tener valores.
   //Devuelve la suma de los valores de monto1 y monto2.
        ...
```

Cuando empleamos parámetros constantes, debemos utilizar el modificador *const* tanto en la declaración de la función como en la definición del encabezado de la función; de esta forma, con la modificación anterior en la definición de la clase, la definición de la función `suma` comenzaría de la siguiente manera:

```
Dinero suma(const Dinero& monto1, const Dinero& monto2)
{
      ...
```

El resto de la definición de la función sería igual que la del cuadro 8.3.

Los parámetros constantes son una forma automática de comprobación de errores. Si la definición de su función contiene un error que provoca un cambio inadvertido en el parámetro constante, entonces la computadora desplegará un mensaje de error.

El modificador de parámetro *const* se puede utilizar con cualquier tipo de parámetro; sin embargo, por lo general se utiliza sólo para parámetros de llamada por referencia para las clases (y a veces para otros parámetros cuyos argumentos correspondientes son grandes).

`const` con funciones miembro

Cuando se llama a una función, los parámetros de llamada por referencia se sustituyen con argumentos y la llamada a la función puede (o no) modificar el valor del argumento. Cuando tenemos una llamada a una función miembro, el objeto que hace la llamada se comporta en forma muy parecida a un parámetro de llamada por referencia. Cuando tenemos una llamada a una función miembro, esa llamada a la función puede modificar el valor del objeto que hace la llamada. Considere el siguiente ejemplo en donde la clase `Dinero` es como en el cuadro 8.3:

```
Dinero m;
m.entrada(cin);
```

Cuando se declara el objeto `m`, el valor de la variable miembro `todo_centavos` se inicializa con 0. La llamada a la función miembro `entrada` modifica el valor de la variable miembro `todo_centavos` y le asigna un nuevo valor determinado por lo que escriba el usuario. Así, la llamada `m.entrada(cin)` modifica el valor de `m`, tal como si `m` fuera un argumento de llamada por referencia.

El modificador *`const`* se aplica a los objetos que hacen llamadas a funciones de la misma manera en que se aplica a los parámetros. Si tenemos una función miembro que no debe modificar el valor de un objeto que haga la llamada, podemos marcar esa función con el modificador *`const`*; entonces la computadora desplegará un mensaje de error si el código de su función modifica de manera inadvertida el valor del objeto que hizo la llamada. En el caso de una función miembro, *`const`* va al final de la declaración de la función, justo antes del punto y coma final, como se muestra a continuación:

```
class Dinero
{
public:
        ...
     void salida(ostream& outs) const;
        ...
```

El modificador *`const`* se debe utilizar tanto en la declaración de la función como en la definición de la misma, así que la definición de la función para `salida` sería la siguiente:

```
void Dinero::salida(ostream& outs) const
{
    ...
```

El resto de la definición de la función sería igual que la del cuadro 8.3.

RIESGO *Uso inconsistente de* `const`

El uso del modificador *`const`* es una cuestión de todo o nada. Si utilizamos *`const`* para un parámetro de algún tipo específico, entonces debemos utilizarlo para todos los demás parámetros que tengan ese tipo y que no sean modificados por la llamada a la función; es más, si el tipo es una clase, entonces también deberemos utilizar el modificador *`const`* para cada función miembro que no modifique el valor de su objeto que hizo la llamada. El motivo tiene que ver con las llamadas a funciones dentro de otras llamadas a funciones. Por ejemplo, considere la siguiente definición de la función `garantia`:

```
void garantia(const Dinero& precio)
{
   cout << "Si no esta satisfecho, le \n"
        << "reembolsaremos el doble de su dinero.\n"
        << "Eso seria una devolucion de $"
        << (2*precio.obtiene_valor()) << endl;
}
```

Si *no* agregamos el modificador *const* a la declaración de la función para la función miembro `obtiene_valor`, entonces en la mayoría de los compiladores la función `garantia` producirá un

mensaje de error. La función miembro `obtiene_valor` no modifica el valor del objeto `precio` que hizo la llamada. Sin embargo, cuando el compilador procese la definición de la función para `garantia`, pensará que `obtiene_valor` va a (o al menos podría) modificar el valor de `precio`. Esto se debe a que cuando el compilador traduce la definición de la función `garantia`, todo lo que conoce acerca de la función miembro `obtiene_valor` es la declaración de la función `obtiene_valor`; si la declaración de la función no contiene la palabra clave *`const`*, la cual le indica al compilador que no se modificará el objeto que hizo la llamada, entonces el compilador supondrá que se va a modificar el objeto que hizo la llamada. Por ende, si utiliza el modificador *`const`* con parámetros de tipo `Dinero`, entonces también debería utilizar *`const`* con todas las funciones miembro de `Dinero` que no modifiquen el valor de su objeto que hizo la llamada. En particular, la declaración de la función miembro `obtiene_valor` debe incluir un *`const`*.

En el cuadro 8.4 hemos reescrito la definición de la clase `Dinero` que aparece en el cuadro 8.3; esta vez hemos utilizado el modificador *`const`* en los lugares apropiados. Las definiciones de las funciones miembro y de las funciones `friend` serían las mismas que las del cuadro 8.3, excepto que debe usarse el modificador *`const`* en los encabezados de las funciones de modo que coincidan con las declaraciones de las funciones que aparecen en el cuadro 8.4.

Modificador de parámetro `const`

Si colocamos el modificador *`const`* antes del tipo para un parámetro de llamada por referencia, este parámetro será un **parámetro constante**. (El encabezado de la definición de la función también debe contener la palabra clave *`const`*, de manera que coincida con la declaración de la función). Cuando agregamos *`const`*, le indicamos al compilador que este parámetro no se debe modificar. Si comete un error en su definición de la función de tal forma que modifique el parámetro constante, entonces la computadora mostrará un mensaje de error. Los parámetros de un tipo de clase que por lo general no se modifican por la función deben ser parámetros constantes de llamada por referencia, en lugar de parámetros de llamada por valor.

Si una función miembro no modifica el valor del objeto que la llama, entonces podemos marcar la función agregando el modificador *`const`* a la declaración de esa función. Si comete un error en su definición de la función de manera tal que modifique el objeto que hace la llamada y la función se marca con *`const`*, entonces la computadora mostrará un mensaje de error. La palabra clave *`const`* se coloca al final de la declaración de la función, justo antes del punto y coma final. El encabezado de la definición de la función también debe tener la palabra *`const`*, de manera que coincida con la declaración de la función.

Ejemplo

```
class Ejemplo
{
public:
    Ejemplo( );
    friend int compara(const Ejemplo& s1, const Ejemplo& s2);
    void entrada( );
    void salida( ) const;
private:
    int cosas;
    double mas_cosas;
};
```

El uso del modificador *`const`* es una cuestión de todo o nada. Debemos utilizarlo siempre que sea apropiado para el parámetro de una clase y siempre que sea apropiado para una función miembro de la clase. Si no utilizamos la palabra clave *`const`* cada vez que sea apropiado para una clase, entonces no deberemos utilizarla para dicha clase.

CUADRO 8.4 *La clase* Dinero *con parámetros constantes*

```
//Clase para montos de dinero.
class Dinero
{
public:
    friend Dinero suma(const Dinero& monto1, const Dinero& monto2);
    //Precondición: monto1 y monto2 han recibido valores.
    //Devuelve la suma de los valores de monto1 y monto2.

    friend bool igual(const Dinero& monto1, const Dinero& monto2);
    //Precondición: monto1 y monto2 han recibido valores.
    //Devuelve true si monto1 y monto2 tienen el mismo valor;
    //en caso contrario devuelve false.

    Dinero(long pesos, int centavos);
    //Inicializa el objeto de manera que su valor represente un monto con los
    //pesos y los centavos que se proporcionan como argumentos. Si el monto es
    //negativo, entonces tanto la cantidad como los centavos deben ser negativos.

    Dinero(long pesos);
    //Inicializa el objeto de manera que su valor represente $pesos.00.

    Dinero( );
    //Inicializa el objeto de manera que su valor represente $0.00.

    double obtiene_valor( ) const;
    //Precondición: el objeto que hizo la llamada debe tener un valor.
    //Devuelve el monto de dinero registrado en los datos del objeto que hace la llamada.

    void entrada(istream& ins);
    //Precondición: Si ins es un flujo de entrada de archivo, entonces ya se ha
    //conectado a un archivo. Un monto de dinero, incluyendo el signo de pesos, se ha
    //escrito en el flujo de entrada ins. La notación para montos negativos es -$100.00.
    //Postcondición: El valor del objeto que hizo la llamada se ajusta al
    //monto de dinero que se leyó del flujo de entrada ins.

    void salida(ostream& outs) const;
    //Precondición: Si outs es un flujo de salida de archivo, entonces ya se ha
    //conectado a un archivo.
    //Postcondición: Se han enviado un signo de pesos y el monto de dinero
    //registrado en el objeto que hizo la llamada al flujo de salida outs.
private:
    long todo_centavos;
};
```

Ejercicios de AUTOEVALUACIÓN

8. Escriba una definición completa de la función miembro `obtiene_valor` que utilizaría con la definición de `Dinero` que aparece en el cuadro 8.4.

9. ¿Por qué sería incorrecto agregar el modificador *const*, como se hace a continuación, a la declaración de la función para la función miembro `entrada` de la clase `Dinero` que aparece en el cuadro 8.4?

```
class Dinero
{
        ...
public:
    void entrada(istream& ins) const;
        ...
```

10. ¿Cuáles son las diferencias y las similitudes entre un parámetro de llamada por valor y un parámetro `const` de llamada por referencia? La declaración de la función que explica esto es:

```
void llamada_por_valor(int x);
void llamada_por_referencia_const(const int & x);
```

11. Dadas las siguientes definiciones:

```
const int x = 17;
class A
{
public:
   A( );
   A(int x );
   int f( )const;
   int g(const A& x);
private:
   int i;
};
```

Cada una de las tres palabras reservadas *const* es una promesa que el compilador debe hacer valer. ¿Cuál es la promesa en cada caso?

8.2 Sobrecarga de operadores

Es un operador hábil.

Línea de la canción de Sade (escrita por Sade Adu y Ray St. John)

En una sección anterior de este capítulo vimos como hacer que la función `suma` fuera una función friend de la clase `Dinero`; también vimos cómo utilizarla para sumar dos objetos de tipo `Dinero` (cuadro 8.3). La función `suma` es adecuada para sumar objetos, pero sería mejor si pudiéramos utilizar sólo el operador + común para sumar valores de tipo `Dinero`, como en la última línea del siguiente código:

```
Dinero total, costo, impuesto;
cout << "Escriba el costo y el impuesto: ";
costo.entrada(cin);
impuesto.entrada(cin);
total = costo + impuesto;
```

en lugar de tener que utilizar la expresión (un poco más complicada):

```
total = suma(costo, impuesto);
```

operadores y funciones

Recuerde que un operador como + es en realidad sólo una función, con la excepción de que la sintaxis relativa a la forma en que se utiliza es un poco distinta a la de una función ordinaria. En una llamada a una función ordinaria los argumentos se colocan en paréntesis después del nombre de la función, como en el siguiente ejemplo:

```
suma(costo, impuesto)
```

Con un operador (binario), los argumentos se colocan en cualquier lado del operador, como se muestra a continuación:

```
costo + impuesto
```

Podemos sobrecargar una función para que tome argumentos de tipos diferentes. Un operador es en realidad una función, así que se puede sobrecargar. En esencia, la manera en la que se sobrecarga un operador tal como + es la misma que la manera en que se sobrecarga el nombre de una función. En esta sección mostraremos cómo se sobrecargan los operadores en C++.

Sobrecarga de operadores

Podemos sobrecargar el operador + (y muchos otros operadores más) de modo que acepte argumentos de un tipo de clase. La diferencia entre sobrecargar el operador + y la definición de la función `suma` (que aparece en el cuadro 8.3) sólo involucra una ligera modificación de la sintaxis. La definición del operador + sobrecargado es en esencia la misma que la definición de la función `suma`. Las únicas diferencias son que se utiliza el nombre + en lugar del nombre `suma` y que se coloca la palabra reservada *`operator`* antes del +. En el cuadro 8.5 hemos reescrito el tipo `Dinero` para incluir el operador + sobrecargado e integramos la definición en un pequeño programa de demostración.

La clase `Dinero`, como la definimos en el cuadro 8.5, sobrecarga también al operador == de manera que pueda usarse para comparar dos objetos de tipo `Dinero`. Si `monto1` y `monto2` son dos objetos de tipo `Dinero`, deseamos que la expresión

```
monto1 == monto2
```

CUADRO 8.5 *Sobrecarga de operadores (parte 1 de 2)*

```
//Programa que demuestra el uso de la clase Dinero. (Ésta es una versión mejorada de
//la clase Dinero que utilizamos en el cuadro 8.3 y reescribimos en el cuadro 8.4.)
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

//Clase para montos de dinero.
class Dinero
{
public:
    friend Dinero operator +(const Dinero& monto1, const Dinero& monto2);
    //Precondición: monto1 y monto2 han recibido valores.
    //Devuelve la suma de los valores de monto1 y monto2.

    friend bool operator ==(const Dinero& monto1, const Dinero& monto2);
    //Precondición: monto1 y monto2 han recibido valores.
    //Devuelve true si monto1 y monto2 tienen el mismo valor;
    //en caso contrario devuelve false.

    Dinero(long pesos, int centavos);

    Dinero(long pesos);

    Dinero( );

    double obtiene_valor( ) const;

    void entrada(istream& ins);

    void salida(ostream& outs) const;
private:
    long todo_centavos;
};
```

Algunos comentarios del cuadro 8.4 se han omitido para ahorrar espacio en este libro, pero deben incluirse en un programa real.

<Aquí van todas las declaraciones de funciones adicionales del cuadro 8.3.>

```
int main( )
{
    Dinero costo(1, 50), impuesto(0, 15), total;
    total = costo + impuesto;

    cout << "costo = ";
    costo.salida(cout);
    cout << endl;
```

CUADRO 8.5 *Sobrecarga de operadores (parte 2 de 2)*

```
    cout << "impuesto = ";
    impuesto.salida(cout);
    cout << endl;
    cout << "factura total = ";
    total.salida(cout);
    cout << endl;
    if (costo == impuesto)
        cout << "Irse a otro estado.\n";
    else
        cout << "Todo parece normal.\n";
    return 0;
}
Dinero operator +(const Dinero& monto1, const Dinero& monto2)
{
    Dinero temp;
    temp.todo_centavos = monto1.todo_centavos + monto2.todo_centavos;
    return temp;
}
bool operator ==(const Dinero& monto1, const Dinero& monto2)
{
    return (monto1.todo_centavos == monto2.todo_centavos);
}
```

<Las definiciones de las funciones miembro son iguales que en el cuadro 8.3, sólo que se agrega `const` a los encabezados de las funciones en varios lugares para que concuerden con las declaraciones de las funciones en la definición de clase anterior. No se necesitan otras modificaciones en las definiciones de la función miembro. Los cuerpos de las definiciones de las funciones miembro son idénticos a los del cuadro 8.3.>

Salida

```
costo = $1.50
impuesto = $0.15
factura total = $1.65
Todo parece normal.
```

devuelva el mismo valor que la siguiente expresión:

```
monto1.todo_centavos == monto2.todo_centavos
```

Como mostramos en el cuadro 8.5, este es el valor que devuelve el operador sobrecargado ==.

Podemos sobrecargar la mayoría, pero no todos los operadores. El operador no necesita ser friend (amigo) de la clase, pero con frecuencia esto será conveniente. En el cuadro titulado "Reglas acerca de la sobrecarga de operadores" podrá encontrar algunos detalles técnicos acerca de cuándo y cómo podemos sobrecargar un operador.

Sobrecarga de operadores

Un operador (binario) tal como +, -, /, %, etc., es simplemente una función a la que se llama mediante el uso de una sintaxis diferente en la lista de argumentos. En un operador, los argumentos se listan antes y después del operador; en una función, los argumentos se listan entre paréntesis después del nombre de la función. La definición de un operador se escribe de modo similar a la definición de una función, excepto que la definición del operador incluye la palabra reservada *operator* antes del nombre del operador. Los operadores predefinidos como + y los demás, se pueden sobrecargar al darles una nueva definición para un tipo de clase.

Un operador puede ser `friend` (amigo) de una clase, aunque esto no se requiere. En el cuadro 8.5 aparece un ejemplo de sobrecarga del operador + como friend.

Ejercicios de AUTOEVALUACIÓN

12. ¿Cuál es la diferencia entre un operador (binario) y una función?

13. Suponga que desea sobrecargar el operador < de tal forma que se pueda aplicar en el tipo `Dinero` definido en el cuadro 8.5. ¿Qué necesita para agregarlo a la descripción de `Dinero` que aparece en el cuadro 8.5?

14. Suponga que desea sobrecargar el operador <= de forma que se pueda aplicar en el tipo `Dinero` que aparece en el cuadro 8.5. ¿Qué necesita para agregarlo a la descripción de `Dinero` en el cuadro 8.5?

15. ¿Es posible utilizar la sobrecarga de operadores para modificar el comportamiento de + sobre los enteros? ¿Por qué sí o por qué no?

Constructores para la conversión automática de tipos

Si nuestra definición de clase contiene los constructores apropiados, el sistema realizará ciertas conversiones automáticas de tipo. Por ejemplo, si nuestro programa contiene la definición de la clase `Dinero` que aparece en el cuadro 8.5, podríamos utilizar el siguiente código en nuestro programa:

```
Dinero monto_base(100, 60), monto_total;
monto_total = monto_base + 25;
monto_total.salida(cout);
```

Reglas para la sobrecarga de operadores

- Al sobrecargar un operador, al menos un argumento del operador sobrecargado resultante debe ser de tipo clase.
- Un operador sobrecargado puede ser (pero no es obligatorio) friend (amigo) de una clase; la función operador podría ser miembro de la clase o una función ordinaria (no friend). (En el apéndice 9 hablaremos sobre la sobrecarga de un operador como miembro de una clase.)
- No podemos crear un nuevo operador. Todo lo que podemos hacer es sobrecargar los operadores existentes, como +, −, *, /, y así sucesivamente.
- No podemos modificar el número de argumentos que recibe un operador. Por ejemplo, no podemos modificar % de un operador binario a un operador unario cuando sobrecarga %, no podemos modificar ++ de un operador unario a un operador binario cuando lo sobrecarga.
- No podemos modificar la precedencia de un operador. Un operador sobrecargado tiene la misma precedencia que la versión común del operador. Por ejemplo, `x*y + z` siempre significa `(x*y) + z`, incluso si `x`, `y`, y `z` son objetos y los operadores + y * están sobrecargados para las clases apropiadas.
- Los siguientes operadores no pueden sobrecargarse: el operador punto (.), el operador de resolución de alcance (`::`), y los operadores `.*` y `?:`, los cuales no explicaremos en este libro.
- Aunque el operador de asignación = puede sobrecargarse para que el significado predeterminado de = se sustituya por un nuevo significado, esto debe hacerse de manera diferente a la que describimos aquí. En el capítulo 12 hablaremos sobre la sobrecarga de =. Algunos otros operadores, incluyendo a `[]` y `->`, también deben sobrecargarse de una manera diferente a la que describimos en este capítulo. Más adelante hablaremos sobre los operadores `[]` y `->`.

La salida será

```
$125.60
```

El código anterior podría tener una apariencia muy sencilla y natural, pero existe un pequeño punto a considerar. El `25` (en la expresión `monto_base + 25`) no es del tipo apropiado. En el cuadro 8.5 solamente sobrecargamos el operador + de modo que se pudiera utilizar con dos valores de tipo `Dinero`. No sobrecargamos + para que se pudiera utilizar con un valor de tipo `Dinero` y uno de tipo entero. La constante `25` es un entero y no es de tipo `Dinero`. La constante `25` puede ser considerada de tipo *`int`* o de tipo *`long`*, pero el `25` no se puede utilizar como un valor de tipo `Dinero`, a menos que la definición de la clase indique de alguna forma al sistema cómo convertir un entero en un valor de tipo `Dinero`. La única manera de que el sistema sepa que `25` significa $25.00 es que incluyamos un constructor que tome un sólo argumento de tipo *`long`*. Cuando el sistema ve la expresión

```
monto_base + 25
```

primero comprueba si el operador + está sobrecargado para la combinación de un valor de tipo `Dinero` y un entero. Dado que no existe tal sobrecarga, el sistema busca a continuación si existe un constructor que tome un solo argumento que sea entero. Si encuentra el constructor que tome un sólo argumento tipo entero, lo utiliza para convertir el entero `25`

en un valor de tipo Dinero. El constructor con un argumento de tipo *long* le indica al sistema cómo convertir un entero como 25 en un valor de tipo Dinero. El constructor de un sólo argumento indica que el 25 debe convertirse en un objeto de tipo Dinero cuya variable miembro todo_centavos sea igual a 2500; en otras palabras, el constructor convierte el 25 en un objeto de tipo Dinero que representa $25.00. (La definición del constructor aparece en el cuadro 8.3.)

Observe que este tipo de conversión no funcionará a menos que exista un constructor a la medida. Por ejemplo, el tipo Dinero (cuadro 8.5) no contiene un constructor que tome un argumento de tipo *double*, por lo que la siguiente instrucción no es válida y producirá un mensaje de error si la coloca dentro de un programa que declare a monto_base y monto_total como de tipo Dinero.

```
monto_total = monto_base + 25.67;
```

Para que este uso de + sea legal, podría modificar la definición de la clase Dinero al sumar otro constructor. La declaración de la función para el constructor que necesita agregar es:

```
class Dinero
{
public:
          . . .
    Dinero(double monto);
    //Inicializa el objeto de modo que su valor represente $monto.
          . . .
```

Dejaremos la escritura de la definición para este nuevo constructor para el ejercicio de autoevaluación 16.

Estos tipos automáticos de conversión (producidos por los constructores) parecen más comunes y atractivos con los operadores numéricos sobrecargados como + y −. Sin embargo, estas conversiones automáticas se aplican de la misma manera a los argumentos para las funciones comunes, los argumentos para las funciones miembro y los argumentos para otros operadores sobrecargados.

Ejercicios de AUTOEVALUACIÓN

16. Escriba una definición para el constructor que explicamos en la sección anterior. El constructor debe agregarse a la clase Dinero del cuadro 8.5. La definición comienza de la siguiente manera:

```
Dinero::Dinero(double monto)
{
```

Sobrecarga de operadores unarios

Además de los operadores biinarios como + en x + y, también existen los operadores unarios como el operador – cuando se utiliza para indicar negación. En la siguiente instrucción, el operador unario – se utiliza para hacer el valor de la variable x igual al negativo del valor de la variable y:

```
x = -y;
```

Los operadores de incremento y decremento ++ y -- también son ejemplos de operadores unitarios.

Podemos sobrecargar operadores unarios así como operadores binarios. Por ejemplo, podemos redefinir el tipo `Dinero` que aparece en el cuadro 8.5 de modo tal que contenga tanto la versión unaria y binaria del operador de resta/negación −. La definición rehecha de la clase aparece en el cuadro 8.6. Suponga que su programa contiene esta definición de clase y el siguiente código:

```
Dinero monto1(10), monto2(6), monto3;
```

Entonces la siguiente instrucción establece el valor de `monto3` igual a `monto1` menos `monto2`:

```
monto3 = monto1 − monto2;
```

Entonces, la siguiente instrucción desplegará `$4.00` en la pantalla:

```
monto3.salida(cout);
```

Por otra parte, la siguiente instrucción establecerá `monto3` igual al negativo de `monto1`:

```
monto3 = −monto1;
```

La siguiente instrucción entonces mostrará `−$10.00` en la pantalla:

```
monto3.salida(cout);
```

++ y --

Podemos sobrecargar los operadores ++ y -- de manera similar a la forma en que sobrecargamos el operador de negación que aparece en el cuadro 8.6. La definición sobrecargada se aplicará al operador cuando se utilice en la posición de prefijo, como en `++x` y `--x`. Las versiones posfijo de ++ y -- (como en `x++` y `x--`) se manipulan de manera distinta, pero no explicaremos estas versiones en posfijo. (Hey, ¡no se puede aprender todo en un primer curso!)

Sobrecarga de >> y <<

<< es un operador

El operador de inserción << que utilizamos con `cout` es un operador binario como los operadores binarios + y −. Por ejemplo, considere la siguiente función:

```
cout << "Hola alla afuera.\n";
```

El operador es <<, el primer operando es el flujo de salida `cout` y el segundo operando es la cadena `"Hola alla afuera.\n"`. Podemos modificar cualquiera de estos operandos. Si `fout` es una cadena de salida de tipo `ofstream` y se conecta a un archivo mediante una llamada a `open`, entonces puede remplazar `cout` con `fout` y la cadena se escribirá en el archivo conectado a `fout`. Claro que también podemos remplazar la cadena `"Hola alla afuera.\n"` con otra cadena, una variable, o un número. Ya que el operador de inserción << es un operador, deberemos poder sobrecargarlo de la misma forma en que se sobrecargan los operadores como + y −. Esto es verdad, pero existen algunos detalles adicionales en los que debemos poner atención cuando se sobrecargan los operadores de entrada y salida, >> y <<.

sobrecarga de <<

En nuestras definiciones anteriores de la clase `Dinero` utilizamos la función `salida` para mostrar valores de tipo `Dinero` (cuadros 8.3 a 8.6). Esto es lo adecuado, pero sería

CUADRO 8.6 *Sobrecarga de un operador unitario*

Ésta es una nueva versión de la clase `Dinero` *proporcionada en el cuadro 8.5.*

```
//Clase para montos de dinero.
class Dinero
{
public:
    friend Dinero operator +(const Dinero& monto1, const Dinero& monto2);

    friend Dinero operator -(const Dinero& monto1, const Dinero& monto2);
    //Precondición: monto1 y monto2 han recibido valores.
    //Devuelve monto1 menos monto2.

    friend Dinero operator -(const Dinero& monto);
    //Precondición: monto ha recibido un valor.
    //Devuelve el negativo del valor de monto.

    friend bool operator ==(const Dinero& monto1, const Dinero& monto2);

    Dinero(long pesos, int centavos);

    Dinero(long pesos);

    Dinero( );

    double obtiene_valor( ) const;

    void entrada(istream& ins);
    void salida(ostream& outs) const;
private:
    long todo_centavos;
};
```

Hemos omitido las directivas `Include` *y los comentarios pero usted debe incluirlos en su programa.*

<Aquí van las declaraciones de funciones adicionales, así como la parte principal del programa.>

```
Dinero operator -(const Dinero& monto1, const Dinero& monto2)
{
    Dinero temp;
    temp.todo_centavos = monto1.todo_centavos - monto2.todo_centavos;
    return temp;
}

Dinero operator -(const Dinero& monto)
{
    Dinero temp;
    temp.todo_centavos = -monto.todo_centavos;
    return temp;
}
```

<Las demás definiciones de funciones son iguales que en el cuadro 8.5.>

mejor si pudiera simplemente utilizar el operador de inserción << para desplegar los valores de tipo `Dinero` de la siguiente manera:

```
Dinero monto(100);
cout << "Yo tengo " << monto <<  " en mi monedero.\n";
```

en lugar de tener que utilizar la función miembro `salida` de la siguiente manera:

```
Dinero monto(100);
cout << "Yo tengo ";
monto.salida(cout);
cout << " en mi monedero.\n";
```

Un problema al sobrecargar el operador << es decidir qué valor se debe devolver al utilizar << en una expresión como la siguiente

```
cout << monto
```

Los dos operandos en esta expresión son `cout` y `monto`; la evaluación de esta expresión debe provocar que el valor de `monto` se escriba en la pantalla. Pero si << es un operador como + o *, entonces la expresión anterior también debe devolver algún valor. Después de todo, las expresiones con otros operandos, como `n1 + n2`, devuelven valores. ¿Pero qué devuelve `cout << monto`? Para obtener la respuesta a esta pregunta, necesitamos revisar una expresión más complicada que involucre a <<.

cadenas de <<

Consideremos la siguiente expresión, la cual implica evaluar una cadena de expresiones mediante el uso de <<:

```
cout << "Yo tengo " << monto << " en mi monedero.\n";
```

Si consideramos que el operador << es análogo a otros operadores como +, entonces la instrucción anterior debería ser (y de hecho lo es) equivalente a la siguiente:

```
( (cout << "Yo tengo ") << monto ) << "en mi monedero.\n";
```

¿Qué valor debe devolver << para que la expresión tenga sentido? Lo primero que se evalúa es la subexpresión:

```
(cout << "Yo tengo ")
```

Si las cosas van a salir bien, entonces más vale que la subexpresión devuelva `cout` para que pueda continuar el cálculo de la siguiente manera:

```
( cout << monto ) << " en mi monedero.\n";
```

Y si todo sigue saliendo bien, también (`cout << monto`) debe devolver `cout` para que el cálculo pueda continuar de la siguiente manera:

```
cout << " en mi monedero.\n";
```

Esto lo explicamos en el cuadro 8.7. El operador << debe devolver su primer argumento, el cual es un flujo de tipo `ostream`.

<< devuelve un flujo

CUADRO 8.7 << *como un operador*

```
cout << "Yo tengo " << monto << " en mi monedero.\n";
```

significa lo mismo que

```
((cout << "Yo tengo ") << monto) << " en mi monedero .\n";
```

y se puede evaluar de la siguiente manera:

Primero evalúa `(cout << "Yo tengo ")`, lo cual devuelve `cout`:

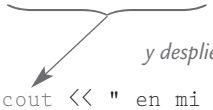
```
((cout << "Yo tengo ") << monto) << " en mi monedero .\n";
```

y despliega la cadena "Yo tengo ".

```
(cout << monto) << " en mi monedero.\n";
```

Luego evalúa `(cout << "Yo tengo ")`, lo cual devuelve `cout`:

```
((cout << monto) << " en mi monedero.\n";
```



y despliega el valor del monto.

```
cout << " en mi monedero.\n";
```

Luego evalúa `cout << " en mi monedero.\n"`, lo cual devuelve `cout`:

```
cout << "en mi monedero.\n";
```

y despliega la cadena " en mi monedero.\n".

```
cout;
```

Dado que no existen más operadores <<, el proceso termina.

Así, la siguiente es la declaración para la sobrecarga del operador << (para utilizarla con la clase Dinero):

```
class Dinero
{
public:
     ...
     friend ostream& operator <<(ostream& outs, const Dinero& monto);
     //Precondición: Si outs es un flujo de salida de archivo,
     //entonces outs
     //ya está conectado a un archivo.
     //Postcondición: se ha enviado un signo de moneda y el
     //monto almacenado
     //en el objeto que hizo la llamada al flujo de salida outs.
     ...
```

Una vez que hayamos sobrecargado el operador de inserción (salida) <<, ya no necesitaremos a la función miembro `salida` y así podremos eliminarla de nuestra definición de la clase `Dinero`. La definición de la sobrecarga del operador << es muy similar a la función miembro `salida`. A manera de plan general, la definición para el operador sobrecargado es la siguiente:

```
ostream& operator <<(ostream& outs, const Dinero& monto)
{
    <Esta parte es la misma que el cuerpo de
    Dinero::salida que aparece en el cuadro 8.3 (excepto que
    todo_centavos se sustituye con monto.todo_centavos).>

    return outs;
}
```

<< y >> devuelven una referencia

Sólo falta explicar una cosa respecto a la declaración y la definición de la función anterior para el operador << sobrecargado. ¿Cuál es el significado de `&` en el tipo devuelto `ostream&`? La respuesta más fácil es que *cada vez que un operador (o una función) devuelve un flujo, se debe agregar un & al final del nombre del tipo de retorno*. Esta regla sencilla le permitirá sobrecargar los operadores << y >>. Sin embargo, aunque esta es una buena regla funcional que le permitirá escribir sus definiciones de clase y sus programas, no es muy satisfactoria. No necesita saber qué significa `&` en realidad, pero si lo explicamos se perderá parte del misterio de la regla que nos dice que agreguemos un `&`.

retorno de una referencia

Cuando agregamos una `&` al nombre de un tipo devuelto, lo que dice es que el operador (o función) devuelve una *referencia*. Todas las funciones y operadores que hemos visto hasta ahora devuelven valores. No obstante, si el tipo de retorno es un flujo, no basta con sólo devolver el valor de ese flujo. En el caso de un flujo, su valor es un archivo completo o el teclado o la pantalla, y pudiera no tener sentido devolver dichas cosas. Por ello es conveniente devolver sólo el flujo en sí, en vez del valor del flujo. Cuando agregamos un `&` al nombre de un tipo devuelto, indica que el operador (o función) devuelve una **referencia**, lo que significa que estamos devolviendo al objeto en sí, en vez del valor del objeto.

referencia

El operador de extracción >> se sobrecarga de manera análoga a la que explicamos para el operador de inserción <<. Sin embargo, con el operador de extracción (entrada) >>, el segundo argumento será el objeto que recibe el valor de entrada, así que el segundo parámetro será el objeto que recibe el valor de entrada, de manera que el segundo parámetro debe ser un parámetro ordinario de llamada por referencia. En forma de plan general la definición para el operador de extracción>> sobrecargado es la siguiente:

```
istream& operator >>(istream& ins, Dinero& monto)
{
    <Esta parte es la misma que el cuerpo de
    Dinero::entrada  que aparece en el cuadro 8.3 (excepto que
    todo_centavos se remplaza con monto.todo_centavos).>

    return ins;
}
```

En el cuadro 8.8 aparecen las definiciones completas de los operadores << y >> sobrecargados, en donde reescribimos la clase `Dinero` de nuevo. Esta vez reescribimos la clase de modo tal que se sobrecarguen los operadores << y >> para que nos permitan utilizarlos con valores de tipo `Dinero`.

Ejercicios de AUTOEVALUACIÓN

17. A continuación mostraremos una definición de una clase llamada `Pares`. Los objetos de tipo `Pares` se pueden utilizar en cualquier situación en donde se necesiten pares ordenados. Su tarea es escribir implementaciones del operador sobrecargado >> y del operador sobrecargado << para que los objetos de la clase `Pares` se introduzcan y se muestren en la forma `(5, 6)` `(5, −4)` `(−5, 4)` o `(−5, −6)`. No necesita implementar constructor alguno u otro miembro, y no necesita comprobar el formato de la entrada.

```
#include <iostream>
using namespace std;
class Pares
{
public:
   Pares( );
   Pares(int primero, int segundo);
   //otros miembros y friend
   friend istream& operator>> (istream& ins, Pares& segundo);
   friend ostream& operator<< (ostream& outs, const Pares& segundo);
private:
   int f;
   int s;
};
```

CUADRO 8.8 *Sobrecarga de << y >> (parte 1 de 4)*

Esta es una versión mejorada de la clase `Dinero` *que explicamos en el cuadro 8.6.*

Aunque omitimos algunos de los comentarios de los cuadros 8.5 y 8.6, debe incluirlos.

```
//Programa que demuestra el uso de la clase Dinero.
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cctype>
using namespace std;

//Clase para montos de dinero.
class Dinero
{
public:
    friend Dinero operator +(const Dinero& monto1, const Dinero& monto2);

    friend Dinero operator -(const Dinero& monto1, const Dinero& monto2);

    friend Dinero operator -(const Dinero& monto);

    friend bool operator ==(const Dinero& monto1, const Dinero& monto2);

    Dinero(long pesos, int centavos);

    Dinero(long pesos);

    Dinero( );

    double obtiene_valor( ) const;

    friend istream& operator >>(istream& ins, Dinero& monto);
    //Sobrecarga el operador >> de manera que pueda usarse para introducir
    //valores de tipo Dinero.
    //La notación para introducir montos negativos es como en -$100.00.
    //Precondición: Si ins es un flujo de entrada de archivo, entonces ya se ha
    //conectado a un archivo.

    friend ostream& operator <<(ostream& outs, const Dinero& monto);
    //Sobrecarga el operador << de manera que pueda usarse para mostrar valores
    //de tipo Dinero.
    //Va antes de cada valor de salida de tipo Dinero con signo de pesos.
    //Precondición: Si outs es un flujo de archivo de entrada,
    //entonces outs ya se ha conectado a un archivo.
private:
    long todo_centavos;
};
```

CUADRO 8.8 *Sobrecarga de << y >> (parte 2 de 4)*

```
int digito_a_int(char c);
//Se utiliza en la definición del operador de entrada >> sobrecargado.
//Precondición: c es uno de los dígitos del '0' al '9'.
//Devuelve el entero para el dígito; por ejemplo, digito_a_int('3') devuelve 3.

int main( )
{
    Dinero monto;
    ifstream flujo_ent;
    ofstream flujo_sal;

    flujo_ent.open("archivoent.dat");
    if (flujo_ent.fail( ))
    {
        cout << "No se pudo abrir el archivo de entrada.\n";
        exit(1);
    }

    flujo_sal.open("archivosal.dat");
    if (flujo_sal.fail( ))
    {
        cout << "No se pudo abrir el archivo de salida.\n";
        exit(1);
    }

    flujo_ent >> monto;
    flujo_sal << monto
              << " se copio del archivo archivoent.dat.\n";
    cout << monto
         << " se copio del archivo archivoent.dat.\n";

    flujo_ent.close( );
    flujo_sal.close( );

    return 0;
}
```

CUADRO 8.8 *Sobrecarga de << y >> (parte 3 de 4)*

```
//Usa iostream, cctype, cstdlib:
istream& operator >>(istream& ins, Dinero& monto)
{
    char un_char, punto_decimal,
         digito1, digito2; //digitos para el monto de los centavos
    long pesos;
    int centavos;
    bool negativo;//se hace true si entrada es negativo.

    ins >> un_char;
    if (un_char == '-')
    {
        negativo = true;
        ins >> un_char; //lee el '$'
    }
    else
        negativo = false;
    //si entrada es legal, entonces un_char == '$'

    ins >> pesos >> punto_decimal >> digito1 >> digito2;

    if ( un_char != '$' || punto_decimal !='.'
         || !isdigit(digito1) || !isdigit(digito2) )
    {
        cout << "Error formato ilegal para introducir dinero\n";
        exit(1);
    }

    centavos = digito_a_int(digito1)*10 + digito_a_int(digito2);

    monto.todo_centavos = pesos*100 + centavos;
    if (negativo)
        monto.todo_centavos = -monto.todo_centavos;

    return ins;
}

int digito_a_int(char c)
{
    return (static_cast<int>(c) - static_cast<int>('0') );
}
```

CUADRO 8.8 *Sobrecarga de << y >> (parte 4 de 4)*

```
//Usa cstdlib e iostream:
ostream& operator <<(ostream& outs, const Dinero& monto)
{
    long centavos_positivos, pesos, centavos;
    centavos_positivos = labs(monto.todo_centavos);
    pesos = centavos_positivos/100;
    centavos = centavos_positivos%100;

    if (monto.todo_centavos < 0)
        outs << "-$" << pesos << '.';
    else
        outs << "$" << pesos << '.';

    if (centavos < 10)
        outs << '0';
    outs << centavos;

    return outs;
}
```

< Aquí van las definiciones de las funciones miembro y de otros operadores sobrecargados.
Vea los cuadros 8.3, 8.4, 8.5, y 8.6 para la definición.>

archivoent.dat
(No lo modifica el programa.)

```
   $1.11  $2.22
$3.33
```

archivosal.dat
(Después de la ejecución del programa.)

```
$1.11 se copio del archivo archivoent.dat.
```

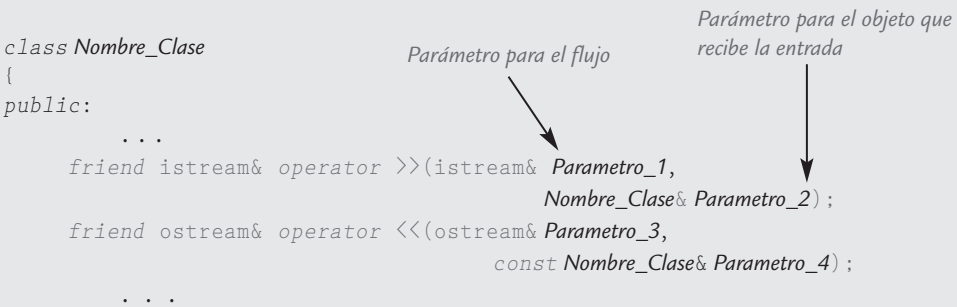
Salida de pantalla

```
$1.11 se copio del archivo archivoent.dat.
```

Sobrecarga de >> y <<

Los operadores de entrada y salida >> y << se pueden sobrecargar como cualquier otro operador. El valor devuelto debe ser el flujo. El tipo para el valor devuelto debe contener el símbolo & al final del nombre del tipo. A continuación se muestran las declaraciones de las funciones y el inicio de las definiciones de las mismas. En el cuadro 8.8 podremos ver un ejemplo.

Declaraciones de funciones




```
class Nombre_Clase
{
public:
        . . .
    friend istream& operator >>(istream& Parametro_1,
                                Nombre_Clase& Parametro_2);
    friend ostream& operator <<(ostream& Parametro_3,
                                const Nombre_Clase& Parametro_4);
        . . .
```

Definiciones

```
istream& operator >>(istream& Parametro_1,
                     Nombre_Clase& Parametro_2)
{
        . . .
}

ostream& operator >>(ostream& Parametro_3,
                     const Nombre_Clase& Parametro_4)
{
        . . .
```

18. A continuación aparece la definición de una clase llamada `Porcentaje`. Los objetos de tipo `Porcentaje` representan porcentajes como 10% o 99%. Escriba las definiciones de los operadores sobrecargados >> y << de tal manera que se puedan utilizar para la entrada y salida con objetos de la clase `Porcentaje`. Suponga que la entrada siempre consta de un entero seguido por el carácter `'%'`, como `25%`. Todos los porcentajes son números enteros y se almacenan en la variable miembro `int` llamada `valor`. No necesita definir los demás operadores sobrecargados y no necesita definir el constructor. Sólo tiene que definir los operadores sobrecargados >> y <<.

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

class Porcentaje
{
public;
    friend bool operator ==(const Porcentaje& primero,
                            const Porcentaje& segundo);
    friend bool operator <(const Porcentaje& primero,
                           const Porcentaje& segundo);
Porcentaje( );
Porcentaje(int valor_porcentaje);
```

```
    friend istream& operator >>(istream& ins,
                                Porcentaje& el_objeto);
    //Sobrecarga el operador >> para introducir valores de tipo Porcentaje.
    //Precondición: Si ins es un flujo de entrada de archivo, entonces
    //ya se ha conectado a un archivo.

    friend ostream& operator <<(ostream& outs,
                            const Porcentaje& un_porcentaje);
    //Sobrecarga el operador << para mostrar valores de tipo Porcentaje.
    //Precondición: Si outs es un flujo de salida de archivo, entonces
    //ya se ha conectado a un archivo.
private:
    int valor;
};
```

Resumen del capítulo

- Una función friend de una clase es una función común, sólo que contiene acceso a los miembros privados de la clase, como lo hacen las funciones miembro.
- Si sus clases contienen cada una un conjunto completo de funciones de acceso y funciones mutantes, entonces la única razón para hacer que una función sea `friend` es para que la definición de la función friend sea más simple y eficiente, pero por lo general es una razón suficiente.
- Un parámetro de un tipo de clase que no se modifique por la función debe ser por lo general un parámetro constante.
- Los operadores como + y == pueden sobrecargarse de tal forma que puedan utilizarse con objetos de un tipo de clase que usted defina.
- Cuando se sobrecargan los operadores >> o <<, el tipo devuelto debe ser un tipo de flujo y una referencia, lo cual se indica cuando se agrega un `&` al nombre del tipo devuelto.

Respuestas a los ejercicios de autoevaluación

1.
```
bool antes(DiaDelAnio fecha1, DiaDelAnio fecha2)
{
return ( (fecha1.obtiene_mes() < fecha2.obtiene_mes())
       ||(fecha1.obtiene_mes() == fecha2.obtiene_mes()
                && fecha1.obtiene_dia() < fecha2.obtiene_dia()) );
}
```

   La expresión booleana anterior indica que `fecha1` va antes que `fecha2`, siempre y cuando el mes de `fecha1` vaya antes que el mes de `fecha2` o que los meses sean los mismos y que el día en `fecha1` vaya antes que el día en `fecha2`.

2. Una función friend y una función miembro se parecen en cuanto a que las dos pueden utilizar cualquier miembro de la clase (ya sea público o privado) dentro de su definición. Sin embargo, una función friend se define y se utiliza tal y como una función ordinaria; no se utiliza el operador punto cundo se hace una llamada a una función friend y no se utiliza calificador de tipo alguno cuando se define una función friend. Por otro lado, una función miembro se llama mediante el uso de un nombre de objeto y el operador punto. Además, la definición de una función miembro incluye un calificador de tipo que consta del nombre de la clase y el operador de resolución de alcance ::.

3. A continuación se muestra la definición modificada de la clase `DiaDelAnio`. La parte sombreada es nueva. Omitimos algunos comentarios para ahorrar espacio, pero en esta definición se deben incluir todos los comentarios que aparecen en la figura 8.2.

```
class DiaDelAnio
{
public:
    friend bool igual(DiaDelAnio fecha1, DiaDelAnio fecha2);
    friend bool despues(DiaDelAnio fecha1, DiaDelAnio fecha2);
    //Precondición: fecha1 y fecha2 contienen valores.
    //Devuelve true si fecha1 va después de fecha2 en el calendario;
    //de lo contrario devuelve false.

    DiaDelAnio(int el_mes, int el_dia);
    DiaDelAnio();
    void entrada();
    void salida();
    int obtiene_mes();
    int obtiene_dia();
private:

void verifica_fecha();
int mes;
int dia;
};
```

También deberá incluir la siguiente definición de la función `despues`:

```
bool despues(DiaDelAnio fecha1, DiaDelAnio fecha2)
{
     return ( (fecha1.mes > fecha2.mes) ||
               ((fecha1.mes == fecha2.mes) && (fecha1.dia > fecha2.dia) ) );
}
```

4. A continuación aparece la definición modificada de la función `Dinero`. La parte sombreada es nueva. Hemos omitido algunos comentarios para ahorrar espacio, pero todos los comentarios que se muestran en el cuadro 8.3 se deben incluir en la siguiente definición.

```
class Dinero
{
public:
    friend Dinero suma(Dinero monto1, Dinero monto2);
    friend Dinero resta(Dinero monto1, Dinero monto2);
    //Precondición: monto1 y monto2 tienen valores.
    //Devuelve monto1 menos monto2.
    friend bool igual(Dinero monto1, Dinero monto2);
    Dinero(long pesos, int centavos);
```

```
    Dinero(long pesos);
    Dinero();
    double obtiene_valor();
    void entrada(istream& ins);
    void salida(ostream& outs);
private:
    long todo_centavos;
};
```

Además, debe agregar la siguiente definición de la función `resta`:

```
Dinero resta(Dinero monto1, Dinero monto2)
{
     Dinero temp;
     temp.todo_centavos = monto1.todo_centavos
                          -monto2.todo_centavos;
     return temp;
}
```

5. A continuación mostramos la definición modificada de la función `Dinero`. La parte sombreada es nueva. Omitimos algunos comentarios para ahorrar espacio, pero todos los comentarios mostrados en el cuadro 8.3 se deben incluir en esta definición.

```
class Dinero
{
public:
    friend Dinero suma(Dinero monto1, Dinero monto2);
    friend bool igual(Dinero monto1, Dinero monto2);
    Dinero(long pesos, int centavos);
    Dinero(long pesos);
    Dinero();
    double obtiene_valor();
    void entrada(istream& ins);

    void salida(ostream& outs);
    //Precondición: Si outs es un flujo de salida de archivo, entonces
    //ya se ha conectado a un archivo.
    //Postcondición: Se ha enviado un signo de moneda y el monto de dinero
    //almacenado en el objeto que hizo la llamada al
    //flujo de salida outs.

    void salida();
    //Postcondición: Se ha mostrado en pantalla un signo de moneda y el monto de
    //dinero
    //almacenado en el objeto que hizo la llamada.
private:
    long todo_centavos;
};
```

También debe agregar la siguiente definición del nombre de la función `salida`. (También se va a quedar la antigua definición de `salida`, de modo que existirán dos definiciones de `salida`.)

```
void Dinero::salida()
{
     salida(cout);
}
```

La siguiente versión larga de la definición de la función también sirve:

```
//Usa cstdlib e iostream
void Dinero::salida()
{
    long centavos_positivos, pesos, centavos;
    centavos_positivos = labs(todo_centavos);
    pesos = centavos_positivos/100;
    centavos = centavos_positivos%100;

    if (todo_centavos < 0)
        cout << "-$" << pesos << '.';
    else
        cout << "$" << pesos << '.';

    if (centavos < 10)
        cout << '0';
    cout << centavos;
}
```

Además, puede sobrecargar la función miembro `entrada` de modo que una llamada como:

```
monedero.entrada();
```

signifique lo mismo que

```
monedero.entrada(cin);
```

Y por supuesto, puede combinar esta mejora con las mejoras de los ejercicios de autoevaluación anteriores para elaborar una clase `Dinero` con muchas mejoras.

6. Si el usuario introduce `$-9.95` (en lugar de `-$9.95`), la función `entrada` leerá el signo `'$'` como valor de `un_char`, el `-9` como el valor de `pesos`, el `'.'` como el valor `punto_decimal`, y el `'9'` y el `'5'` como los valores de `digito1` y `digito2`. Esto significa que `pesos` será igual a `-9` y `centavos` igual a `95`, por lo que el monto `se establecerá` igual al valor que representa `-$9.00` más `0.95`, el cual es `-8.05`. Una manera de atrapar este problema es probar si el valor de `pesos` es negativo (dado que el valor `pesos` debe ser un valor absoluto). Para hacer esto, reescriba la porción del mensaje de error de la siguiente manera:

```
if ( un_char != '$' || punto_decimal != '.'
       || !isdigit(digito1) || !isdigit(digito2)
       || pesos < 0 )    <-------- Nuevo
{
       cout <<"Error formato ilegal para introducir dinero\n";
       exit(1);
}
```

Aún así, este código no mostrará un mensaje de error para una entrada incorrecta con una cantidad igual a cero pesos, como en `$-0.95`. Sin embargo, con el material que hemos aprendido hasta aquí, aunque sería posible realizar una prueba para este caso, el código se complicaría mucho y se dificultaría aún más su lectura.

7.
```
#include <iostream>
using namespace std;
int main()
{
    int x;
    cin >> x;
```

```
    cout << x << endl;
    return 0;
}
```

Si la computadora interpreta la entrada con ceros (0) a la izquierda como un número en base ocho, entonces con los datos de entrada 077, la salida deberá ser `63`. La salida deberá ser `77` si el compilador no interpreta los datos con un cero a la izquierda que indique que está en base ocho.

8. La única modificación de la versión que aparece en el cuadro 8.3 es que el modificador *const* se agrega al encabezado de la función, así que la definición es

```
double Dinero::obtiene_valor() const
{
     return (todo_centavos * 0.01);
}
```

9. La función miembro `entrada` modifica el valor del objeto que la invoca, por lo cual el compilador mostrará un mensaje de error si agrega el modificador *const*.

10. *Similitudes*: El método de llamada de cada parámetro impide que se modifiquen los argumentos del objeto que hace la llamada. *Diferencias*: La llamada por valor hace una copia de los argumentos del objeto que hace la llamada, de modo que utiliza más memoria que una llamada por referencia constante.

11. En la declaración *const int* `x=17;`, la palabra reservada *const* promete al compilador que el código escrito por el autor no modificará el valor de `x`.

    En la declaración *int* `f()` *const*;, la palabra reservada *const* es una promesa al compilador de que el código escrito por el autor para implementar la función `f` no modificará cosa alguna en el objeto que hace la llamada.

    En la declaración *int* `g(const A& x);`, la palabra reservada `const` es una promesa al compilador de que el código escrito por el autor de la clase no modificará el argumento conectado por medio de `x`.

12. En la diferencia entre un operador binario (como +, *, /, etcétera.) y una función, se involucra la sintaxis de la forma en que se hacen las llamadas. En la llamada a una función, los argumentos se proporcionan entre paréntesis después del nombre de la función. Con un operador, los argumentos se proporcionan antes y después del operador. Además se debe utilizar la palabra reservada *operator* en la declaración y en la definición de un operador sobrecargado.

13. Aquí mostramos la definición modificada de la clase `Dinero`. La parte en pantalla es nueva. Omitimos algunos comentarios para ahorrar espacio, pero todos los comentarios que se encuentran en el cuadro 8.5 se deben incluir en esta definición.

```
class Dinero
{
public:
    friend Dinero operator +(const Dinero& monto1,
                             const Dinero& monto2);
    friend bool operator ==(const Dinero& monto1,
                            const Dinero& monto2);
    friend bool operator < (const Dinero& monto1,
                            const Dinero& monto2);
    //Precondición: monto1 y monto2 han recibido
    //valores.
    //Devuelve true si monto1 es menor que monto2;
    //de lo contrario, devuelve falso.
```

```
    Dinero(long pesos, int centavos);
    Dinero(long pesos);
    Dinero ();
    double obtiene_valor() const;
    void entrada (istream& ins);
    void salida(ostream& outs) const;
private:
    long todo_centavos;
};
```

También debe modificar la siguiente definición del operador sobrecargado <:

```
bool operator < (const Dinero& monto1,
                                    const Dinero& monto2)
{
     return (monto1.todo_centavos < monto2.todo_centavos);
}
```

14. Aquí mostramos la definición modificada de la clase `Dinero`. La parte en pantalla es nueva. Omitimos algunos comentarios para ahorrar espacio, pero todos los comentarios que se encuentran en el cuadro 8.5 se deben incluir en esta definición. Incluimos las modificaciones en los ejercicios anteriores dentro de esta respuesta, dado que es natural utilizar el operador sobrecargado < dentro de la definición del operador <= sobrecargado.

```
class Dinero
{
public:
    friend Dinero operator +(const Dinero& monto1,
                             const Dinero& monto2);
    friend bool operator ==(const Dinero& monto1,
                            const Dinero& monto2);
    friend bool operator < (const Dinero& monto1,
                            const Dinero& monto2);
    //Precondición: monto1 y monto2 han recibido
    //valores.
    //Devuelve true si monto1 es menor que monto2;
    //de lo contrario, devuelve false.
    friend bool operator <= (const Dinero& monto1,
                            const Dinero& monto2);
    //Precondición: monto1 y monto2 han recibido
    //valores.
    //Devuelve true si monto1 es menor o igual que
    //monto2; de lo contrario, devuelve false.
    Dinero(long pesos, int centavos);
    Dinero(long pesos);
    Dinero();
    double obtiene_valor() const;
    void entrada (istream& ins);
    void salida(ostream& outs) const;
private:
    long todo_centavos;
};
```

También debe modificar la siguiente definición del operador sobrecargado <= (así como la definición del operador sobrecargado < que se proporciona en el ejercicio anterior):

```
bool operator <= (const Dinero& monto1,
                                    const Dinero& monto2)
{
    return ((monto1.todo_centavos < monto2.todo_centavos)
            ||(monto1.todo_centavos == monto2.todo_centavos));
}
```

15. Cuando se sobrecarga un operador, al menos uno de los argumentos del operador debe ser de un tipo de clase. Esto evita que se modifique el comportamiento de + para los enteros. En realidad, este requerimiento evita que se modifique el efecto de cualquier operador dentro de cualquier tipo predefinido.

16.

```
//Utiliza cmath (para floor)
Dinero::Dinero(double monto)
{
     todo_centavos = floor(monto*100);
}
```

Esta definición sólo descarta cualquier monto que sea menor a un centavo. Por ejemplo, convierte `12.34999` en el entero `1234`, el cual representa un monto de `$12.34`. Es posible definir el constructor para que haga otras cosas con cualquier fracción de un centavo.

17. `istream& operator>>(istream& ins, Pares& segundo)`

```
{
  char ch;
  ins >> ch;        //descarta el '(' inicial
  ins >> segundo.f;
  ins >> ch;        //descarta la coma ','
  ins >> segundo.s;
  ins >> ch;        //descarta el ')' final
  return ins;
}
ostream& operator>>(ostream& outs, const Pares& segundo)
{
  outs << '(';
  outs << segundo.f;
  outs << ','; //Puede usar ", "
               //para obtener espacio adicional
  outs << segundo.s;
  outs << ')';
  return outs;
}
```

18.

```
//Usa iostream;
istream& operator >>(istream& ins, Porcentaje& el_objeto)
{
    char signo_porcentaje;
    ins >> el_objeto.valor;
    ins >> signo_porcentaje; //Descarta el signo %.
   return ins;
}
```

```
//Utiliza iostream;
ostream& operator <<(ostream& outs,
                                  const Porcentaje& un_porcentaje)
{
    outs << un_porcentaje.valor << '%';
    return outs;
}
```

Proyectos de programación

1. Modifique la definición de la clase `Dinero` que se muestra en el cuadro 8.8 de manera que agregue lo siguiente:

   a) Sobrecargue los operadores `<`, `<=`, `>`, y `>=` para que se puedan aplicar al tipo `Dinero`. (*Sugerencia*: Vea el ejercicio de autoevaluación 13).

   b) Agregue la función miembro siguiente a la definición de la clase. (A continuación mostramos la definición de la función tal y como debe aparecer en la definición de la clase. La definición de la función en sí incluye el calificador `Dinero::`.)

```
Dinero porcentaje(int cantidad_porcentaje) const:
//Devuelve el porcentaje del monto de dinero en el
//objeto que hace la llamada. Por ejemplo, si cantidad_porcentaje es 10,
//entonces el valor devuelto es el 10% del monto de
//dinero representado por el objeto que hace la llamada.
```

   Por ejemplo, si `monedero` es un objeto de tipo `Dinero` cuyo valor representa un monto de $100.10, entonces la llamada

```
Monedero.porcentaje(10);
```

   devolverá el `10%` de `$100.10`; esto es, devolverá un valor de tipo `Dinero` que represente el monto `$10.01`.

2. En el ejercicio de autoevaluación 17 le pedimos que sobrecargara el operador `>>` y el operador `<<` para la clase `Pares`. Complete y pruebe este ejercicio. Implemente el constructor predeterminado y los constructores con uno y dos parámetros `int`. El constructor de un solo parámetro debe inicializar el primer miembro del par; el segundo miembro del par debe ser `0`.

   Sobrecargue el `operador` binario + para que sume pares de acuerdo con la siguiente regla

```
(a, b) + (c, d) = (a + c, b + d)
```

   Sobrecargue `operator−` de la misma forma.

   Sobrecargue `operator*` en `Pares` e `int` de acuerdo a la siguiente regla

```
(a, b) * c = (a * c, b * c)
```

   Escriba un programa que pruebe todas las funciones miembro y los operadores sobrecargados dentro de la definición de su clase.

3. En el ejercicio de autoevaluación 18 le pedimos que sobrecargara el operador `>>` y el operador `<<` para la clase `Porcentaje`. Complete y pruebe este ejercicio. Implemente el constructor predeterminado y el constructor con un parámetro de tipo `int`. Sobrecargue los operadores + y – para sumar y restar porcentajes. Además, sobrecargue el operador * para permitir la multiplicación de un porcentaje por un entero.

   Escriba un programa que pruebe todas las funciones miembro y los operadores sobrecargados de la definición de su clase.

4. Defina una clase para números racionales. Un número racional es un número que se puede representar como el cociente de dos enteros. Por ejemplo, 1/2, 3/4, 64/2, y así sucesivamente para todos los números racionales. (Por 1/2, etcétera., queremos indicar el significado común de la fracción, no la división entera que esta expresión produciría dentro de un programa en C++). Represente los números racionales como dos valores de tipo `int`, uno para el numerador y otro para el denominador. Llame a esta clase `Racional`.

   Incluya un constructor con dos argumentos que se puedan utilizar para establecer las variables miembro de un objeto a cualquier valor válido. Incluya además un constructor que contenga solamente un parámetro de tipo `int`; llame a este parámetro `numero_entero` y defina el constructor de tal forma que el objeto se inicialice con un número racional igual a `numero_entero/1`. Incluya además un constructor predeterminado que inicialice un objeto con 0 (esto es, con 0/1).

   Sobrecargue los operadores de entrada y salida: >> y <<. Los números se escribirán y se mostrarán en la forma `1/2`, `15/32`, `300/401`, y así sucesivamente. Observe que el numerador, el denominador o ambos pueden contener un signo menos, por lo que `-1/2`, `15/-32`, y `-300/-401` también son posibles entradas. Sobrecargue todos los siguientes operadores de manera que se apliquen de manera correcta al tipo `Racional`: ==, <, <=, >, >=, +, −, *, y /. Además escriba un programa para probar su clase.

   Sugerencias: Dos números racionales *a/b* y *c/d* son iguales si *a*d* es igual a *c*b*. Si *b* y *d* son números racionales *positivos*, *a/b* es menor que *c/d* siempre y cuando *a*d* sea menor que *c*b*. Debe incluir una función para normalizar los valores almacenados de modo que, después de la normalización, el denominador sea positivo y el numerador y el denominador sean lo más pequeños posibles. Por ejemplo, después de la normalización, `4/-8` se representaría como `-1/2`. Además, debería también escribir un programa de prueba para su clase.

5. Defina una clase para los números complejos. Un número complejo es un número de la forma

   a + *b***i*

   en donde para nuestros fines, *a* y *b* son números de tipo `double`, e *i* es un número que representa la cantidad $\sqrt{-1}$. Represente un número complejo como dos valores de tipo `double`. Nombre a las variables miembro `real` e `imaginario`. (La variable para el número que se multiplica por *i* es aquella que se llama `imaginario`). Llame a la clase `Complejo`.

   Incluya un constructor con dos parámetros de tipo `double` que se puedan utilizar para asignar cualquier valor a las variables miembro de un objeto. Incluya además un constructor con un solo parámetro de tipo `double`; llame a este parámetro `parte_real` y defina un constructor de modo que el objeto se inicialice con `parte_real + 0*i`. Incluya además un constructor predeterminado que inicialice un objeto con `0` (esto es, con `0 + 0*i`). Sobrecargue todos los operadores siguientes para que se apliquen correctamente al tipo `Complejo`: ==, +, −, *, >> y <<. Escriba un programa de prueba para su clase.

   Sugerencias: Para sumar o restar dos números complejos, sume o reste las dos variables miembro de tipo `double`. El producto de dos números complejos se da mediante la siguiente fórmula:

   (*a* + *b***i*)*(*c* + *d***i*) == (*a***c* − *b***d*) + (*a***d* + *b***c*)**i*

   En el archivo de interfaz, deberá definir una constante `i` de la siguiente manera:

   ```
   const Complejo i(0, 1);
   ```

   Esta constante definida `i` será la misma que la *i* que explicamos con anterioridad.

9 Compilación por separado y espacios de nombres

9 Compilación por separado y espacios de nombres

De mi propia biblioteca con volúmenes que aprecio más allá de mi ducado.

WILLIAM SHAKESPEARE, *La tempestad*

Introducción

En este capítulo cubriremos dos temas relacionados con la manera en que se puede organizar un programa de C++ en piezas separadas. En la sección 9.1 que trata acerca de la compilación por separado veremos cómo puede distribuirse un programa en C++ a través de varios archivos, de manera que cuando cambien ciertas partes del programa sólo se tengan que volver a compilar esas partes. Además, podemos utilizar las partes separadas con mayor facilidad en otras aplicaciones.

En la sección 9.2 hablaremos sobre los espacios de nombres, que presentamos en el capítulo 2. Los espacios de nombres nos permiten reutilizar los nombres de las clases, las funciones y otros elementos mediante el proceso de calificar los nombres para indicar distintos usos. Los espacios de nombres dividen el código en secciones, de manera que éstas puedan reutilizar los mismos nombres con distintos significados. Los espacios de nombres permiten un tipo de significado local para los nombres, el cual es más general que las variables locales.

Prerrequisitos

En este capítulo utilizaremos material de los capítulos 2 al 8.

9.1 Compilación por separado

Tu "si" es el único pacificador;
hay mucha virtud en un "si".
William Shakespeare, *Como gustes*

C++ cuenta con recursos para dividir un programa en partes que se mantienen en archivos separados, se compilan por separado y luego se enlazan entre sí cuando se ejecuta el programa (o justo antes). Podemos colocar la definición para una clase (y sus definiciones de funciones asociadas) en archivos que estén separados de los programas que utilizan esa clase. De esta forma, podemos construir una biblioteca de clases para que muchos programas puedan utilizar esa misma clase. Se puede compilar la clase una vez y luego utilizarla en muchos programas distintos, de la misma forma en que se utilizan las bibliotecas predefinidas (como las que tienen los archivos de encabezado `iostream` y `cstdlib`). Es más, podemos definir la clase en sí en dos archivos, de manera que la especificación de lo que hace la clase esté separada de la forma en que ésta se implementa. Si usted define su clase en base a los lineamientos que le hemos proporcionado y sólo modifica la implementación de la clase, entonces sólo tendrá que volver a compilar el archivo que contiene la implementación de su clase. Los demás archivos, incluyendo los que contienen los programas que utilizan la clase, no necesitan modificarse ni volver a compilarse. En esta sección le diremos cómo llevar a cabo esta compilación de clases por separado.

Repaso de los ADT

Hay que recordar que un ADT (tipo de datos abstracto) es una clase que se ha definido de tal manera que se separen la interfaz y la implementación de la clase. Todas sus definiciones de clases deberían ser ADT. Para poder definir una clase como ADT, hay que separar la especificación de la manera en que el programador utiliza la clase de los detalles que especifican cómo se implementa esa clase. Esta separación debe ser tan completa que se pueda modificar la implementación sin necesidad de modificar cualquier programa que utilice la clase de ninguna forma. Para poder asegurar esta separación podemos basarnos en estas tres reglas:

1. Haga que todas las variables miembro sean miembros privados de la clase.
2. Haga que cada una de las operaciones básicas para el ADT (la clase) sean una función miembro pública de la clase, una función friend, una función ordinaria o un operador sobrecargado. Agrupe la definición de la clase y las declaraciones de funciones y operadores. A este grupo, junto con sus comentarios complementarios, se le conoce como la **interfaz** para el ADT. Se debe especificar completamente cómo se va a utilizar cada función u operador en un comentario que se proporcione junto con la clase o con la declaración de la función o del operador.

interfaz

3. Impida el acceso a la implementación de las operaciones básicas para el programador que utilice el tipo de datos abstracto. La **implementación** consiste en las definiciones de las funciones y las definiciones de los operadores sobrecargados (junto con cualquier función de ayuda u otro elemento adicional que requieran estas definiciones).

implementación

En C++, la mejor manera de asegurarse de seguir estas reglas es mediante la colocación de la interfaz y la implementación de la clase ADT en archivos separados. Como es de imaginar, al archivo que contiene la interfaz se le conoce a menudo como **archivo de interfaz** y al que contiene la implementación se le conoce como **archivo de implementación**. Los detalles exactos de cómo preparar, compilar y utilizar estos archivos varían un poco de una versión de C++ a otra, pero el esquema básico es el mismo en todas las versiones de C++.

archivo de interfaz y archivos de implementación

En especial, los detalles de lo que deben incluir lo archivos son los mismos en todos los sistemas. Lo único que varía son los comandos que se deben utilizar para compilar y enlazar estos archivos. En el siguiente `Caso de estudio` mostraremos los detalles sobre lo que deben incluir estos archivos.

Los miembros privados son parte de la implementación.

Una clase ADT tiene variables miembro privadas. Estas variables miembro (y las funciones miembro privadas) presentan un problema para nuestra filosofía básica de colocar la interfaz y la implementación de un ADT en archivos separados. La parte pública de la definición de la clase para un ADT es parte de la interfaz para el ADT, pero la parte privada es parte de la implementación. Esto representa un problema, ya que C++ no nos permite dividir la definición de la clase entre dos archivos. Por ende, se requiere cierto tipo de compromiso. El único compromiso sensible (el que utilizaremos) es colocar la definición completa de la clase en el archivo de interfaz. Como un programador que utilice la clase ADT no puede utilizar ninguno de los miembros privados de esa clase, éstos (en efecto) seguirán ocultos para el programador.

ADT

A un tipo de datos se le denomina tipo de datos abstractos (ADT) si los programadores que utilizan ese tipo de datos no tienen acceso a los detalles sobre la implementación de los valores y las operaciones. Todas las clases que usted defina deberán ser ADT. Una clase ADT es una clase que se define en base a las buenas prácticas de programación, en las que la interfaz y la implementación se separan de la clase. (Cualquier operación básica que no sea miembro de la clase, como los operadores sobrecargados, se considera parte del ADT, aún y cuando tal vez no formen oficialmente parte de la definición de la clase.)

CASO DE ESTUDIO `TiempoDigital`: Una clase que se compila por separado

archivo de interfaz

El cuadro 9.1 contiene el archivo de interfaz para una clase ADT llamada `TiempoDigital`. Esta clase contiene valores que representan las horas del día, como 9:30. Sólo los miembros públicos de la clase son parte de la interfaz. Los miembros privados son parte de la implementación, aun y cuando se encuentran en el archivo de interfaz. La etiqueta *`private:`* nos advierte que estos miembros privados no forman parte de la interfaz pública. Todo lo que un programador necesita saber para poder utilizar el ADT `TiempoDigital` se explica en el comentario al principio del archivo, y en los comentarios que están en la sección pública de la definición de la clase. Esta interfaz indica al programador cómo utilizar las dos versiones de la función miembro llamada `avanza`, los constructores y los operadores sobrecargados ==, >> y <<. La función miembro llamada `avanza`, los operadores sobrecargados y la instrucción de asignación son las únicas formas mediante las cuales un programador puede manipular objetos y valores de esta clase. Como se indica en el comentario de la parte superior del archivo de interfaz, este ADT utiliza notación de 24 horas; por ejemplo, si se introduce 1:30 PM se muestra como 13:30. En los comentarios que se proporcionan con las funciones miembro se incluyen éste y otros detalles que debe conocer para poder utilizar la clase `TiempoDigital` con efectividad.

archivos de encabezado

Hemos colocado la interfaz en un archivo llamado `tiempod.h`. El sufijo `.h` indica que éste es un archivo de encabezado. Un archivo de interfaz siempre es un archivo de encabezado y, por ende, siempre termina con ese sufijo `.h`. Cualquier programa que utilice la clase `TiempoDigital` deberá contener una directiva `include` como la siguiente, en la que se nombra este archivo:

```
#include "tiempod.h"
```

CUADRO 9.1 *Archivo de interfaz para* `TiempoDigital` *(parte 1 de 4)*

```
//Archivo de encabezado tiempod.h: Esta es la INTERFAZ para la clase TiempoDigital.
//Los valores de este tipo son horas del día. Estos valores se introducen
//y se muestran en notación de 24 horas, como en 9:30 para las 9:30 AM y 14:45
//para las 2:45 PM.
#include <iostream>
using namespace std;

class TiempoDigital
{
public:
    friend bool operator ==(const TiempoDigital& tiempo1, const TiempoDigital& tiempo2);
    //Regresa verdadero si tiempo1 y tiempo2 representan la misma hora;
    //de lo contrario, regresa falso.

    TiempoDigital(int the_hour, int el_minuto);
    //Precondición: 0 <= la_hora <= 23 y 0 <= el_minuto <= 59.
    //Inicializa el valor de tiempo con la_hora y el_minuto.

    TiempoDigital( );
    //Inicializa el valor de tiempo con 0:00 (que equivale a media noche).

    void avanza(int minutos_agregados);
    //Precondición: El objeto tiene un valor de tiempo.
    //Postcondición: El tiempo se ha cambiado a minutos_agregados minutos después.

    void avanza(int horas_agregadas, int minutos_agregados);
    //Precondición: El objeto tiene un valor de tiempo.
    //Postcondición: El valor de tiempo ha avanzado
    //horas_agregadas horas más minutos_agregados minutos.

    friend istream& operator >>(istream& ins, TiempoDigital& the_object);
    //Sobrecarga el operador >> para valores de entrada de tipo TiempoDigital.
    //Precondición: Si ins es un flujo de entrada de archivo, entonces ins ya
    //se ha conectado a un archivo.

    friend ostream& operator <<(ostream& outs, const TiempoDigital& el_objeto);
    //Sobrecarga el operador << para mostrar valores de tipo TiempoDigital.
    //Precondición: Si outs es un flujo de salida de archivo, entonces outs
    //ya se ha conectado a un archivo.
private:
    int hora;
    int minuto;
};
```

Para la definición de los tipos `istream` y `ostream`, *que se utilizan como tipos de los parámetros.*

Esta es parte de la implementación. No es parte de la interfaz. La palabra `private` *indica que no es parte de la interfaz pública.*

include

Al escribir una directiva `include`, debemos indicar si el archivo de encabezado ya está predefinido, o si lo escribimos nosotros. Si es predefinido, debemos escribir el nombre del archivo de encabezado encerrado entre los signos < y >, como `<iostream>`. Si nosotros escribimos dicho archivo, entonces debemos encerrar su nombre entre comillas, como `"tiempod.h"`. Con esta distinción indicamos al compilador en dónde debe buscar el archivo de encabezado. Si éste va encerrado entre < y >, el compilador busca en cualquier parte en donde se encuentren los archivos de encabezado predefinidos en su implementación de C++. Si el nombre del archivo de encabezado va entre comillas, el compilador busca en el directorio actual o en cualquier parte en que se encuentren los archivos de encabezado definidos por el programador en su sistema.

Cualquier programa que utilice nuestra clase `TiempoDigital` debe contener la directiva `include` anterior que nombra el archivo de encabezado `tiempod.h`. Esto es suficiente para poder compilar el programa, pero no para poder ejecutarlo. Para poder ejecutar el programa debemos escribir (y compilar) las definiciones de las funciones miembro y de los operadores sobrecargados. Hemos colocado estas definiciones de las funciones y los operadores en otro archivo, al cual lo llamaremos **archivo de implementación**. Aunque no es algo requerido por la mayoría de los compiladores, es tradicional dar al archivo de interfaz y al archivo de implementación el mismo nombre. No obstante, los dos archivos terminan con distintos sufijos. Hemos colocado la interfaz para nuestra clase ADT en el archivo llamado `tiempod.h` y la implementación para nuestra clase ADT en un archivo llamado `tiempod.cpp`. El sufijo que se utiliza para el archivo de implementación dependerá de la versión de C++. Para el archivo de implementación debe usar el mismo sufijo que utiliza para los archivos que contienen programas en C++. Si sus archivos de programa terminan en `.cxx`, entonces debe usar `.cxx` en lugar de `.cpp`. Si sus archivos de programa terminan en `.CPP`, entonces sus archivos de implementación deben terminar en `.CPP` en vez de `.cpp`. Nosotros utilizaremos `.cpp`, ya que la mayoría de los compiladores aceptan este sufijo para un archivo de código fuente en C++. En el cuadro 9.2 se muestra el archivo de implementación para nuestra clase ADT llamada `TiempoDigital`. Una vez que expliquemos cómo interactúan los diversos archivos para nuestro ADT entre sí, regresaremos al cuadro 9.2 para hablar sobre los detalles relacionados con las definiciones en este archivo de implementación.

archivo de implementación

nombres de archivos

Para poder utilizar la clase ADT `TiempoDigital` en un programa, éste debe contener la siguiente directiva `include`:

```
#include "tiempod.h"
```

Hay que tener en cuenta que tanto el archivo de implementación como el del programa deben contener esta directiva `include` que nombra al archivo de interfaz. Por lo general, al archivo que contiene el programa (es decir, el archivo que contiene la parte `main` del programa) se le conoce como **archivo de aplicación** o **archivo controlador**. El cuadro 9.3 contiene un archivo de aplicación con un programa muy simple que utiliza y demuestra el funcionamiento de la clase ADT `TiempoDigital`.

archivo de aplicación

Los detalles exactos acerca de cómo se ejecuta este programa completo, que consta de tres archivos, dependen del sistema que esté utilizando. No obstante, los detalles básicos son los mismos en todos los sistemas. Debe compilar el archivo de implementación y el archivo de aplicación que contiene la parte `main` de su programa. No compile el archivo de interfaz, que en este ejemplo es el archivo `tiempod.h` que se muestra en el cuadro 9.1. No necesita compilar este archivo, ya que el compilador piensa que el contenido de este archivo de interfaz ya está contenido en cada uno de los otros dos archivos. Recuerde que tanto el archivo de implementación como el archivo de aplicación contienen la directiva

compilación y ejecución del programa

```
#include "tiempod.h"
```

CUADRO 9.2 *Archivo de interfaz para* `TiempoDigital` *(parte 1 de 4)*

```
//Archivo de implementación tiempod.cpp (Tal vez su sistema requiera un
//sufijo distinto a .cpp): Esta es la IMPLEMENTACIÓN del ADT TiempoDigital.
//La interfaz para la clase TiempoDigital está en el archivo de encabezado dtime.h.
#include <iostream>
#include <cctype>
#include <cstdlib>
#include "tiempod.h"
using namespace std;

//Estas DECLARACIONES DE FUNCIONES son para usarse en la definición del
//operador de entrada >> sobrecargado:

void lee_hora(istream& ins, int& la_hora);
//Precondición: La siguiente entrada en el flujo ins es una hora en notación,
//como 9:45 o 14:45.
//Postcondición: la_hora se ha ajustado a la parte del tiempo
//correspondiente a la hora.
//Se ha descartado el signo de dos puntos y la siguiente entrada a leer es el minuto.

void lee_minuto(istream& ins, int& el_minuto);
//Lee el minuto del flujo ins después de que lee_hora ha leído la hora.

int digito_a_int(char c);
//Precondición: c es uno de los dígitos del '0' al '9'.
//Devuelve el entero para el dígito; por ejemplo, digito_a_int('3') devuelve un 3.

bool operator ==(const TiempoDigital& tiempo1, const TiempoDigital& tiempo2)
{
    return (tiempo1.hora == tiempo2.hora && tiempo1.minuto == tiempo2.minuto);
}

//Usa iostream y cstdlib:
TiempoDigital::TiempoDigital(int la_hora, int el_minuto)
{
    if (la_hora < 0 || la_hora > 23 || el_minuto < 0 || el_minuto > 59)
    {
        cout << "Argumento ilegal para el constructor de TiempoDigital.";
        exit(1);
    }

    else
    {
        hora = la_hora;
        minuto = el_minuto;
    }
}
```

CUADRO 9.2 *Archivo de interfaz para* `TiempoDigital` *(parte 2 de 4)*

```
TiempoDigital::TiempoDigital( ) : hora(0), minuto(0)
{
    //Cuerpo vacío de manera intencional.
}

void TiempoDigital::avanza(int minutos_agregados)
{
    int total_minutos = minuto + minutos_agregados;
    minuto = total_minutos%60;

    int ajuste_hora = total_minutos/60;
    hora = (hora + ajuste_hora)%24;
}

void TiempoDigital::avanza(int horas_agregadas, int minutos_agregados)
{
    hora = (hora + horas_agregadas)%24;
    avanza(minutos_agregados);
}

//Usa iostream:
ostream& operator <<(ostream& outs, const TiempoDigital& el_objeto)
{
    outs << el_objeto.hora << ':';
    if (el_objeto.minuto < 10)
        outs << '0';
    outs << el_objeto.minuto;
    return outs;
}

//Usa iostream:
istream& operator >>(istream& ins, TiempoDigital& el_objeto)
{
    lee_hora(ins, el_objeto.hora);
    lee_minuto(ins, el_objeto.minuto);
    return ins;
}

int digito_a_int(char c)
{
    return (static_cast<int>(c) - static_cast<int>('0') );
}
```

CUADRO 9.2 *Archivo de interfaz para* `TiempoDigital` *(parte 3 de 4)*

```
//Usa iostream, cctype y cstdlib:
void lee_minuto(istream& ins, int& el_minuto)
{
    char c1, c2;
    ins >> c1 >> c2;

    if (!(isdigit(c1) && isdigit(c2)))
    {
        cout << "Error: entrada ilegal para lee_minuto\n";
        exit(1);
    }

    el_minuto = digito_a_int(c1)*10 + digito_a_int(c2);

    if (el_minuto < 0 || el_minuto > 59)
    {
        cout << "Error: entrada ilegal para lee_minuto\n";
        exit(1);
    }
}

//Usa iostream, cctype y cstdlib:
void lee_hora(istream& ins, int& la_hora)
{
    char c1, c2;
    ins >> c1 >> c2;
    if ( !( isdigit(c1) && (isdigit(c2) || c2 == ':' ) ) )
    {
        cout << "Error: entrada ilegal para lee_hora\n";
        exit(1);
    }

    if (isdigit(c1) && c2 == ':')
    {
        la_hora = digito_a_int(c1);
    }
    else //(isdigit(c1) && isdigit(c2))
    {
        la_hora = digito_a_int(c1)*10 + digito_a_int(c2);
        ins >> c2;//descarta el ':'
        if (c2 != ':')
        {
            cout << "Error: entrada ilegal para lee_hora\n";
            exit(1);
        }
    }
```

CUADRO 9.2 *Archivo de interfaz para* `TiempoDigital` *(parte 4 de 4)*

```
    if ( la_hora < 0 || la_hora > 23 )
    {
        cout << "Error: entrada ilegal para lee_hora\n";
        exit(1);
    }
}
```

La compilación del programa invoca automáticamente a un procesador, el cual lee esta directiva `include` y la reemplaza con el texto del archivo `tiempod.h`. Por ende, el compilador ve el contenido de `tiempod.h` y es por ello que este archivo no necesita compilarse por separado. (De hecho, el compilador ve el contenido de `tiempod.h` dos veces: una cuando compila el archivo de implementación y la otra cuando compila el archivo de aplicación.) Esta copia del archivo `tiempod.h` sólo es conceptual. El compilador actúa como si se copiara el contenido de `tiempod.h` en cada archivo que tiene la directiva `include`. No obstante, si analiza ese archivo después de compilarlo, sólo encontrará la directiva `include`, no el contenido del archivo `tiempod.h`.

Una vez compilados el archivo de implementación y el de aplicación, aún necesita conectar estos archivos para que puedan trabajar en conjunto. A este proceso se le conoce como **enlazar** los archivos; esto se realiza mediante una herramienta separada llamada **enlazador**. Los detalles acerca de cómo llamar al enlazador dependen del sistema que se esté utilizando. Una vez enlazados los archivos, podrá ejecutar el programa. (A menudo el enlace se realiza de manera automática como parte del proceso de ejecutar el programa.)

enlace

Este proceso parece complicado, pero muchos sistemas cuentan con herramientas que se encargan de la mayor parte de estos detalles de manera automática o semi-automática. En cualquier sistema, los detalles se convierten rápidamente en rutina.

Los cuadros 9.1, 9.2 y 9.3 contienen un programa completo dividido en piezas, el cual se coloca en tres archivos distintos. En vez de ello, podríamos combinar el contenido de estos tres archivos en uno solo, para después compilarlo y ejecutarlo sin tener que lidiar con las directivas `include` y el proceso de enlazar archivos separados. ¿Por qué tomarse la molestia de usar tres archivos separados? Hay varias ventajas a la hora de dividir su programa en archivos separados. Como se tiene la definición y la implementación de la clase `TiempoDigital` en archivos que están separados del archivo de aplicación, podríamos utilizar esta clase en muchos programas distintos sin necesidad de volver a escribir la definición de la clase en cada uno de esos programas. Lo que es más, sólo habrá que compilar el archivo de implementación una vez, sin importar cuántos programas utilicen la clase `TiempoDigital`. Pero hay más ventajas todavía. Como la interfaz está separada de la implementación de nuestra clase ADT `TiempoDigital`, podemos modificar el archivo de implementación sin tener que cambiar ninguno de los programas que utilicen el ADT. De hecho, ni siquiera tendremos que volver a compilar el programa. Si modificamos el archivo de implementación, sólo tenemos que volver a compilar este archivo y volver a enlazar los archivos. Es bueno poder ahorrar un poco de tiempo en la recompilación de archivos, pero la gran ventaja es no tener que reescribir código. Podemos usar la clase ADT en muchos programas sin tener que escribir el código de esa clase en cada programa.

¿Por qué archivos separados?

CUADRO 9.3 *Archivo de aplicación que utiliza* TiempoDigital

```
//Archivo de aplicación tiempodemo.cpp: (Tal vez su sistema requiera algún sufijo
//distinto a .cpp.) Este programa demuestra el uso de la clase TiempoDigital.
#include <iostream>
#include "tiempod.h"
using namespace std;

int main( )
{
    TiempoDigital reloj, reloj_ant;

    cout << "Introduzca la hora en notacion de 24 horas: ";
    cin >> reloj;

    reloj_ant = reloj;
    reloj.avanza(15);
    if (reloj == reloj_ant)
        cout << "Algo esta mal.";
    cout << "Usted escribio " << reloj_ant << endl;
    cout << "15 minutos despues la hora sera "
         << reloj << endl;

    reloj.avanza(2, 15);
    cout << "2 horas y 15 minutos despues de eso\n"
         << "la hora sera "
         << reloj << endl;

    return 0;
}
```

Diálogo de ejemplo

```
Introduzca la hora en notacion de 24 horas: 11:15
Usted escribio 11:15
15 minutos despues la hora sera 11:30
2 horas y 15 minutos despues de eso
la hora sera 13:45
```

Podemos modificar la implementación de la clase ADT sin tener que reescribir la parte de cualquier programa que utilice esa clase.

Definición de una clase en archivos separados: Resumen

Podemos definir una clase y colocar la definición de esa clase y la implementación de sus funciones miembro en archivos separados. De esta forma, podemos compilar la clase en forma separada de cualquier programa que la utilice, y podemos usar esta misma clase en cualquier cantidad de programas distintos. La clase y el programa que la utiliza se colocan en tres archivos de la siguiente manera:

1. Coloque la definición de la clase en un archivo de encabezado, al cual se le conoce como **archivo de interfaz**. El nombre de este archivo de encabezado termina en `.h`. El archivo de interfaz también contiene las declaraciones para cualquier función y operador sobrecargado que defina las operaciones básicas para la clase, pero que no estén listados en la definición de la misma. Incluya comentarios que expliquen cómo se utilizan todas estas funciones y operadores.
2. Las definiciones de todas las funciones y los operadores sobrecargados que mencionamos en el paso 1 (sin importar que sean miembros o friends, o ninguna de las dos cosas) se colocan en otro archivo, al cual se le conoce como **archivo de implementación**. Este archivo debe contener una directiva `include` que nombre al archivo de interfaz que describimos antes. Esta directiva `include` utiliza comillas alrededor del nombre del archivo, como en el siguiente ejemplo:

   ```
   #include "tiempod.h"
   ```

   Por tradición, el archivo de interfaz y el archivo de implementación tienen el mismo nombre, pero terminan con distintos sufijos. El archivo de interfaz termina en `.h`. El archivo de implementación termina con el mismo sufijo que se utiliza para los archivos que contienen un programa completo en C++. El archivo de implementación se compila por separado antes de usarlo en cualquier programa.
3. Cuando quiera utilizar la clase en un programa, coloque la parte `main` del programa (y cualquier definición de función adicional, declaraciones de constantes, etcétera) en otro archivo, al cual se le conoce como **archivo de aplicación**. Este archivo también debe contener una directiva `include` que nombre al archivo de interfaz, como en el siguiente ejemplo:

   ```
   #include "tiempod.h"
   ```

   El archivo de aplicación se compila de manera separada del archivo de implementación. Podemos escribir cualquier número de estos archivos de aplicación para usarlos con un par de archivos de interfaz y de implementación. Para ejecutar un programa completo, primero es necesario enlazar el código objeto que se produce al compilar el archivo de aplicación y el que se produce al compilar el archivo de implementación. (En algunos sistemas el enlace tal vez se realice de manera automática o semi-automática.)

detalles de implementación

Ahora que hemos explicado cómo se utilizan los diversos archivos en nuestra clase ADT y nuestro programa, veamos la implementación de nuestra clase ADT (cuadro 9.2) con más detalle. La mayoría de los detalles de implementación son simples, pero hay dos cosas que ameritan un comentario. Observe que la función miembro llamada `avanza` está sobrecargada, por lo cual tiene dos definiciones. Además observe que la definición para el

operador de extracción >> sobrecargado (entrada) utiliza dos "funciones de ayuda" llamadas `lee_hora` y `lee_minuto`, y que estas dos funciones de ayuda utilizan una tercera función de ayuda llamada `digito_a_int`. Hablemos ahora sobre estos puntos.

`avanza`

La clase `TiempoDigital` (cuadros 9.1 y 9.2) tiene dos funciones miembro llamadas `avanza`. Una versión toma un solo argumento, el cual es un entero que proporciona el número de minutos que se debe avanzar el tiempo. La otra versión toma dos argumentos: uno para el número de horas y el otro para el número de minutos, y avanza el tiempo esa cantidad de horas más esa cantidad de minutos. Observe que la definición de la versión de `avanza` con dos argumentos incluye una llamada a la versión de `avanza` con un argumento. Analice la definición de la versión con dos argumentos que se proporciona en el cuadro 9.2. En primer lugar, la hora se avanza la cantidad de `horas_agregadas` horas, y luego la versión de un solo argumento de `avanza` se utiliza para avanzar el tiempo una cantidad adicional de `minutos_agregados` minutos. Al principio esto puede parecer extraño, pero es completamente legal. Las dos funciones llamadas `avanza` son dos funciones distintas que, en cuanto a lo que al compilador respecta, tienen por coincidencia el mismo nombre. La situación en este caso no es distinta a la que se daría si una de las dos versiones de la función sobrecargada `avanza` se hubiera llamado `otra_avanza`.

Ahora hablemos sobre las funciones de ayuda. Las funciones `lee_hora` y `lee_minuto` leen la entrada un carácter a la vez y después la convierten en valores enteros que se colocan en las variables miembro `hora` y `minuto`. Las funciones `lee_hora` y `lee_minuto` leen la hora y el minuto un dígito a la vez, por lo que leen valores de tipo *`char`*. Esto es más complicado que leer la entrada como valores *`int`*, pero nos permite llevar a cabo la comprobación de errores para ver si la entrada está formada de manera correcta y para emitir un mensaje de error si la entrada no está bien formada. Estas funciones de ayuda llamadas `lee_hora` y `lee_minuto` utilizan otra función de ayuda llamada `digito_a_int`, la cual es igual a la función `digito_a_int` que utilizamos en nuestra definición de la clase `Dinero` en el cuadro 8.3. La función `digito_a_int` convierte un dígito tal como `'3'` en un número tal como `3`.

Componentes reutilizables

Una clase ADT que se desarrolla y se codifica en archivos separados es un componente de software que puede utilizarse una y otra vez en varios programas distintos. La **reutilización**, como se da el caso en estas clases ADT, es una meta importante que se busca cumplir al diseñar componentes de software. Un componente reutilizable ahorra esfuerzo, ya que no necesita volver a diseñarse, codificarse y evaluarse para cada una de las aplicaciones en las que se vaya a utilizar. Además es muy probable que un componente reutilizable sea más confiable que un componente que se utilice sólo una vez, por dos razones. En primer lugar, puede darse el lujo de invertir más tiempo y esfuerzo en un componente si éste se va a utilizar muchas veces. En segundo lugar, si el componente se utiliza una y otra vez significa que se prueba una y otra vez. Cada vez que se usa el componente de software constituye una prueba de ese componente. El uso de un componente de software muchas veces en una variedad de contextos es una de las mejores formas de descubrir cualquier error remanente en el software.

Uso de #ifndef

Acabamos de ver un método para colocar un programa en tres archivos: dos para la interfaz y la implementación de la clase, y uno para la parte correspondiente a la aplicación del programa. Un programa puede distribuirse en más de tres archivos. Por ejemplo, un programa podría utilizar varias clases en donde cada una de ellas podría mantenerse en un par de archivos por separado. Supongamos que tenemos un programa esparcido entre varios archivos y que más de un archivo tiene una directiva `include` para un archivo de interfaz de clase tal como el siguiente:

```
#include "tiempod.h"
```

Bajo estas circunstancias, podríamos tener archivos que incluyan otros archivos, y estos otros archivos podrían a su vez incluir a otros más. Esto nos puede llevar fácilmente a una situación en la que, en efecto, un archivo contenga las definiciones de `tiempod.h` más de una vez. C++ no permite definir una clase más de una vez, aún si las definiciones repetidas son idénticas. Lo que es más, si utilizamos el mismo archivo de encabezado en muchos proyectos distintos, es casi imposible llevar el registro de si se ha incluido la definición de la clase más de una vez. Para evitar este problema, C++ nos proporciona una manera de marcar una sección de código para decir "si ya se incluyó esto una vez, no debe incluirse de nuevo". La forma de hacer esto es bastante intuitiva, aunque la notación parezca un poco extraña si no estamos acostumbrados a ella. Examinaremos un ejemplo, explicando los detalles sobre la marcha.

La siguiente directiva "define" `TIEMPOD_H`:

```
#define TIEMPOD_H
```

Lo que esto significa es que el preprocesador del compilador coloca a `TIEMPOD_H` en una lista para indicar que lo ha visto. `Define` tal vez no sea la palabra ideal aquí, porque no estamos definiendo algún significado para `TIEMPOD_H`, simplemente lo estamos colocando en una lista. El punto importante es que podemos utilizar otra directiva para probar si `TIEMPOD_H` se ha definido o no, y de esta manera podemos probar si ya se ha procesado una sección de código o no. Podemos utilizar cualquier identificador (que no sea palabra clave) en lugar de `DTIEMPO_H`, aunque más adelante veremos que existen convenciones estándar acerca de cuál identificador debemos utilizar.

La siguiente directiva evalúa si se ha definido `TIEMPOD_H` o no:

```
#ifndef TIEMPOD_H
```

Si ya se ha definido `TIEMPOD_H`, entonces se omite todo lo que haya entre esta directiva y la primera ocurrencia de la siguiente directiva:

```
#endif
```

(Una manera equivalente de declarar esto, que podría aclarar la forma en que se deletrean las directivas, es la siguiente: si `TIEMPOD_H` no está definido, entonces el compilador procesará todo lo que haya hasta el siguiente `#endif`. Esta *no* es la razón de la `n` en `#ifndef`. Tal vez esto nos lleve a pensar si existe una directiva `#ifdef` así como la directiva `#ifndef`. En realidad si la hay, y tiene el significado obvio, pero en este libro no tenemos contemplado el uso de `#ifdef`.)

Ahora considere el siguiente código:

```
#ifndef TIEMPOD_H
#define TIEMPOD_H
<la definición de una clase>
#endif
```

Si este código se encuentra en un archivo llamado `tiempod.h`, entonces no importa cuántas veces nuestro programa contenga

```
#include "tiempod.h"
```

ya que la clase se definirá sólo una vez.

La primera vez que se procesa

```
#include "tiempod.h"
```

se define la bandera `TIEMPOD_H` y se define la clase. Supongamos ahora que el compilador se encuentra de nuevo con

```
#include "tiempod.h"
```

Cuando se procesa la directiva `include` esta segunda vez, la directiva

```
#ifndef TIEMPOD_H
```

indicará que hay que omitir todo hasta

```
#endif
```

y por consecuencia, la clase no se define de nuevo.

En el cuadro 9.4 hemos reescrito el archivo de encabezado `tiempod.h` que se muestra en el cuadro 9.1, sólo que esta vez utilizamos estas directivas para evitar múltiples

CUADRO 9.4 *Cómo evitar múltiples definiciones de una clase*

```
//Archivo de encabezado tiempod.h: Esta es la INTERFAZ para la clase TiempoDigital.
//Los valores de este tipo son horas del día. Los valores se introducen y se muestran en
//notación de 24 horas, como en 9:30 para las 9:30 AM y 14:45 para las 2:45 PM.

#ifndef TIEMPOD_H
#define TIEMPOD_H

#include <iostream>
using namespace std;

class TiempoDigital
{

<La definición de la clase TiempoDigital es la misma que en el cuadro 9.1.>

};

#endif //TIEMPOD_H
```

definiciones. Con la versión de `tiempod.h` que se muestra en el cuadro 9.4, si un archivo contiene la siguiente directiva `include` más de una vez, la clase `TiempoDigital` se definirá sólo una vez de todas formas:

```
#include "tiempod.h"
```

Podríamos usar algún otro identificador en lugar de `TIEMPOD_H`, pero la convención común es utilizar el nombre del archivo escrito todo en mayúsculas, con el guión bajo en vez del punto. Recomendamos seguir esta convención para que otros puedan leer su código con mayor facilidad y no tenga que recordar el nombre de bandera. De esta forma, el nombre de bandera se determina de manera automática y no hay nada arbitrario que recordar.

Estas mismas directivas pueden usarse para omitir código en otros archivos que no sean de encabezado, aunque en este libro no tendremos oportunidad de utilizar estas directivas más que en los archivos de encabezado.

TIP DE PROGRAMACIÓN

Definición de otras bibliotecas

No necesitamos definir una clase para utilizar la compilación por separado. Si tenemos una colección de funciones relacionadas que deseamos convertir en una biblioteca diseñada por nosotros, podemos colocar las declaraciones de las funciones y los comentarios complementarios en un archivo de encabezado y las definiciones de las funciones en un archivo de implementación, como vimos con las clases ADT. Después de eso, podremos utilizar esta biblioteca en nuestros programas de la misma forma en que utilizaríamos una clase que hayamos colocado en archivos separados.

Ejercicios de AUTOEVALUACIÓN

1. Suponga que va a definir una clase ADT para después utilizarla en un programa. Usted quiere dividir las partes correspondientes a la clase y al programa en archivos separados, como se describió en este capítulo. Especifique si cada uno de los siguientes elementos debe colocarse en el archivo de interfaz, en el archivo de implementación o en el de aplicación:

   a) La definición de la clase.

   b) La declaración de una función que servirá como operación del ADT, pero que no es miembro, ni friend de la clase.

   c) La declaración para un operador sobrecargado que servirá como operación del ADT, pero que no es miembro, ni friend de la clase.

   d) La definición para una función que servirá como operación del ADT, pero que no es miembro, ni friend de la clase.

   e) La definición para una función friend que servirá como operación del ADT.

   f) La definición para una función miembro.

   g) La definición para un operador sobrecargado que servirá como operación del ADT, pero que no es miembro, ni friend de la clase.

   h) La definición para un operador sobrecargado que servirá como operación del ADT y es friend de la clase.

   i) La parte `main` del programa.

2. ¿Cuál de los siguientes archivos tiene un nombre que termina en `.h`: el archivo de interfaz para una clase, el archivo de implementación para la clase o el archivo de aplicación que utiliza la clase?
3. Cuando se define una clase en archivos separados, hay un archivo de interfaz y un archivo de implementación. ¿Cuál de estos archivos necesita compilarse? (¿Ambos? ¿Ninguno? ¿Sólo uno? De ser así, ¿cuál de los dos?)
4. Suponga que define una clase en archivos separados y la usa en un programa. Ahora suponga que modifica el archivo de implementación. ¿Cuál de los siguientes archivos necesita volver a compilarse (en caso de ser necesario): el archivo de interfaz, el archivo de implementación o el archivo de aplicación?
5. Suponga que desea modificar la implementación de la clase `TiempoDigital` que se muestra en los cuadros 9.1 y 9.2. En sí, lo que desea modificar es la manera en que se registra el tiempo. En vez de utilizar las dos variables privadas `hora` y `minuto`, desea utilizar una sola variable *`int`* (privada), a la cual llamará `minutos`. En esta nueva implementación, la variable privada `minutos` registrará el tiempo como el número de minutos transcurridos desde la hora 0:00 (es decir, desde media noche). Por lo tanto, 1:30 se registrará como `90` minutos, ya que han transcurrido 90 minutos después de la media noche. Describa cómo necesita modificar los archivos de interfaz y de implementación que se muestran en los cuadros 9.1 y 9.2. No necesita escribir todos los archivos completos; sólo indique qué elementos necesita modificar y cómo (de una manera muy general) los modificaría.
6. ¿Cuál es la diferencia entre un ADT que se define en C++ y una clase que se define en C++?

9.2 Espacios de nombres

¿Qué hay en un nombre? Una rosa con
cualquier otro nombre tendría el mismo aroma.
William Shakespeare, *Romeo y Julieta*

espacio de nombres

Cuando un programa utiliza distintas clases y funciones escritas por distintos programadores, existe la posibilidad de que dos programadores utilicen el mismo nombre para dos cosas distintas. Los espacios de nombres son una manera de enfrentar este problema. Un espacio de nombres es una colección de definiciones de nombres, como las definiciones de clases y las declaraciones de variables.

Los espacios de nombres y las directivas `using`

Ya hemos utilizado el espacio de nombres llamado `std`. Este espacio de nombres contiene todos los nombres definidos en los archivos de la biblioteca estándar (como `iostream` y `cstdlib`) que utilizamos. Por ejemplo, cuando se coloca lo siguiente al principio de un archivo,

```
#include <iostream>
```

se colocan todas las definiciones de los nombres (como `cin` y `cout`) en el espacio de nombres `std`. Nuestro programa no sabrá nada acerca de los nombres en el espacio de nombres `std` a menos que le especifiquemos que está utilizando el espacio de nombres `std`. Hasta ahora, la única forma que conocemos para especificar el espacio de nombres `std` (o cualquier otro espacio de nombres) es mediante la siguiente directiva *`using`*:

```
using namespace std;
```

Una buena manera de saber por qué es conveniente incluir esta directiva *`using`* sería pensar por qué *`no`* sería conveniente incluirla. Si no se incluye esta directiva *`using`* para el espacio de nombres `std`, entonces podemos definir `cin` y `cout` para que tengan otro

significado distinto al significado estándar. (Tal vez usted quiera redefinir `cin` y `cout` porque desea que se comporten de una manera algo distinta a las versiones estándar.) Su significado estándar está en el espacio de nombres `std`, y si no utilizara la directiva *`using`* (o algo parecido) su código no sabría nada acerca del espacio de nombres `std`, de manera que su código sólo conocería las definiciones de `cin` y `cout` que usted les dé.

Cada pieza de código que escribimos se encuentra en algún espacio de nombres. Si no colocamos el código en un espacio de nombres específico, entonces el código se encuentra en un espacio de nombres conocido como el espacio de nombres global. Hasta ahora no hemos colocado ninguno de los códigos que hemos escrito en un espacio de nombres, por lo que todo nuestro código se encuentra en el espacio de nombres global. Este espacio de nombres no tiene una directiva *`using`* debido a que es el que siempre estamos utilizando. Podríamos decir que siempre hay una directiva *`using`* automática implícita que indica que estamos utilizando el espacio de nombres global.

espacio de nombres global

Podemos utilizar más de un espacio de nombres a la vez. Por ejemplo, siempre estamos usando el espacio de nombres global y por lo general también utilizamos el espacio de nombres `std`. ¿Qué ocurre si se define un nombre en dos espacios de nombres y estamos utilizando ambos espacios de nombres? Esto produce un error (ya sea un error de compilación o un error en tiempo de ejecución, dependiendo de los detalles exactos). Podemos tener el mismo nombre definido en dos espacios de nombres distintos, pero entonces sólo podríamos usar uno de esos espacios de nombres a la vez.[1] Sin embargo, esto no significa que no se puedan utilizar los dos espacios de nombres en el mismo programa. Podemos usar cada uno de ellos en distintos momentos en el mismo programa.

Por ejemplo, suponga que `ns1` y `ns2` son dos espacios de nombres, y suponga que `mi_funcion` es una función *`void`* sin argumentos que está definida en ambos espacios de nombres, pero se define de distintas formas en cada uno de los dos espacios de nombres. Entonces, lo siguiente sería legal:

```
{
  using namespace ns1;
  mi_funcion();
}
{
  using namespace ns2;
  mi_funcion();
}
```

La primera invocación utilizaría la definición de `mi_funcion` que se da en el espacio de nombres `ns1`, y la segunda invocación utilizaría la definición de `mi_funcion` que se da en el espacio de nombres `ns2`.

Recuerde que un bloque es una lista de instrucciones, declaraciones y posiblemente otro código, todo lo cual va encerrado entre corchetes `{}`. Una directiva *`using`* al principio de un bloque se aplica sólo a ese bloque. Por lo tanto, la primera directiva *`using`* se aplica sólo en el primer bloque, y la segunda directiva *`using`* se aplica sólo en el segundo bloque. La forma usual de parafrasear esto es decir que el alcance del espacio de nombres `ns1` es el primer bloque, mientas que el alcance del espacio de nombres `ns2` es el segundo bloque. Debido a esta regla de alcance, podemos usar dos espacios de nombres conflictivos en el mismo programa (como en un programa que contenga los dos bloques que describimos en el párrafo anterior).

alcance

Cuando se utiliza una directiva *`using`* en un bloque, por lo general viene siendo el bloque que consiste en el cuerpo de la definición de una función. Si coloca una directiva *`using`* al

[1] Como veremos más adelante en este capítulo, hay formas de usar dos espacios de nombres al mismo tiempo, incluso aunque contengan el mismo nombre, pero esto es un punto delicado que no nos concierne todavía.

principio de un archivo (como lo hemos hecho hasta ahora), entonces la directiva `using` se aplica a todo el archivo completo. Por lo general, una directiva `using` debe colocarse cerca del principio de un archivo, o al principio de un bloque.

Regla de alcance para las directivas `using`

El alcance de una directiva `using` es el bloque en el que aparece (o dicho de manera más precisa, desde la posición de la directiva `using` hasta el final del bloque). Si la directiva `using` se encuentra fuera de todos los bloques, entonces se aplica a todo el contenido del archivo que sigue después de la directiva `using`.

Creación de un espacio de nombres

agrupamiento de espacio de nombres

Para poder colocar cierto código en un espacio de nombres, sólo necesitamos colocarlo en una agrupación de espacio de nombres de la siguiente forma:

```
namespace Nombre_Espacio_nombres
{
    Cierto_codigo
}
```

Cuando incluimos una de estas agrupaciones en nuestro código, se dice que colocamos los nombres definidos en `Cierto_código` en el espacio de nombres `Nombre_Espaciode-nombres`. Estos nombres (en realidad, la definición de estos nombres) pueden estar disponibles mediante la directiva `using`

```
using namespace Nombre_Espacio_nombres;
```

Por ejemplo, el siguiente fragmento de código tomado del cuadro 9.5 coloca la declaración de una función en el espacio de nombres `savitch1`:

```
namespace savitch1
{
    void saluda( );
}
```

Si analiza de nuevo el cuadro 9.5, podrá ver que la definición de la función `saluda` también se coloca en el espacio de nombres `savitch1`. Esto se hace mediante el siguiente agrupamiento de espacio de nombres adicional:

```
namespace savitch1
{
    void saluda( )
    {
       cout << "Saludos desde el espacio de nombres savitch1.\n";
    }
}
```

Observe que puede tener cualquier número de estos agrupamientos de espacio de nombre para un solo espacio de nombres. En el cuadro 9.5 utilizamos dos agrupamientos de espacio de nombres para el espacio de nombres `savitch1` y otros dos agrupamientos para el espacio de nombres `savitch2`.

Cada uno de los nombres que se define en un espacio de nombres está disponible dentro del agrupamiento de espacio de nombres, pero los nombres también pueden estar dis-

ponibles para el código fuera del espacio de nombres. La declaración y definición de la función en el espacio de nombres `savitch1` puede hacerse disponible mediante la siguiente directiva `using`:

```
using namespace savitch1
```

como se muestra en el cuadro 9.5.

CUADRO 9.5 *Demostración de los espacios de nombres (parte 1 de 2)*

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

namespace savitch1
{
    void saluda( );
}

namespace savitch2
{
    void saluda( );
}

void gran_saludo( );

int main( )
{
    {
        using namespace savitch2;
        saluda( );
    }

    {
        using namespace savitch1;
        saluda( );
    }

    gran_saludo( );

    return 0;
}
```

Los nombres en este bloque utilizan las definiciones en los espacios de nombres `savitch2`, `std` *y en el espacio de nombres global.*

Los nombres en este bloque utilizan las definiciones en los espacios de nombres `savitch1`, `std` *y en el espacio de nombres global.*

Los nombres aquí sólo utilizan las definiciones en el espacio de nombres `std` *y en el espacio de nombres global.*

CUADRO 9.5 *Demostración de los espacios de nombres (parte 2 de 2)*

```
namespace savitch1
{
     void saluda( )
     {
          cout << "Saludos desde el espacio de nombres savitch1.\n";
     }
}

namespace savitch2
{
     void saluda( )
     {
          cout << "Saludos desde el espacio de nombres savitch2.\n";
     }
}

void gran_saludo( )
{
     cout << "Un gran saludo global!\n";
}
```

Diálogo de ejemplo

```
Saludos desde el espacio de nombres savitch2.
Saludos desde el espacio de nombres savitch1.
Un gran saludo global!
```

Ejercicios de AUTOEVALUACIÓN

7. Considere el programa que se muestra en el cuadro 9.5. ¿Podríamos utilizar el nombre `saluda` en lugar de `gran_saludo`?

8. En el ejercicio de autoevaluación 7, vimos que no se podía agregar una definición para la siguiente función (en el espacio de nombres global):

   ```
   void saluda( );
   ```

¿Puede agregar una definición para la siguiente declaración de función al espacio de nombres global?

```
void saluda(int cuantos );
```

9. ¿Puede un espacio de nombres tener más de un agrupamiento de espacio de nombres?

Calificación de los nombres

Suponga que se enfrenta a la siguiente situación: tiene dos espacios de nombres llamados `ns1` y `ns2`. Desea utilizar la función `fun1` definida en `ns1` y la función `fun2` definida en el espacio de nombres `ns2`. El problema es que tanto `ns1` como `ns2` definen una función llamada `mi_funcion`. (Suponga que ninguna de las funciones aquí descritas tienen argumentos, por lo que no se aplica la sobrecarga.) No sería conveniente utilizar lo siguiente:

```
using namespace ns1;
using namespace ns2;
```

Esto produciría definiciones conflictivas para `mi_funcion`.

Lo que necesitamos es una manera de decir que se va a utilizar `fun1` en el espacio de nombres `ns1` y `fun2` en el espacio de nombres `ns2` y no se va a utilizar nada más en los espacios de nombres `ns1` y `ns2`. La respuesta a este problema son las siguientes **declaraciones** *`using`*:

declaración `using`

```
using ns1::fun1;
using ns2::fun2;
```

Una declaración `using` de la forma

```
using Espacio_de_nombres::Un_nombre
```

hace que (la definición de) el nombre *`Un_nombre`* del espacio de nombres *`Espacio_de_nombres`* esté disponible, pero no hace que cualquier otro nombre en *`Espacio_de_nombres`* esté disponible.

Ya habíamos utilizado antes el operador de resolución de alcance, `::`. Por ejemplo, en el cuadro 9.2 utilizamos la siguiente definición de función:

```
void TiempoDigital::avanza(int horas_agregadas, int
minutos_agregados)
{
    hora = (hora + horas_agregadas)%24;
    avanza(minutos_agregados);
}
```

En este caso, el `::` significa que estamos definiendo la función `avanza` para la clase `TiempoDigital`, en vez de usar cualquier otra función llamada `avanza` de cualquier otra clase. De manera similar,

```
using ns1::fun1;
```

significa que estamos usando la función llamada `fun1` que está definida en el espacio de nombres `ns1`, en vez de cualquier otra definición de `fun1` en cualquier otro espacio de nombres.

Ahora suponga que desea usar el nombre `fun1` que está definido en el espacio de nombres `ns1`, pero sólo pretende usarlo una vez (o un pequeño número de veces). Para ello

puede nombrar la función (u otro elemento) mediante el uso del espacio de nombres y el operador de resolución de alcance, como en el siguiente ejemplo:

```
ns1::fun1( );
```

Esta forma se utiliza a menudo cuando se especifica un tipo de parámetro. Por ejemplo, considere

```
int obtiene_numero(std::istream flujo_entrada)
. . .
```

En la función `obtiene_numero`, el parámetro `flujo_entrada` es de tipo `istream`, en donde `istream` está definido en el espacio de nombres `std`. Si este uso del nombre del tipo `istream` es el único nombre que necesita del espacio de nombres `std` (o si todos los nombres que necesita se califican de manera similar con `std::`), entonces *no* necesita la directiva

```
using namespace std;
```

Un punto sutil acerca de los espacios de nombres (*opcional*)

Existen dos diferencias entre una declaración `using` tal como

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

y una *directiva* `using` tal como

```
using namespace std;
```

Las diferencias son:

1. Una declaración `using` (como `using std::cout`) hace que sólo esté disponible un nombre en el espacio de nombres para nuestro código, mientras que una directiva `using` (como `using namespace std;`) hace que estén disponibles todos los nombres en un espacio de nombres.
2. Una declaración `using` introduce un nombre (como `cout`) en nuestro código, de manera que no pueda utilizarse ese nombre de ninguna otra forma. No obstante, una directiva `using` sólo introduce de manera potencial los nombres en el espacio de nombres.

El punto 1 es bastante obvio. El punto 2 tiene varias sutilezas. Por ejemplo, suponga que los espacios de nombres `ns1` y `ns2` proveen definiciones para `mi_funcion`, pero no tienen otros conflictos de nombres. Entonces lo siguiente no generará problemas:

```
using namespace ns1;
using namespace ns2;
```

siempre y cuando (dentro del alcance de estas directivas) el nombre conflictivo `mi_funcion` nunca se utilice en nuestro código.

Por otro lado, lo siguiente es ilegal aún si nunca se utiliza la función `mi_funcion`:

```
using ns1::mi_funcion;
using ns2::mi_funcion;
```

Algunas veces este punto sutil puede ser importante, pero no afecta a la mayoría del código de rutina.

Ejercicios de AUTOEVALUACIÓN

10. Escriba la declaración para una función *void* llamada `wow`. Esta función tiene dos parámetros, el primero de tipo `velocidad` como se define en el espacio de nombres `autopista` y el segundo de tipo `velocidad` como se define en el espacio de nombres `indy500`.

11. Considere las siguientes declaraciones de funciones de la definición de la clase `Dinero` en el cuadro 8.4.

```
void entrada(istream& ins);
void salida(ostream& outs) const;
```

Reescriba estas declaraciones de funciones de manera que no necesite ir antes de ellas la siguiente instrucción:

```
using namespace std;
```

(No necesita regresar a ver el cuadro 8.4 para hacer esto.)

Espacios de nombres sin nombre

En nuestra definición de la clase `TiempoDigital` de los cuadros 9.1 y 9.2 utilizamos tres funciones de ayuda: `digito_a_int`, `lee_hora` y `lee_minuto`. Estas funciones de ayuda son parte de la implementación para la clase ADT `TiempoDigital`, por lo que colocamos sus definiciones en el archivo de implementación (cuadro 9.2). No obstante, esto en realidad no oculta estas tres funciones. Nos gustaría que estas funciones fueran locales para el archivo de implementación de la clase `TiempoDigital`. Sin embargo, así como están no son locales para el archivo de implementación de la clase `TiempoDigital`. De hecho, no podemos definir otra función con el nombre `digito_a_int` (o `lee_hora`, o `lee_minuto`) en un programa de aplicación que utilice la clase `TiempoDigital`. Esto viola el principio de ocultamiento de información. Para ocultar verdaderamente estas funciones de ayuda y hacerlas locales para el archivo de implementación de `TiempoDigital`, necesitamos colocarlas en un espacio de nombres especial, conocido como el espacio de nombres sin nombre.

unidad de compilación

espacio de nombres sin nombre

Una **unidad de compilación** es un archivo (tal como un archivo de implementación) junto con todos los archivos que se incluyen (mediante `#include`) en el archivo, tal como el archivo de encabezado de interfaz para la clase. Cada unidad de compilación tiene un **espacio de nombres sin nombre**. Una agrupación de espacio de nombres para el espacio de nombres sin nombre se escribe de la misma forma que para cualquier otro espacio de nombres, sólo que no se proporciona un nombre, como en el siguiente ejemplo:

```
namespace
{
    void ejemplo_funcion( )
       .
       .
       .
}  //espacio de nombres sin nombre
```

Todos los nombres definidos en el espacio de nombres sin nombre son locales para la unidad de compilación, por lo cual se pueden reutilizar para alguna otra tarea fuera de la unidad de compilación. Por ejemplo, las cuadros 9.6 y 9.7 muestran una versión modificada

(nuestra versión final) de los archivos de interfaz y de implementación para la clase `TiempoDigital`. Observe que todas las funciones de ayuda (`lee_hora`, `lee_minuto` y `digito_a_int`) se encuentran en el espacio de nombres sin nombre y, por lo tanto, son locales para la unidad de compilación. Como se muestra en el cuadro 9.8, los nombres en el espacio de nombres sin nombre pueden reutilizarse para algo más fuera de la unidad de compilación. En el cuadro 9.8, el nombre de función `lee_hora` se reutiliza para otra función distinta en el programa de aplicación.

Si analiza de nuevo el archivo de implementación en el cuadro 9.7, podrá ver que las funciones de ayuda `digito_a_int`, `lee_hora` y `lee_minuto` se utilizan fuera del espacio de nombres sin nombre, sin necesidad de usar un calificador de espacio de nombres. Cualquier nombre definido en el espacio de nombres sin nombre puede utilizarse en cualquier parte de la unidad de compilación, sin necesidad de calificarlo. (Desde luego que esto necesita ser así, ya que el espacio de nombres sin nombre no tiene un nombre con el que se pueda usar para calificar sus nombres.)

Es interesante observar cómo interactúan los espacios de nombres sin nombre con la regla de C++ que establece que no se pueden tener dos definiciones de un nombre (en el mismo espacio de nombres). Hay un espacio de nombres sin nombre en cada unidad de compilación. Es muy posible que se traslapen las unidades de compilación. Por ejemplo, el

CUADRO 9.6 *Colocación de una clase en un espacio de nombres: Archivo de encabezado*

```
//Archivo de encabezado tiempod.h: Esta es la interfaz para la clase TiempoDigital.
//Los valores de este tipo son horas del día. Los valores se introducen y se muestran
//en notación de 24 horas, como 9:30 para las 9:30 AM y 14:45 para las 2:45 PM.

#ifndef TIEMPOD_H
#define TIEMPOD_H

#include <iostream>
using namespace std;

namespace tiempodsavitch
{

    class TiempoDigital
    {

    <La definición de la clase TiempoDigital es la misma que en el cuadro 9.1.>

    };
}//fin de tiempodsavitch

#endif //TIEMPOD_H
```

Un agrupamiento para el espacio de nombres `tiempodsavitch`. *Otro agrupamiento para el espacio de nombres* `tiempodsavitch` *se encuentra en el archivo de implementación* `tiempod.cpp`.

CUADRO 9.7 *Colocación de una clase en un espacio de nombres: Archivo de implementación (parte 1 de 2)*

```
//Archivo de implementación dtime.cpp (Tal vez su sistema requiera un sufijo
//distinto a .cpp): Esta es la IMPLEMENTACIÓN del ADT TiempoDigital.
//La interfaz para la clase TiempoDigital está en el archivo de encabezado tiempod.h.
#include <iostream>
#include <cctype>
#include <cstdlib>
#include "tiempod.h"
using namespace std;

namespace
{
    //Estas declaraciones de funciones son para usarse en la definición
    //del operador de entrada >> sobrecargado:

    void lee_hora(istream& ins, int& la_hora);
    //Precondición: La siguiente entrada en el flujo ins es una hora en notación,
    //como 9:45 o 14:45.
    //Postcondición: la_hora se ha ajustado a la parte del tiempo
    //correspondiente a la hora.
    //Se ha descartado el signo de dos puntos y la siguiente entrada a leer es el minuto.

    void lee_minuto(istream& ins, int& el_minuto);
    //Lee el minuto del flujo ins después de que lee_hora ha leído la hora.

    int digito_a_int(char c);
    //Precondición: c es uno de los dígitos del '0' al '9'.
    //Devuelve el entero para el dígito; por ejemplo, digito_a_int('3') devuelve un 3.
}//espacio de nombres sin nombre


namespace tiempodsavitch
{
    bool operator ==(const TiempoDigital& tiempo1, const TiempoDigital& tiempo2)
   <El resto de la definición de == es igual que en el cuadro 9.2>

    TiempoDigital::TiempoDigital( ) : hora(0), minuto(0)
 <El resto de la definición de este constructor es igual que en el cuadro 9.2>


    //Usa iostream y cstdlib:
    TiempoDigital::TiempoDigital(int la_hora, int el_minuto)
 <El resto de la definición de este constructor es igual que en el cuadro 9.2.>
```

Un agrupamiento para el espacio de nombres sin nombre.

Un agrupamiento para el espacio de nombres `tiempodsavitch`. *Otro agrupamiento está en el archivo* `tiempod.h`.

CUADRO 9.7 *Colocación de una clase en un espacio de nombres: Archivo de implementación (parte 2 de 2)*

```
 void TiempoDigital::avanza(int minutos_agregados)
<El resto de la definición de esta función avanza es igual que en el cuadro 9.2.>

 void TiempoDigital::avanza(int horas_agregadas, int minutos_agregados)
<El resto de la definición de esta función avanza es igual que en el cuadro 9.2.>

 ostream& operator <<(ostream& outs, const TiempoDigital& el_objeto)
<El resto de la definición de << es igual que en el cuadro 9.2.>

 //Usa iostream y las funciones en el espacio de nombres sin nombre:
 istream& operator >>(istream& ins, TiempoDigital& el_objeto)
 {
     lee_hora(ins, el_objeto.hora);
     lee_minuto(ins, el_objeto.minuto);
     return ins;
 }
} //tiempodsavitch

namespace
{
    int digito_a_int(char c)
    <El resto de la definición de digito_a_int es igual que en el cuadro 9.2.>

    //Usa iostream, cctype y cstdlib:
    void lee_minuto(istream& ins, int& el_minuto)
    <El resto de la definición de lee_minuto es igual que en el cuadro 9.2.>


    //Usa iostream, cctype, y cstdlib:
    void lee_hora(istream& ins, int& la_hora)
    <El resto de la definición de lee_hora es igual que en el cuadro 9.2.>

}//espacio de nombres sin nombre
```

Las funciones definidas en el espacio de nombres sin nombre son locales para esta unidad de compilación (este archivo y los archivos incluidos). Pueden usarse en cualquier parte de este archivo, pero no tienen significado fuera de esta unidad de compilación.

Otro agrupamiento para el espacio de nombres sin nombre.

CUADRO 9.8 *Colocación de una clase en un espacio de nombres: Programa de aplicación (parte 1 de 2)*

```
//Este es el archivo de aplicación: tiempodemo.cpp. Este programa
//demuestra cómo ocultar las funciones de ayuda en un espacio de nombres sin nombre.

#include <iostream>
#include "tiempod.h"

void lee_hora(int& la_hora);

int main( )
{
     using namespace std;

     using namespace tiempodsavitch;

     int la_hora;
     lee_hora(la_hora);

     TiempoDigital reloj(la_hora, 0), reloj_ant;

     reloj_ant = reloj;
     reloj.avanza(15);
     if (reloj == reloj_ant)
          cout << "Algo esta mal.";
     cout << "Usted escribio " << reloj_ant << endl;
     cout << "15 minutos despues la hora sera "
          << reloj << endl;

     reloj.avanza(2, 15);
     cout << "2 horas y 15 minutos despues de eso\n"
          << "la hora sera "
          << reloj << endl;

     return 0;
}
```

Si coloca aquí las directivas using entonces el comportamiento del programa será el mismo.

Esta función `lee_hora` *es distinta a la del archivo de implementación* `tiempod.cpp` *(el cual se muestra en el cuadro 9.7).*

CUADRO 9.8 *Colocación de una clase en un espacio de nombres: Programa de aplicación (parte 2 de 2)*

```
void lee_hora(int& la_hora)
{
    using namespace std;

    cout << "Vamos jugar un juego de tiempo.\n"
         << "Pretendamos que acaba de cambiar la hora.\n"
         << "Puede escribir la media noche ya sea como 0 o 24,\n"
         << "pero yo siempre la escribire como 0.\n"
         << "Escriba la hora como un numero (0 a 24): ";
    cin >> la_hora;
    if (la_hora == 24)
        la_hora = 0;
}
```

Diálogo de ejemplo

```
Vamos jugar un juego de tiempo.
Pretendamos que acaba de cambiar la hora.
Puede escribir la media noche ya sea como 0 o 24,
pero yo siempre la escribire como 0.
Escriba la hora como un numero (0 a 24): 11
Usted escribio 11:00
15 minutos despues la hora sera 11:15
2 horas y 15 minutos despues de eso
la hora sera 13:30
```

archivo de implementación para una clase y un programa de aplicación por lo general incluyen el archivo de encabezado (archivo de interfaz) para la clase. Por ende, el archivo de encabezado está en dos unidades de compilación y por ello participa en dos espacios de nombres sin nombre. Aunque suene peligroso, esto por lo común no produce problemas siempre y cuando el espacio de nombres de cada unidad de compilación tenga sentido, si se le considera en forma individual. Por ejemplo, si se define un espacio de nombres sin nombre en el archivo de encabezado, no puede definirse de nuevo en el espacio de nombres sin nombre en el archivo de implementación o en el archivo de aplicación. De esta forma, se evita un conflicto de nombres.

TIP DE PROGRAMACIÓN

Elección de un nombre para un espacio de nombres

Es conveniente que incluya su apellido o cualquier otra cadena única en los nombres de sus espacios de nombres, para reducir la probabilidad de que alguien más utilice el mismo nombre de espacio de nombres que usted. Cuando hay varios programadores que escriben código para el mismo proyecto, es importante que los espacios de nombres que deban ser distintos en realidad tengan distintos nombres. De no ser así, es muy fácil llegar a tener varias definiciones de los mismos nombres en el mismo alcance. Esta es la razón por la cual incluimos el nombre `savitch` en el espacio de nombres `tiempodsavitch` de el cuadro 9.7.

Espacio de nombres sin nombre

Puede utilizar el **espacio de nombres sin nombre** para hacer que una definición sea local para una unidad de compilación (esto es, para un archivo y sus archivos incluidos). Cada unidad de compilación tiene un espacio de nombres sin nombre. Todos los identificadores definidos en el espacio de nombres sin nombre son locales para la unidad de compilación. Para colocar una definición en el espacio de nombres sin nombre, hay que colocarla en un agrupamiento de espacio de nombres sin ningún nombre para el espacio de nombres, como se muestra a continuación:

```
namespace
{
    Definicion_1
    Definicion_2
        .
        .
        .
    Ultima_definicion
}
```

Puede usar cualquier nombre en el espacio de nombres sin nombre en cualquier lugar dentro de la unidad de compilación, sin necesidad de calificarlo. En los cuadros 9.6, 9.7 y 9.8 podrá ver un ejemplo completo.

RIESGO *Confundir el espacio de nombres global y el espacio de nombres sin nombre*

No confunda el espacio de nombres global con el espacio de nombres sin nombre. Si no coloca la definición de un nombre en un agrupamiento de espacio de nombres, entonces se encuentra en el espacio de nombres global. Para colocar la definición de un nombre en el espacio de nombres sin nombre, debe colocarla en un agrupamiento de espacio de nombres que empiece de la siguiente manera, sin un nombre:

```
namespace
{
```

Se puede acceder al nombre en el espacio de nombres global y al nombre en el espacio de nombres sin nombre sin necesidad de un calificador. No obstante, los nombres en el espacio de nombres global tienen alcance global (en todos los archivos del programa), mientras que los nombres en un espacio de nombres sin nombre son locales para una unidad de compilación.

Esta confusión entre el espacio de nombres global y el espacio de nombres sin nombre no surge mucho al escribir código, ya que hay una tendencia en pensar que los nombres del espacio de nombres global están "en ningún espacio de nombres", aún y cuando esto no sea técnicamente correcto. No obstante, la confusión puede surgir con facilidad cuando se analiza el código.

Ejercicios de AUTOEVALUACIÓN

12. ¿Se comportaría el programa de el cuadro 9.8 de manera distinta si se sustituye la directiva `using`

```
using namespace tiempodsavitch;
```

con la siguiente declaración `using`?

```
using tiempodsavitch::TiempoDigital;
```

13. ¿Cuál es la salida que produce el siguiente programa?

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

namespace sally
{
    void mensaje( );
}
namespace
{
    void mensaje( );
}

int main( )
{
   {
       mensaje( );

       using sally::mensaje;
       mensaje( );
   }

   mensaje( );
   return 0;
}

namespace sally
{
    void mensaje( )
    {
```

```
        cout << "Saludos desde Sally.\n";
    }
}

namespace
{
    void mensaje( );
    {
       cout << "Saludos desde sin nombre.\n";
    }
}
```

14. En el cuadro 9.7 hay dos agrupamientos para el espacio de nombres sin nombre: uno para las declaraciones de las funciones de ayuda y otro para las definiciones de las funciones de ayuda. ¿Podemos eliminar el agrupamiento para las declaraciones de las funciones de ayuda? De ser así, ¿cómo podemos hacerlo?

Resumen del capítulo

- En C++, los tipos de datos abstractos (ADT) se implementan como clases con todas las variables miembro privadas, y las operaciones se implementan como funciones miembro y no miembro públicas y operadores sobrecargados.
- Se puede definir un ADT como una clase y se pueden colocar la definición de la clase y la implementación de sus funciones miembro en archivos separados. Así, se puede compilar la clase ADT de manera separada de cualquier programa que la utilice y se puede utilizar esta misma clase ADT en cualquier número de programas distintos.
- Un espacio de nombres es una colección de definiciones de nombres, como las definiciones de clases y las declaraciones de variables.
- Existen tres formas de utilizar un nombre desde un espacio de nombres: hacer que todos los nombres en el espacio de nombres estén disponibles mediante una directiva *using*, hacer el nombre individual esté disponible mediante una declaración *using* para el nombre solo, o mediante la calificación del nombre con el nombre del espacio de nombres y el operador de resolución de alcance.
- Se puede colocar una definición en un espacio de nombres si se coloca la definición en un agrupamiento de espacio de nombres para ese espacio de nombres.
- El espacio de nombres sin nombre puede usarse para hacer que la definición de un nombre sea local para una unidad de compilación.

Respuestas a los ejercicios de autoevaluación

1. Las partes (a), (b) y (c) van en el archivo de interfaz; las partes (d) a (h) van en el archivo de implementación. (Todas las definiciones de las operaciones de un ADT de cualquier tipo van en el archivo de implementación.) La parte (i) (es decir, la parte `main` de su programa) va en el archivo de aplicación.
2. El nombre del archivo de interfaz termina en `.h`.
3. Sólo necesita compilarse el archivo de implementación. El archivo de interfaz no necesita compilarse.

4. Sólo necesita volver a compilarse el archivo de implementación. Sin embargo, necesita volver a enlazar los archivos.

5. Necesita eliminar las variables miembro privadas `hora` y `minuto` del archivo de interfaz que se muestra en el cuadro 9.1, para sustituirlas con la variable miembro `minutos` (con `s`). No necesita hacer ningún otro cambio en el archivo de interfaz. En el archivo de implementación necesita modificar las definiciones de todos los constructores y demás funciones miembro, así como las definiciones de los operadores sobrecargados para que funcionen para esta nueva forma de registrar el tiempo. (En este caso no necesita modificar las funciones de ayuda `lee_hora`, `lee_minuto` o `digito_a_int`, pero tal vez esto no se aplique para alguna otra clase o incluso para alguna otra reimplementación de esta clase.) Por ejemplo, la definición del operador sobrecargado >> podría modificarse de la siguiente forma:

```
istream& operator >>(istream& ins, TiempoDigital& el_objeto)
{
     int hora_entrada, minuto_entrada;
     lee_hora(ins, hora_entrada);
     lee_minuto(ins, minuto_entrada);
     el_objeto.minutos = minuto_entrada + 60*hora_entrada;
     return ins;
}
```

   No necesita modificar ninguno de los archivos de aplicación para los programas que utilicen la clase. No obstante y como se ha modificado el archivo de interfaz (así como el archivo de implementación), tendrá que recompilar todos los archivos de aplicación y junto con el archivo de implementación.

6. La respuesta corta es que un ADT es sólo una clase que se define en base a las buenas prácticas de programación de separar la interfaz de la implementación. Además, cuando describimos una clase como un ADT, consideramos que las operaciones básicas de los elementos que no son miembros (como los operadores sobrecargados) son parte del ADT, aún y cuando no forman (en sentido técnico) parte de la clase de C++.

7. No. Si sustituye `gran_saludo` con `saluda`, entonces tendrá una definición para el nombre `saluda` en el espacio de nombres global. Hay partes del programa en donde están disponibles al mismo tiempo todas las definiciones de nombres en el espacio de nombres `savitch1` y todas las definiciones de nombres en el espacio de nombres global. En esas partes del programa habría dos definiciones distintas para

```
void greeting( );
```

8. Sí, la definición adicional no produciría problemas. Esto se debe a que siempre se permite la sobrecarga. Por ejemplo, cuando están disponibles los espacios de nombres `savitch1` y el global, el nombre de la función `saluda` estaría sobrecargado. El problema en el ejercicio de autoevaluación 7 era que algunas veces habría dos definiciones del nombre de la función `saluda` con las mismas listas de parámetros.

9. Sí, un espacio de nombres puede tener cualquier número de agrupamientos. Por ejemplo, a continuación se muestran dos agrupamientos para el espacio de nombres `savitch1` que aparecen en el cuadro 9.5:

```
namespace savitch1
{
    void saluda( );
}
namespace savitch1
{
    void saluda( );
     {
```

```
            cout << "Saludos desde el espacio de nombres savitch1.\n";
        }
    }
```

10. `void wow(autopista::velocidad v1, indy500:: velocidad v2);`

11. `void entrada(std::istream& ins);`
    `void salida(std::ostream& outs) const;`

12. El programa se comportaría exactamente de la misma forma.

13. `Saludos desde sin nombre.`
    `Saludos desde Sally.`
    `Saludos desde sin nombre.`

14. Sí, puede eliminar el agrupamiento para las declaraciones de las funciones de ayuda, siempre y cuando el agrupamiento con las definiciones de las funciones de ayuda ocurra antes de utilizar las funciones de ayuda. Por ejemplo, podría eliminar el espacio de nombres con las declaraciones de las funciones de ayuda y mover el agrupamiento con las definiciones de las funciones de ayuda justo antes del agrupamiento de espacio de nombres para el espacio de nombres `tiempodsavitch`.

Proyectos de programación

1. Agregue la siguiente función miembro a la clase ADT `TiempoDigital` que definimos en los cuadros 9.1 y 9.2:

   ```
   void TiempoDigital::intervalo_desde(const TiempoDigital& un_tiempo_anterior,
                   int& horas_en_intervalo, int& minutos_en_intervalo) const
   ```

   Esta función calcula el intervalo de tiempo transcurrido entre dos valores de tipo `TiempoDigital`. Uno de los valores de tipo `TiempoDigital` es el objeto que llama a la función miembro `intervalo_desde`, y el otro valor de tipo `TiempoDigital` se proporciona como el primer argumento. Por ejemplo, considere el siguiente código:

   ```
   TiempoDigital actual(5, 45), anterior(2, 30);
   int horas, minutos;
   actual.intervalo_desde(anterior, horas, minutos);
   cout << "El intervalo de tiempo entre " << anterior
        << " y " << actual << endl
        << "es de"<< horas << "horas y "
        << minutos << " minutos.\n";
   ```

   En un programa que utilice esta versión modificada del ADT `TiempoDigital`, este código debe producir la siguiente salida:

   ```
   El intervalo de tiempo entre 2:30 y 5:45
   es de 3 horas y 15 minutos.
   ```

   Debe permitir que el tiempo dado por el primer argumento sea posterior en el día al tiempo del objeto que hace la llamada. En este caso, se asume que el tiempo que se da como primer argumento es del día anterior. También debe escribir un programa para probar esta clase ADT modificada.

2. Realice el ejercicio de autoevaluación 5 con todos los detalles. Escriba la clase ADT completa, incluyendo los archivos de interfaz y de implementación. Además, escriba un programa para probar su clase ADT.

3. Realice de nuevo el proyecto de programación 1 del capítulo 8, pero esta vez defina la clase ADT `Dinero` en archivos separados para la interfaz y la implementación, de manera que la implementación pueda compilarse en forma separada de cualquier programa de aplicación.

4. Realice de nuevo el proyecto de programación del capítulo 8, pero esta vez defina la clase ADT `Pares` en archivos separados para la interfaz y la implementación, de manera que la implementación pueda compilarse en forma separada de cualquier programa de aplicación.

5. Realice de nuevo (o por primera vez) el proyecto de programación 4 del capítulo 8. Defina su clase ADT en archivos separados para que pueda compilarse por separado.

6. Realice de nuevo (o por primera vez) el proyecto de programación 5 del capítulo 8. Defina su clase ADT en archivos separados, para que pueda compilarse por separado.

10 Arreglos

10 Arreglos

Es un error garrafal proponer teorías
antes de tener datos.

SIR ARTHUR CONAN DOYLE, *Escándalo en Bohemia (Sherlock Holmes)*

Introducción

Usamos un *arreglo* para procesar una colección de datos del mismo tipo, como una lista de temperaturas o una lista de nombres. En este capítulo presentaremos los fundamentos de la definición y uso de arreglos en C++ y muchas de las técnicas básicas que se emplean para diseñar algoritmos y programas que usan arreglos.

Prerrequisitos

Las secciones 10.1, 10.2, 10.3 y 10.5 utilizan material de los capítulos del 2 al 5 y del 7. No utilizan material de los capítulo 6, 8 o 9. La sección 10.4 utiliza material de los capítulos del 2 al 9. La sección 10.5 no depende de la sección 10.4.

10.1 Introducción a los arreglos

Supongamos que queremos escribir un programa que lea los puntos obtenidos en cinco exámenes y manipule esos puntajes de alguna forma. Por ejemplo, el programa podría calcular el puntaje más alto y luego mostrar cuánto le falta a cada uno de los puntajes para alcanzar al más alto. No sabremos cuál es el puntaje más alto hasta no leerlos todos. Por tanto, deberemos mantener en la memoria los cinco puntajes para que, una vez determinado el puntaje más alto, cada puntaje se pueda comparar con él.

Para retener los cinco puntajes requeriremos algo equivalente a cinco variables de tipo `int`. Podríamos usar cinco variables individuales de tipo `int`, pero no es fácil seguir la pista a cinco variables, y probablemente más adelante quisieramos cambiar nuestro programa para manejar 100 puntajes; sin lugar a dudas 100 variables sería poco práctico. Un arreglo es la solución perfecta. Un **arreglo** se parece mucho a una lista de variables, con un mecanismo uniforme de autonombramiento, que se puede declarar en una sola línea de código simple. Por ejemplo, los nombres de las cinco variables individuales que necesitamos podrían ser `puntos[0]`, `puntos[1]`, `puntos[2]`, `puntos[3]` y `puntos[4]`. La parte que no cambia —en este caso `puntos`— es el nombre del arreglo. La parte que puede cambiar es el entero encerrado en corchetes `[]`.

arreglo

Declaración y referencias de arreglos

En C++, un arreglo que consiste en cinco variables de tipo `int` se puede declarar así:

```
int puntos[5];
```

Esta declaración equivale a indicar que las siguientes cinco variables son todas de tipo *`int`*:

```
puntos[0], puntos[1], puntos[2], puntos[3], puntos[4]
```

variable indizada
variable con subíndice
elemento
índice o subíndice
tamaño declarado

Estas variables individuales que juntas constituyen el arreglo reciben diferentes nombres. Nosotros las llamaremos **variables indizadas**, aunque también se conocen como **variables con subíndice** o **elementos** del arreglo. El número encerrado en corchetes se llama **índice** o **subíndice**. En C++, *los índices comienzan con* `0`*, no con* `1` *ni con ningún otro número que no sea* `0`. El número de variables indizadas de un arreglo es el **tamaño declarado** del arreglo, o simplemente el **tamaño** del arreglo. Cuando se declara un arreglo, su tamaño se encierra en corchetes después del nombre del arreglo. Las variables indizadas se numeran (también usando corchetes) desde `0` hasta el entero que es *uno menos que el tamaño del arreglo*.

En nuestro ejemplo, las variables indizadas eran de tipo *`int`*, pero un arreglo puede tener variables indizadas de cualquier tipo. Por ejemplo, para declarar un arreglo de variables indizadas de tipo *`double`*, simplemente usamos el nombre de tipo *`double`* en lugar de *`int`* en la declaración del arreglo. Eso sí, todas las variables indizadas de un arreglo deben ser del mismo tipo. Este tipo es el **tipo base** del arreglo. Así pues, en nuestro ejemplo del arreglo `puntos`, el tipo base es *`int`*.

tipo base

Podemos declarar arreglos y variables ordinarias juntos. Por ejemplo, podríamos haber usado lo siguiente, que declara las dos variables `int siguiente` y `max` además del arreglo `puntos`:

```
int siguiente, puntos[5], max;
```

Podemos usar una variable indizada como `puntos[3]` en cualquier lugar que se pueda usar una variable ordinaria de tipo *`int`*.

No confunda las dos formas de usar los corchetes `[]` con un nombre de arreglo. Cuando se usan en una declaración, como

```
int puntos[5];
```

el número encerrado en los corchetes especifica cuántas variables indizadas tiene el arreglo. Cuando se usan en cualquier otro lugar, el número encerrado en los corchetes indica de cuál variable indizada se trata. Por ejemplo, `puntos[0]` a `puntos[4]` son varibles indizadas.

El índice entre corchetes no tiene que ser una constante entera. Podemos usar cualquier expresión en los corchetes en tanto la evaluación de la expresión dé uno de los enteros de `0` hasta uno menos que el tamaño del arreglo. Por ejemplo, lo que sigue asigna el valor `99` a `puntos[3]`:

```
int n = 2;
puntos[n + 1] = 99;
```

Aunque se vean diferentes, `puntos[n + 1]` y `puntos[3]` son la misma variable indizada en el código anterior. La razón es que, en ese código, la evaluación de `n + 1` da `3`.

La identidad de una variable indizada, como `puntos[i]`, la determina el valor de su índice, que en este caso es `i`. Así pues, podemos escribir programas que digan cosas como "hacer esto y lo otro con la `i`-ésima variable indizada", donde el programa calcula el valor de `i`. Por ejemplo, el programa del cuadro 10.1 lee puntajes y los procesa en la forma que describimos al principio del capítulo.

TIP DE PROGRAMACIÓN

Emplee ciclos `for` con arreglos

El segundo ciclo *`for`* del cuadro 10.1 ilustra una forma común de procesar un arreglo usando un ciclo *`for`*:

```
for (i = 0; i < 5; i++)
    cout << puntos[i] << " es "
         << (max - puntos[i]) << endl;
```

La instrucción *`for`* es ideal para manipular arreglos.

RIESGO *Los índices de arreglos siempre comienzan con cero*

Los índices de un arreglo siempre comienzan con `0` y terminan con un entero que es uno menor que el tamaño del arreglo.

CUADRO 10.1 *Programa que usa un arreglo*

```
//Lee 5 puntajes e indica la diferencia entre
//cada puntaje y el puntaje más alto.
#include <iostream>

int main()
{
    using namespace std;
    int i, puntos[5], max;

    cout << "Escriba 5 puntajes:\n";
    cin >> puntos[0];
    max = puntos[0];
    for (i = 1; i < 5; i++)
    {
        cin >> puntos[i];
        if (puntos[i] > max)
            max = puntos[i];
        //max es el mayor de los valores puntos[0], ..., puntos[i].
    }
    cout << "El puntaje mas alto es " << max << endl
         << "Los puntajes y sus\n"
         << "diferencias respecto al mas alto son:\n";
    for (i = 0; i < 5; i++)
        cout << puntos[i] << " es "
                << (max - puntos[i]) << endl;

    return 0;
}
```

Diálogo de ejemplo

```
Escriba 5 puntajes:
5 9 2 10 6
El puntaje mas alto es 10
Los puntajes y sus
diferencias respecto al mas alto son:
5 es 5 menor
9 es 1 menor
2 es 8 menor
10 es 0 menor
6 es 4 menor
```

TIP DE PROGRAMACIÓN

Use una constante definida para el tamaño de un arreglo

Examine nuevamente el programa del cuadro 10.1. Sólo funciona con clases que tienen exactamente cinco estudiantes. Pocas clases tienen exactamente cinco estudiantes. Una manera de hacer más versátil un programa es usar una constante definida para el tamaño de cada arreglo. Por ejemplo, el programa del cuadro 10.1 podría reescribirse para utilizar la siguiente constante definida:

```
const int NUM_ESTUDIANTES = 5;
```

La línea con la declaración del arreglo sería

```
int i, puntos[NUM_ESTUDIANTES], max;
```

Por supuesto todos los lugares que tengan `5` para el tamaño del arreglo deberían cambiarse para tener `NUM_ESTUDIANTES` en lugar de `5`. Si se hace este cambio al programa (o mejor aún, si el programa se hubiese escrito de esta manera desde el principio), entonces el programa se podría reescribir para trabajar con cualquier número de estudiantes cambiando simplemente la línea que define a la constante `NUM_ESTUDIANTES`.

Observe que para el tamaño del arreglo *no podemos* utilizar una variable de la siguiente manera:

```
cout << "Ingrese el numero de estudiantes:\n";
cin >> numero;
int puntos[numero]; //NO VALIDO EN VARIOS COMPILADORES
```

Algunos compiladores, no todos, le permitirán especificar el tamaño de un arreglo con una variable de esta manera. Sin embargo, para garantizar la portabilidad lo mejor es no hacerlo, aun cuando su compilador se lo permita. (En el capítulo 12 analizaremos otro tipo de arreglo en el que se puede especificar el tamaño cuando el programa está ejecutándose.)

Arreglos en memoria

Antes de explicar cómo se representan los arreglos en la memoria de una computadora, repasemos la forma en que las variables simples, como las de tipo `int` o `double`, se representan en la memoria. Recuerde que la memoria de una computadora consiste en una lista de posiciones numeradas llamadas *bytes*[1]. El número de un byte se conoce como su **dirección**. Una variable simple se implementa como una porción de memoria que consiste de un número consecutivo de bytes. El número de bytes se determina por el tipo de la variable. Así pues, una variable en memoria se describe con dos elementos de información: una dirección en la memoria (que da la ubicación del primer byte dedicado a esa variable) y el tipo de la variable, que dice cuántos bytes de memoria requiere la variable. Cuando hablamos de la *dirección de una variable*, es esta dirección de la que estamos hablando. Cuando un programa almacena un valor en la variable, lo que realmente sucede es que el valor (codificado como ceros y unos) se coloca en esos bytes de memoria que se asignaron a esa variable. Así mismo, cuando damos una variable como argumento (de llamada por referencia) de una función, es la dirección de la variable lo que realmente se proporciona a la función que llama. Ahora repasaremos la forma en que los arreglos se almacenan en la memoria.

dirección

[1] Un byte está compuesto por ocho bits, pero el tamaño exacto de un byte no es importante en este momento.

Declaración de arreglos

Sintaxis:

Nombre_de_Tipo Nombre_de_Arreglo[*Tamaño_Declarado*];

Ejemplos:

```
int gran_arreglo[100];
double a[3];
double b[5];
char calif[10], una_calif;
```

Una declaración de arreglo, de la forma que se muestra aquí, define *Tamaño_Declarado* variables indizadas, es decir, las variables indizadas *Nombre_de_Arreglo*[0] hasta *Nombre_de_Arreglo*[*Tamaño_Declarado*−1]. Cada variable indizada es una variable del tipo *Nombre_de_Tipo*.

El arreglo a consiste en las variables indizadas a[0], a[1] y a[2], todas de tipo *double*. El arreglo b consiste en las variables indizadas b[0], b[1], b[2], b[3] y b[4], todas también de tipo *double*. Podemos combinar declaraciones de arreglos con la declaración de variables simples, como en el caso de la variable una_calif que se muestra arriba.

arreglos en memoria

Las variables indizadas de un arreglo se representan en la memoria igual que las variables ordinarias, pero en el caso de los arreglos hay algunos detalles adicionales. Las posiciones de las distintas variables indizadas de un arreglo siempre se colocan adyacentes en la memoria. Por ejemplo, consideremos lo siguiente:

```
int a[6];
```

Cuando declaramos este arreglo, la computadora reserva suficiente memoria para contener seis variables de tipo *int*. Además, siempre coloca estas variables una tras otra en la memoria. Luego, recuerda la dirección de la variable indizada a[0], pero no recuerda la dirección de ninguna otra variable indizada. Cuando el programa necesita la dirección de alguna otra variable indizada, la computadora calcula la dirección de esta otra variable a partir de la dirección de a[0]. Por ejemplo, si comenzamos en la dirección de a[0] y contamos suficiente memoria para tres variables de tipo *int*, estaremos en la dirección de a[3]. Para obtener la dirección de a[3], la computadora parte de la dirección de a[0] (que es un número). A continuación, la computadora suma a la dirección de a[0] el número de bytes necesario para contener tres variables de tipo *int*. El resultado es la dirección de a[3]. Esta implementación se muestra en forma de diagrama en el cuadro 10.2.

Muchas de las peculiaridades de los arreglos en C++ sólo pueden entenderse en términos de estos detalles relativos a la memoria. Por ejemplo, en la siguiente sección Riesgo usamos estos detalles para explicar qué sucede cuando un programa usa un índice de arreglo no válido.

RIESGO *Índice de arreglo fuera de intervalo*

El error de programación más común que se comete al usar arreglos es tratar de hacer referencia a un elemento inexistente del arreglo. Por ejemplo, considere la siguiente declaración de arreglo:

```
int a[6];
```

Al utilizar el arreglo `a`, toda expresión que se use como índice deberá dar uno de los enteros `0` a `5` al evaluarse. Por ejemplo, si el programa contiene la variable indizada `a[i]`, la evaluación de `i` debe dar uno de los seis enteros `0`, `1`, `2`, `3`, `4` o `5`. Si la evaluación de `i` da alguna otra cosa, será un error. Cuando la evaluación de una expresión índice da algún valor distinto de los permitidos por la declaración del arreglo, decimos que el índice está **fuera de intervalo** o simplemente que **no es válido**. En casi todos los sistemas, el resultado de un índice de arreglo no válido es que el programa hará algo mal, posiblemente desastroso, y lo hará sin advertirnos de ello.

índice de arreglo no válido

Por ejemplo, supongamos que nuestro sistema es típico, que declaramos el arreglo `a` como se mostró antes, y que nuestro programa contiene lo siguiente:

```
a[i] = 238;
```

Supongamos ahora que el valor de `i`, por alguna razón, es `7`. La computadora procede como si `a[7]` fuera una variable indizada válida, y calcula la dirección donde `a[7]` estaría (si existiera) y coloca el valor `238` en esa posición de memoria. Sin embargo, no existe ninguna variable indizada `a[7]`, y la memoria que recibe este `238` probablemente pertenece a alguna otra variable, tal vez una llamada `mas_cosas`. Así, hemos modificado sin querer el valor de `mas_cosas`. La situación se ilustra en el cuadro 10.2.

Los índices de arreglo fuera de intervalo suelen presentarse más comúnmente en la primera o la última iteración de un ciclo que procesa el arreglo. Asegúrese de que todos los ciclos que procesan arreglos inicien y terminen con índices de arreglo válidos.

Inicialización de arreglos

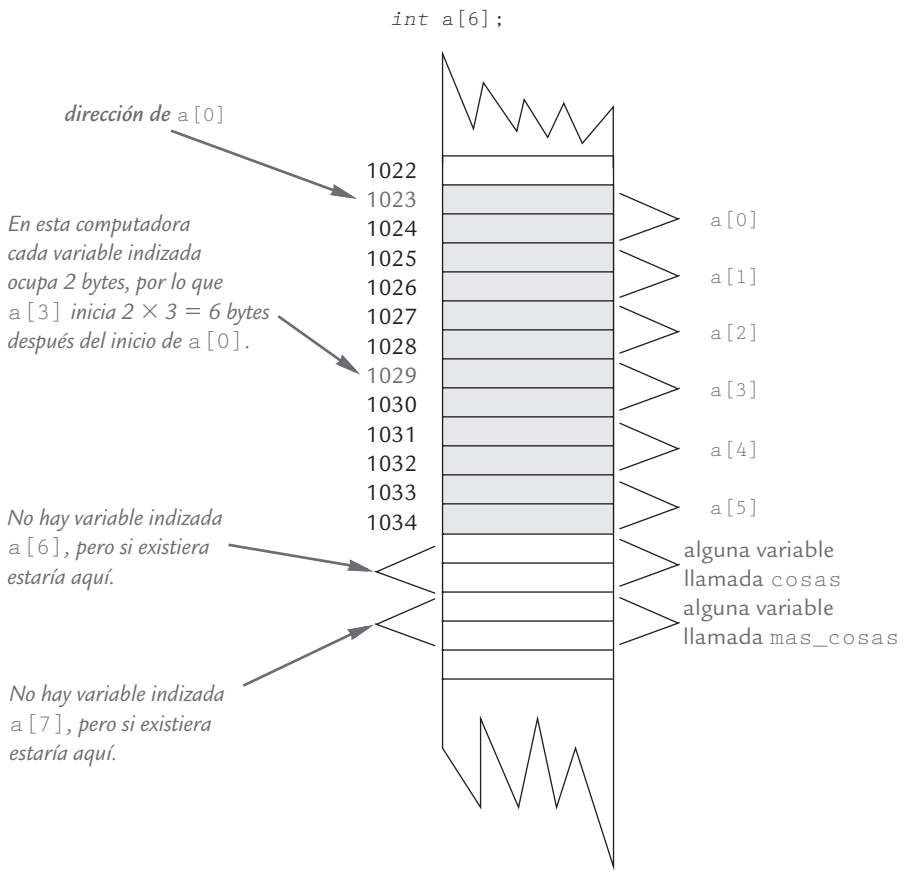
Podemos inicializar un arreglo al declararlo. Al hacerlo, los valores de las distintas variables indizadas se encierran en llaves y se separan con comas. Por ejemplo,

```
int chicos[3] = {2, 12, 1};
```

La declaración anterior equivale al siguiente código:

```
int chicos[3];
chicos[0] = 2;
```

CUADRO 10.2 *Un arreglo en la memoria*




```
chicos[1] = 12;
chicos[2] = 1;
```

Si la lista incluye menos valores que variables indizadas, los valores se usarán para inicializar las primeras variables indizadas, y el resto de las variables indizadas se inicializarán con un cero del tipo base del arreglo. En esta situación, las variables indizadas para las que no se proporcionan inicializadores se inicializan con 0. Sin embargo, los arreglos sin inicializadores y otras variables declaradas dentro de una definición de función o en la parte `main` del programa no se inicializan. Aunque las variables indizadas de arreglos (y otras variables) podrían inicializarse automáticamente con `0` en algunas situaciones, no podemos ni debemos contar con que así es.

Si inicializamos un arreglo cuando lo declaramos, podemos omitir su tamaño, y dicho arreglo quedará declarado automáticamente con el tamaño mínimo necesario para dar cabida a los valores de inicialización. Por ejemplo, la siguiente declaración

```
int b[] = {5, 12, 11};
```

equivale a

```
int b[3] = {5, 12, 11};
```

Ejercicios de AUTOEVALUACIÓN

1. Describa la diferencia en los significados de `int a[5];` y de `a[4]`. ¿Qué significan el `[5]` y el `[4]` en cada caso?

2. En la declaración del arreglo

   ```
   double puntos[5];
   ```

   identifique lo siguiente:

   a) El nombre del arreglo.
   b) El tipo base.
   c) El tamaño declarado del arreglo.
   d) El intervalo de valores que puede tener un índice que acceda a este arreglo.
   e) Una de las variables indizadas (o elementos) de este arreglo.

3. Identifique los errores, si los hay, de las siguientes declaraciones de arreglos.

   a) `int x[4] = {8, 7, 6, 4, 3};`
   b) `int x[] = {8, 7, 6, 4};`
   c) `const int TAM = 4;`
   d) `int x[TAM];`

4. ¿Qué salida produce el siguiente código?

   ```
   char simbolo[3] = {'a', 'b', 'c'};

   for (int indice = 0; indice < 3; indice++)
       cout << simbolo[indice];
   ```

5. ¿Qué salida produce el siguiente código?

   ```
   double a[3] = {1.1, 2.2, 3.3};

   cout << a[0] << " " << a[1] << " " << a[2] << endl;

   a[1] = a[2];
   cout << a[0] << " " << a[1] << " " << a[2] << endl;
   ```

6. ¿Qué salida produce el siguiente código?

```
int i, temp[10];
for (i = 0; i < 10; i++)
   temp[i] = 2*i;
for (i = 0; i < 10; i++)
   cout << temp[i] << " ";
cout << endl;
for (i = 0; i < 10; i = i + 2)
   cout << temp[i] << " ";
```

7. ¿Qué defecto tiene el siguiente fragmento de código?

```
int arreglo_muestra[10];
for (int indice = 1; indice <= 10; indice++)
     arreglo_muestra[indice] = 3*indice;
```

8. Suponga que esperamos que los elementos de un arreglo `a` estén ordenados de modo que

   `a[0] ≤ a[1] ≤ a[2]≤...`

   Sin embargo, para estar seguros queremos que el programa pruebe el arreglo y emita una advertencia si encuentra que algunos elementos no están en orden. Se supone que el siguiente código genera dicha advertencia, pero contiene un error; ¿en qué consiste?

```
double a[10];
    <Aquí sigue código para llenar el arreglo a>.
for (int indice = 0; indice < 10; indice++)
    if (a[indice] > a[indice + 1])
        cout << "Los elementos " << indice << " y "
             << (indice + 1) << " no estan en orden.";
```

9. Escriba código C++ que llene un arreglo con 20 valores de tipo `int` que se leen del teclado. No es necesario escribir todo un programa, sólo el código que hace esto, pero sí debe incluir las declaraciones del arreglo y de todas las variables.

10. Suponga que tiene la siguiente declaración de arreglo en un programa:

```
int tu_arreglo[7];
```

    Y suponga también que en su implementación de C++ las variables de tipo `int` ocupan dos bytes de memoria. Cuando ejecute su programa, ¿cuánta memoria consumirá este arreglo? Suponga que, al ejecutar su programa, el sistema asigna la dirección de memoria 1000 a la variable indizada `tu_arreglo[0]`. ¿Qué dirección tendrá la variable indizada `tu_arreglo[3]`?

10.2 Arreglos en funciones

Podemos usar como argumentos de funciones tanto variables indizadas como arreglos enteros. Primero trataremos el uso de variables indizadas como argumentos de funciones.

Variables indizadas como argumentos de funciones

Una variable indizada puede ser un argumento de una función igual que lo puede ser cualquier variable. Por ejemplo, suponga que un programa contiene las siguientes declaraciones:

```
int i, n, a[10];
```

Si `mi_funcion` recibe un argumento de tipo `int`, entonces lo siguiente es válido:

```
mi_funcion(n);
```

Puesto que una variable indizada del arreglo `a` también es una variable de tipo `int`, igual que `n`, lo siguiente también es válido:

```
mi_funcion(a[3]);
```

Hay un detalle sutil que aplica al uso de variables indizadas como argumentos. Por ejemplo, considere la siguiente llamada de función:

```
mi_funcion(a[i]);
```

Si el valor de `i` es `3`, entonces el argumento es `a[3]`. Por otra parte, si el valor de `i` es `0`, esta llamada equivale a la siguiente:

```
mi_funcion(a[0]);
```

La expresión indizada se evalúa para determinar exactamente cuál variable indizada se da como argumento.

El cuadro 10.3 contiene un ejemplo sencillo del uso de variables indizadas como argumentos de función. Este programa otorga cinco días de vacaciones adicionales a tres empleados de un negocio pequeño. El programa es muy sencillo, pero ilustra el uso de variables indizadas como argumentos de funciones. Observe la función `ajustar_dias`. Esta función tiene un parámetro formal llamado `dias_antes` que es de tipo `int`. Además, dicha función se invoca en la parte `main` del programa mediante el argumento `vacacion[numero]` para diversos valores de `numero`. Cabe señalar que el parámetro formal `dias_antes` no tiene nada de especial; es sólo un parámetro formal ordinario de tipo `int`, que es el tipo base del arreglo `vacacion`. En el cuadro 10.3 las variables indizadas son argumentos de llamada por valor. Los mismos comentarios aplican a los argumentos de llamada por referencia. Una variable indizada puede ser un argumento de llamada por valor o uno de llamada por referencia.

CUADRO 10.3 *Variable indizada como argumento*

```
//Ilustra el uso de una variable indizada como argumento.
//Suma 5 al número de días de vacación que puede tomar cada empleado.
#include <iostream>

const int NUMERO_DE_EMPLEADOS = 3;

int ajustar_dias(int dias_antes);
//Devuelve dias_antes más 5.

int main()
{
    using namespace std;
    int vacacion[NUMERO_DE_EMPLEADOS], numero;

    cout << "De los dias de vacaciones permitidos para los empleados"
         << " del 1 al " << NUMERO_DE_EMPLEADOS << ":\n";
    for (numero = 1; numero <= NUMERO_DE_EMPLEADOS; numero++)
        cin >> vacacion[numero-1];

    for (numero = 0; numero < NUMERO_DE_EMPLEADOS; numero++)
        vacacion[numero] = ajustar_dias(vacacion[numero]);
    cout << "Los nuevos numeros de dias de vacaciones son:\n";
    for (numero = 1; numero <= NUMERO_DE_EMPLEADOS; numero++)
        cout << "Empleado numero " << numero
             << " dias de vacaciones = " << vacacion[numero-1] << endl;
    return 0;
}
int ajustar_dias(int dias_antes)
{
    return (dias_antes + 5);
}
```

Diálogo de ejemplo

```
De los dias de vacaciones permitidos para los empleados del 1 al 3:
10 20 5
Los nuevos numeros de dias de vacaciones son:
Empleado numero 1 dias de vacaciones = 15
Empleado numero 2 dias de vacaciones = 25
Empleado numero 3 dias de vacaciones = 10
```

Ejercicios de AUTOEVALUACIÓN

11. Considere la siguiente definición de función:

```
void triplicador(int& n)
{
n = 3*n;
}
```

¿Cuáles de las siguientes son llamadas de función aceptables?

```
int a[3] = {4, 5, 6}, numero = 2;
triplicador(numero);
triplicador(a[2]);
triplicador(a[3]);
triplicador(a[numero]);
triplicador(a);
```

12. ¿Qué defecto tiene (si acaso lo hay) el siguiente código? La definición de `triplicador` se da en el ejercicio de autoevaluación 11.

```
int b[5] = {1, 2, 3, 4, 5};

for (int i = 1; i <= 5; i++)
    triplicador(b[i]);
```

Arreglos enteros como argumentos de funciones

Una función puede tener un parámetro formal para un arreglo entero de modo que cuando se invoque la función el argumento que se inserte en lugar de este parámetro formal sea un arreglo entero. Sin embargo, el parámetro formal de un arreglo entero no es ni de llamada por valor ni de llamada por referencia; es un nuevo tipo de parámetro formal llamado **parámetro de arreglo**. Comencemos con un ejemplo.

parámetro de arreglo

La función que se define en el cuadro 10.4 tiene un parámetro de arreglo `a`, que será sustituido por todo un arreglo cuando se invoque la función. Ésta también tiene un parámetro ordinario de llamada por valor (`tamanio`) que supuestamente es un valor entero igual al tamaño del arreglo. Esta función llena su argumento tipo arreglo (es decir, llena todas las variables indizadas del arreglo) con los valores que se introducen mediante el teclado, y luego despliega un mensaje en la pantalla que da el índice del último elemento del arreglo empleado.

El parámetro formal `int a[]` es un parámetro de arreglo. Los corchetes sin expresión índice adentro son lo que C++ usa para indicar un parámetro de arreglo. Un parámetro de arreglo no es exactamente un parámetro de llamada por referencia, pero en la práctica se comporta de forma muy parecida a uno. Sigamos este ejemplo detalladamente para ver cómo funciona un argumento arreglo en este caso. (Un **argumento tipo arreglo** o simplemente **argumento arreglo** es, desde luego, un arreglo que se inserta en lugar de un parámetro de arreglo, como `a[]`).

argumento tipo arreglo

Cuando se invoca la función `llenar`, debe tener dos argumentos: el primero da un arreglo de enteros y el segundo deberá dar el tamaño declarado del arreglo. Por ejemplo, la siguiente es una llamada de función aceptable:

```
int puntos[5], numero_de_puntajes = 5;
llenar(puntos, numero_de_puntajes);
```

cuándo utilizar `[]`

Esta llamada a `llenar` llenará el arreglo `puntos` con cinco enteros introducidos mediante el teclado. Observe que el parámetro formal `a[]` (que usamos en la declaración de la función y en el encabezado de la definición de la función) se da con corchetes pero sin expresión de índice. (Podemos insertar un número dentro de los corchetes de un parámetro de arreglo, pero el compilador simplemente hará caso omiso del número, por lo que no usaremos tales números en este libro.) En cambio, el argumento dado en la llamada de función (`puntos` en este ejemplo) se da sin corchetes ni expresión de índice.

¿Qué sucede con el argumento arreglo `puntos` en esta llamada de función? En términos muy informales, el argumento `puntos` se *inserta* en el cuerpo de la función en lugar del parámetro de arreglo formal `a`, y luego se ejecuta el cuerpo de la función. Así pues, la llamada de función

```
llenar(puntos, numero_de_puntajes);
```

CUADRO 10.4 *Función con un parámetro de arreglo*

Declaración de función

```
void llenar(int a[], int tamanio);
//Precondición: tamanio es el tamaño declarado del arreglo a.
//El usuario introducirá tamanio enteros con el teclado.
//Postcondición: El arreglo a se llenó con tamanio enteros
//introducidos con el teclado.
```

Definición de función

```
//Usa iostream:
void llenar(int a[], int tamanio)
{
using namespace std;
cout << "Teclee " << tamanio << " numeros:\n";
for (int i = 0; i < tamanio; i++)
    cin >> a[i];
tamanio--;
cout << "El ultimo indice de arreglo empleado fue " << tamanio << endl;
}
```

Equivale al código:

```
{
   using namespace std;
   tamanio = 5;
   cout << "Escriba " << tamanio << " numeros:\n";
   for (int i = 0; i < tamanio; i++)
        cin >> puntos[i];
   tamanio--;
   cout << "El ultimo indice de arreglo empleado fue " << tama-
   nio << endl;
}
```

5 es el valor de numero_de_puntajes

El parámetro formal `a` es diferente al tipo de parámetros que hemos visto hasta ahora. El parámetro formal `a` simplemente es un marcador de posición para el argumento `puntos`. Cuando se invoca la función `llenar` con `puntos` como argumento arreglo, la computadora actúa como si `a` hubiera sido reemplazado por el argumento correspondiente `puntos`. *Cuando usamos un arreglo como argumento en una llamada de función, cualquier acción que se efectúe con el parámetro de arreglo se efectúa con el argumento arreglo, así que la función puede modificar los valores de las variables indizadas del argumento arreglo*. Si se modifica el parámetro formal en el cuerpo de la función (por ejemplo, con una instrucción `cin`), el argumento arreglo será modificado.

Hasta aquí parecería que un parámetro de arreglo no es más que un parámetro de llamada por referencia para un arreglo. Eso no está lejos de la verdad, pero un parámetro de arreglo es un poco diferente. Para entender mejor la diferencia, repasemos algunos detalles de los arreglos.

arreglos en memoria

Recuerde que un arreglo se almacena en un trozo contiguo de la memoria. Por ejemplo, consideremos la siguiente declaración del arreglo `puntos`:

```
int puntos[5];
```

Cuando declaramos este arreglo, la computadora reserva suficiente memoria para contener cinco variables de tipo *int*, las cuales se guardan una tras otra en la memoria de la computadora. La computadora no recuerda las direcciones de cada una de estas cinco variables indizadas; sólo recuerda la dirección de la variable indizada `puntos[0]`. Por ejemplo, cuando nuestro programa necesita `puntos[3]`, la computadora calcula la dirección de `puntos[3]` a partir de la dirección de `puntos[0]`. La computadora sabe que `puntos[3]` se encuentra tres variables *int* después de `puntos[0]`. Así pues, para obtener la dirección de `puntos[3]`, la computadora toma la dirección de `puntos[0]` y le suma un número que representa la cantidad de memoria ocupada por tres variables *int*; el resultado es la dirección de `puntos[3]`.

argumento arreglo

Visto de esta forma, un arreglo tiene tres partes: la dirección (ubicación en memoria) de la primera variable indizada, el tipo base del arreglo (que determina cuánta memoria ocupa cada variable indizada) y el tamaño del arreglo (el número de variables indizadas). Cuando usamos un arreglo como argumento de una función, sólo damos a la función la primera de estas tres partes. Cuando insertamos un argumento arreglo en lugar de su parámetro formal correspondiente, lo único que se inserta es la dirección de la primera variable indizada del arreglo. El tipo base del argumento arreglo debe coincidir con el tipo del parámetro formal, por lo que la función también conoce el tipo base del arreglo. *Sin embargo, el argumento arreglo no le dice a la función qué tamaño tiene el arreglo*. Cuando se ejecuta el código del cuerpo de la función, la computadora sabe en qué posición de la memoria inicia el arreglo y cuánta memoria ocupa cada variable indizada, pero (a menos que tomemos medidas especiales) *no sabe cuántas variables indizadas hay en el arreglo*. Es por ello que resulta indispensable

incluir siempre otro argumento `int` que indique a la función el tamaño del arreglo. (Y esa es también la razón por la que un parámetro de arreglo *no es* lo mismo que un parámetro de llamada por referencia. Podemos pensar en un parámetro de arreglo como una forma débil de parámetro de llamada por referencia, en la cual se dice a la función todo acerca del arreglo *excepto su tamaño*.)[2]

Arreglos de argumentos de diferente tamaño pueden sustituir al mismo parámetro de arreglo

Estos parámetros de arreglo tal vez parezcan un poco extraños, pero tienen por lo menos una propiedad muy agradable como resultado directo de su aparentemente extraña definición. La mejor manera de ilustrar esta ventaja es volver a examinar nuestro ejemplo de la función `llenar` que dimos en el cuadro 10.4. *Podemos usar esa misma función para llenar un arreglo de cualquier tamaño*, en tanto el tipo base del arreglo sea `int`. Por ejemplo, supongamos que tenemos las siguientes declaraciones de arreglos:

```
int puntos[5], tiempo[10];
```

La primera de las siguientes llamadas a `llenar` llena el arreglo `puntos` con cinco valores, y la segunda llena el arreglo `tiempo` con diez valores:

```
llenar(puntos, 5);
llenar(tiempo, 10);
```

Podemos usar la misma función con argumentos arreglo de diferentes tamaños, porque el tamaño es un argumento aparte.

El modificador de parámetros `const`

Cuando usamos un argumento arreglo en una llamada de función, la función puede modificar los valores almacenados en el arreglo. Normalmente no hay problema con esto, pero en una definición de función complicada podríamos escribir código que altere de forma inadvertida uno o más de los valores guardados en un arreglo, aunque no se deba modificar dicho arreglo. Como verificación, podemos indicarle a la computadora que no pensamos modificar el argumento arreglo, y entonces la computadora se asegurará de que el código no modifique inadvertidamente algún valor del arreglo. Para decirle a la computadora que un argumento arreglo no debe ser modificado por la función, insertamos el modificador `const` antes del parámetro de arreglo correspondiente. Un parámetro de arreglo modificado con `const` es un **parámetro de arreglo constante**.

`const` parámetro de arreglo constante

Por ejemplo, la siguiente función envía a la salida los valores de un arreglo, pero no los modifica:

```
void mostrar_al_mundo(int a[], int tamanio_de_a)
//Precondición: tamanio_de_a es el tamaño declarado del arreglo a.
//Todas las variables indizadas de a han recibido valores.
//Postcondición: Los valores de a se han escrito en la pantalla.
{
    cout << "El arreglo contiene los siguientes valores:\n";
    for (int i = 0; i < tamanio_de_a; i++)
        cout << a[i] << " ";
    cout << endl;
}
```

[2] Si ha escuchado acerca de apuntadores (y esto suena como apuntadores) ciertamente un argumento arreglo se pasa al enviar un apuntador a su primera variable de índice (0). Esto se discutirá en el capítulo 12. Si usted todavía no ha aprendido sobre apuntadores, puede ignorar sin ningún problema este pie de página.

Parámetros formales y argumentos que son arreglos

Un argumento de una función puede ser un arreglo entero, pero en tal caso no es ni de llamada por valor ni de llamada por referencia; es un nuevo tipo de argumento llamado **argumento arreglo**. Cuando insertamos un argumento arreglo en lugar de un **parámetro de arreglo**, lo único que damos a la función es la dirección en memoria de la primera variable indizada del argumento arreglo (la que tiene el índice 0). El argumento arreglo no le indica a la función el tamaño del arreglo. Por tanto, cuando una función tiene un parámetro de arreglo, normalmente también debe tener otro parámetro formal de tipo `int` que dé el tamaño del arreglo (como en el siguiente ejemplo).

Un argumento arreglo se parece a un argumento de llamada por referencia en el siguiente sentido: si el cuerpo de la función modifica el parámetro de arreglo, entonces cuando la función se invoca lo que en realidad sufre ese cambio es el argumento arreglo. Así pues, una función puede modificar los valores de un argumento arreglo (es decir, puede modificar los valores de sus variables indizadas).

La sintaxis de una declaración de función con un parámetro de arreglo es:

Sintaxis:

Tipo_Devuelto Nombre_de_Funcion(..., *Tipo_Base Nombre_de_Arreglo*[], ...);

Ejemplo:

```
void arreglo_suma(double& suma, double a[], int tamanio);
```

Esta función opera correctamente, pero como medida de seguridad adicional podemos añadir el modificador `const` al encabezado de la función así:

```
void mostrar_al_mundo(const int a[], int tamanio_de_a)
```

Con la adición de este modificador `const`, la computadora desplegará un mensaje de error si la definición de la función contiene un error que modifica cualquiera de los valores del argumento arreglo. Por ejemplo, la siguiente es una versión de la función `mostrar_al_mundo` que contiene un error que inadvertidamente modifica el valor del argumento arreglo. Por fortuna, esta versión de la definición de la función incluye el modificador `const`, y un mensaje de error nos dirá que el arreglo `a` se modifica. Ese mensaje de error ayudará a explicar la equivocación:

```
void mostrar_al_mundo(const int a[], int tamanio_de_a)
//Precondición: tamanio_de_a es el tamaño declarado del arreglo a.
//Todas las variables indizadas de a han recibido valores.
//Postcondición: Los valores de a se han escrito en la
//pantalla.
{
     cout << "El arreglo contiene los siguientes valores:\n";
     for (int i = 0; i < tamanio_de_a; a[i]++)
          cout << a[i] << " ";
     cout << endl;
}
```

Error, pero el compilador no lo notará si no usamos el modificador `const`.

Si no hubiéramos usado el modificador `const` en la definición de función anterior y si cometiéramos el error que se muestra, la función se compilaría y ejecutaría sin mensajes de error. Sin embargo, el código contendría un ciclo infinito que incrementa continuamente `a[0]` y escribe su nuevo valor en la pantalla.

El problema con esta versión incorrecta de `mostrar_al_mundo` es que se incrementa un valor indebido en el ciclo `for`. Se incrementa la variable indizada `a[i]`, cuando lo que debe incrementarse es el índice `i`. En esta versión incorrecta, el índice `i` tiene inicialmente el valor `0` y este valor nunca se modifica. Pero `a[i]`, que es lo mismo que `a[0]`, sí se incrementa. Al incrementarse la variable indizada `a[i]`, se modifica un valor del arreglo, y como incluimos el modificador `const`, la computadora generará un mensaje de advertencia. Ese mensaje deberá darnos una idea de cuál es el problema.

Normalmente tenemos un declaración de función en nuestro programa además de la definición de la función. Cuando usamos el modificador `const` en la definición de una función, también deberemos usarlo en la declaración de la función, a fin de que el encabezado y la declaración sean consistentes.

Podemos usar el modificador `const` con cualquier tipo de parámetro, pero lo normal es usarlo sólo con parámetros de arreglo y parámetros de llamada por referencia para clases, que se discutieron en el capítulo 8.

RIESGO *Uso inconsistente del modificador de parámetros* `const`

El modificador de parámetros `const` es una cuestión de todo o nada. Si lo usamos para un parámetro de arreglo de un tipo dado, deberemos usarlo para todos los demás parámetros de arreglo que tengan ese tipo y no sean modificados por la función. La razón tiene que ver con las llamadas a funciones dentro de llamadas a funciones. Consideremos la definición de la función `mostrar_diferencia`, que se da a continuación junto con la declaración de una función que se usa en la definición:

```
double calcular_promedio(int a[], int numero_usado);
//Devuelve el promedio de los primeros numero_usado elementos
//del arreglo a. El arreglo a no se modifica.

void mostrar_diferencia(const int a[], int numero_usado)
{
     double promedio = calcular_promedio(a, numero_usado);
     cout << "Promedio de los " << numero_usado
          << " numeros = " << promedio << endl
          << "Los numeros son:\n";
     for (int indice = 0; indice < numero_usado; indice++)
          cout << a[indice] << " difiere del promedio en "
               << (a[indice] - promedio) << endl;
}
```

Lo anterior producirá un mensaje de error o de advertencia con algunos compiladores. La función `calcular_promedio` no modifica su parámetro `a`, pero cuando el compilador procesa la definición de la función `mostrar_diferencia` pensará que `calcular_promedio` sí modifica el valor de su parámetro `a` (o al menos que podría hacerlo). La razón es que, cuando el compilador está

traduciendo la definición de la función `mostrar_diferencia`, lo único que conoce de la función `calcular_promedio` es la declaración de función, y ésta no contiene un *`const`* que le indique al compilador que el parámetro `a` no se modificará. Por tanto, si usamos *`const`* con el parámetro `a` en la función `mostrar_diferencia`, también deberemos usarlo con el parámetro `a` en la función `calcular_promedio`. La declaración de la función `calcular_promedio` deberá ser la siguiente:

```
double calcular_promedio(const int a[], int numero_usado);
```

Funciones que devuelven un arreglo

Una función no debe devolver un arreglo de la misma manera que devuelve un valor de tipo *`int`* o *`double`*. Hay una manera para obtener algo más o menos equivalente a una función que devuelve un arreglo. Lo que se debe hacer es devolver un apuntador a un arreglo. Sin embargo, hasta ahora no hemos visto los apuntadores. Analizaremos cómo devolver un apuntador a un arreglo cuando hablemos acerca de la interacción de arreglos y apuntadores en el capítulo 12. Hasta ese momento no podremos escribir una función que devuelva un arreglo.

CASO DE ESTUDIO Gráfico de producción

En este caso de estudio usaremos arreglos en el diseño descendente de un programa. Como argumentos de las funciones que realizan subtareas emplearemos tanto variables indizadas como arreglos enteros.

Definición del problema

La Compañía Fabricante de Cucharas de Plástico Apex nos ha encargado escribir un programa que despliegue un gráfico de barras para ilustrar la productividad de cada una de sus cuatro plantas manufactureras durante cualquier semana dada. Las plantas mantienen cifras de producción individuales para cada departamento, como el departamento de cucharitas, el de cucharas soperas, el de cucharas blancas para coctel, el de cucharas de color para coctel, etcétera. Además, cada planta tiene diferente número de departamentos. Por ejemplo, sólo una planta fabrica cucharas de color para coctel. Las entradas se introducen planta por planta y consisten en una lista de números que dan la producción de cada departamento de la planta. La salida consistirá en un gráfico de barras como la siguiente:

```
Planta #1 **********
Planta #2 **********
Planta #3 ********************
Planta #4 *****
```

Cada asterisco representa 1000 unidades de producción.

Decidimos leer las entradas individualmente para cada departamento de una planta. Puesto que los departamentos no pueden producir cantidades negativas de cucharas, sabemos que la cifra de producción de cada departamento será no negativa. Por lo tanto, podemos usar un valor negativo como centinela para marcar el final de las cifras de producción de cada planta.

Puesto que la producción se da en unidades de millares, debemos ajustar su escala dividiéndola entre 1000. Esto presenta un problema porque la computadora debe desplegar un número entero de asteriscos; no puede mostrar 1.6 asteriscos para indicar 1600 unidades. Por tanto, redondearemos las cifras al millar más cercano. Así, 1600 será igual a 2000 y producirá dos asteriscos. A continuación describimos con precisión las entradas y salidas del programa:

ENTRADA

Hay cuatro plantas manufactureras numeradas de la 1 al 4. Se dan las siguientes entradas para cada planta una lista de números que dan la producción de cada departamento de la planta. La lista termina con un valor negativo que actúa como centinela.

SALIDA

Un gráfico barras que muestra la producción total de cada planta. Cada asterisco del gráfico equivale a 1000 unidades. La producción de cada planta se redondea al millar de unidades más cercano.

Análisis del problema

Emplearemos un arreglo llamado `produccion`, que contendrá la producción total de cada una de las cuatro plantas. En C++ los índices de arreglos siempre comienzan con `0`, pero como las plantas están numeradas del 1 al 4, no del 0 al 3, no usaremos el número de planta como índice del arreglo. En vez de ello, colocaremos la producción total de la planta número `n` en la variable indizada `produccion[n-1]`. La producción total de la planta número 1 se guardará en `produccion[0]`, las cifras para la planta 2 estarán en `produccion[1]`, etcétera.

Puesto que la producción se da en millares de unidades, el programa ajustará la escala de los elementos del arreglo. Si la producción total de la planta número 3 es de 4040 unidades, el valor que se asignará inicialmente a `produccion[2]` será `4040`. Luego, se cambiará la escala de este valor de modo que el valor almacenado en `produccion[2]` sea `4` y se desplieguen cuatro asteriscos en el gráfico para representar la producción de la planta 3.

subtareas

La tarea de nuestro programa se puede dividir en las siguientes subtareas:

- `capturar_datos`: Leer los datos de entrada de cada planta y asignar a la variable indizada `produccion[numero_de_planta-1]` un valor igual a la producción total de esa planta, donde `numero_de_planta` es el número de la planta.
- `escala`: Para cada `numero_de_planta`, modificar el valor de la variable indizada `produccion[numero_de_planta-1]` para que dé el número correcto de asteriscos.
- `graficar`: Despliegue el gráfico de barras.

El arreglo `produccion` entero será un argumento de las funciones que realizan estas subtareas. Como es normal con los parámetros de arreglo, esto implica que necesitamos un parámetro formal adicional para el tamaño del arreglo, que en este caso es igual al número de plantas. Usaremos una constante definida para el número de plantas, y esta constante servirá como tamaño del arreglo `produccion`. En el cuadro 10.5 se muestra la parte `main` del programa, junto con las declaraciones de las funciones de las funciones que realizan las subtareas y la constante definida para el número de plantas. Observe que, dado que no hay razón para modificar el parámetro de arreglo de la función `graficar`, lo hemos convertido en un parámetro constante añadiendo el modificador *`const`*. El material

de el cuadro 10.5 es el esqueleto de nuestro programa, y si está en un archivo aparte lo podremos compilar para encontrar cualesquier errores de sintaxis antes de definir las funciones que corresponden a las declaraciones de función que se muestran.

Después de compilar el archivo del cuadro 10.5, estaremos listos para diseñar la implementación de las funciones de las tres subtareas. Para cada una de estas tres funciones, diseñaremos un algoritmo, escribiremos el código de la función, y probaremos la función antes de diseñar la siguiente función.

Diseño del algoritmo para `capturar_datos`

La declaración de función y el comentario descriptivo de la función `capturar_datos` se muestran en el cuadro 10.5. Como se indica en el cuerpo de la parte `main` del programa (que también aparece en el cuadro 10.5), cuando se invoque `capturar_datos` el parámetro formal `a` será sustituido por el arreglo `produccion`, y dado que el último número de planta es igual al número de plantas, el parámetro formal `ultimo_numero_de_planta` será sustituido por `NUMERO_DE_PLANTAS`. El algoritmo de `capturar_datos` es sencillo:

Para `numero_de_planta` igual a `1`, `2`, etcétera, hasta `ultimo_numero_de_planta` hacer lo siguiente:

Leer todos los datos de la planta cuyo número es `numero_de_planta`.

Sumar los números.

Hacer `produccion[numero_de_planta-1]` igual a ese total.

Codificación de `capturar_datos`

El algoritmo de la función `capturar_datos` se traduce en el siguiente código:

```
//Usa iostream:
void capturar_datos(int a[], int ultimo_numero_de_planta)
{
    using namespace std;
    for (int numero_de_planta = 1;
         numero_de_planta <= ultimo_numero_de_planta;
         numero_de_planta++)
    {
         cout << endl
              << "Escriba los datos de producción de la planta
              numero "
              << numero_de_planta << endl;
         obtener_total(a[numero_de_planta - 1]);
    }
}
```

Este código es cosa de rutina porque todo el trabajo corre por cuenta de la función `obtener_total`, que todavía no hemos diseñado. Pero antes de analizar esa función, observemos algunos aspectos de la función `capturar_datos`. Tome nota de que almacenamos las cifras de la planta número `numero_de_planta` en la variable indizada cuyo índice es `numero_de_planta-1`. La razón es que los índices de arreglos siempre comienzan con `0`, mientras que los números de planta comienzan con 1. Observe también que usamos una variable indizada como argumento de la función `obtener_total`. La función `obtener_total` se encarga realmente de todo el trabajo de la función `capturar_datos`.

`obtener_total`

La función `obtener_total` se encarga de todas las operaciones de entrada para una planta. Lee las cifras de producción de esa planta, calcula la sumatoria de las cifras y

CUADRO 10.5 *Bosquejo para el programa graficador*

```
//Lee datos y muestra una gráfica de la productividad de cada planta.
#include <iostream>
const int NUMERO_DE_PLANTAS = 4;

void capturar_datos(int a[], int ultimo_numero_de_planta);
//Precondición: ultimo_numero_de_planta es el tamaño declarado del arreglo a.
//Postcondición: Para numero_de_planta = 1 hasta ultimo_numero_de_planta:
//a[numero_de_planta-1] es la producción total de la planta numero_de_planta.

void escala(int a[], int tamanio);
//Precondición: a[0] a a[tamanio-1] tienen valores no negativos.
//Postcondición: a[i] se convirtió a los millares (redondeando a enteros)
//que había originalmente en a[i], para toda i tal que 0 <= i <= tamanio-1.

void graficar(const int num_asteriscos[], int ultimo_numero_de_planta);
//Precondición: num_asteriscos[0], a num_asteriscos[ultimo_numero_de_planta-1]
//tienen valores no negativos.
//Postcondición: Se mostró una gráfica de barras que indica que la planta
//número N produjo num_asteriscos[N-1] millares de unidades, para cada N tal
//que 1 <= N <= ultimo_numero_de_planta.

int main()
{
    using namespace std;
    int produccion[NUMERO_DE_PLANTAS];

    cout << "Este programa muestra una grafica de la\n"
         << "produccion de cada planta de la compania.\n";

    capturar_datos(produccion, NUMERO_DE_PLANTAS);
    escala(produccion, NUMERO_DE_PLANTAS);
    graficar(produccion, NUMERO_DE_PLANTAS);

    return 0;
}
```

guarda el total en la variable indizada para esa planta. Sin embargo, `obtener_total` no necesita saber que su argumento es una variable indizada. Para una función como `obtener_total`, una variable indizada es como cualquier otra variable de tipo *`int`*. Por tanto, `obtener_total` tendrá un parámetro de llamada por referencia ordinario de tipo *`int`*. Eso implica que `obtener_total` no es más que una función de entrada ordinaria como otras que vimos antes de entrar en el tema de los arreglos. La función `obtener_total` lee una lista de números que termina con un valor centinela, obtiene la sumatoria de los números conforme los lee, y hace que el valor de su argumento, que es una variable de tipo *`int`*, sea igual a la sumatoria. La función `obtener_total` no contiene nada nuevo para nosotros. En el cuadro 10.6 se muestran las definiciones de las funciones `obtener_total` y `capturar_datos`. Las funciones se han incorporado en un programa de prueba sencillo.

Prueba de `capturar_datos`

Toda función debe probarse en un programa en el que sea la única función que no se ha probado. La función `capturar_datos` incluye una llamada a la función `obtener_total`. Por tanto, deberemos probar `obtener_total` en su propio programa controlador. Una vez probada exhaustivamente `obtener_total`, podremos usarla en un programa, como el del cuadro 10.6, para probar la función `capturar_datos`.

Al probar la función `capturar_datos`, deberemos incluir pruebas con todas las posibles cifras de producción de una planta. Deberemos incluir una planta sin cifras de producción (como hicimos con la planta 4 en el cuadro 10.6), deberemos incluir una prueba para una planta con una sola cifra de producción (como en el caso de la planta 3 en el cuadro 10.6) y una para una planta con varias cifras de producción (como las plantas 1 y 2 en el cuadro 10.6). Deberemos probar cifras de producción de cero y distintas de cero, y es por ello que incluimos un `0` en la lista de entrada de la planta 2 en el cuadro 10.6

Diseño del algoritmo para `escala`

La función `escala` modifica el valor de todas las variables indizadas del arreglo `produccion` de modo que indique el número de asteriscos que hay que desplegar. Puesto que debe haber un asterisco por cada 1000 unidades de producción, el valor de cada variable indizada se deberá dividir entre `1000.0`. Luego, a fin de obtener un número entero de asteriscos, el resultado se redonderá al entero más cercano. Podemos usar este método para cambiar la escala de los valores de cualquier arreglo `a` de cualquier tamaño, y es por ello que la declaración de función de `escala`, que mostramos en el cuadro 10.5 y repetimos a continuación, se expresa en términos de un arreglo arbitrario `a` de tamaño arbitrario:

```
void escala(int a[], int tamanio);
//Precondición: a[0] a a[tamanio−1] tienen valores
//no negativos.
//Postcondición: a[i] se convirtió a los millares
//(redondeando a enteros) que había originalmente en a[i],
//para toda i tal que 0 <= 1 <= tamanio−1.
```

Cuando invoquemos la función `escala`, el parámetro de arreglo `a` será sustituido por el arreglo `produccion`, y el parámetro formal `tamanio` será sustituido por `NUMERO_DE_PLANTAS`, así que la llamada a la función será:

```
escala(produccion, NUMERO_DE_PLANTAS);
```

CUADRO 10.6 *Prueba de la función* capturar_datos *(parte 1 de 3)*

```
//Prueba la función capturar_datos.
#include <iostream>
const int NUMERO_DE_PLANTAS = 4;

void capturar_datos(int a[], int ultimo_numero_de_planta);
//Precondición: ultimo_numero_de_planta es el tamaño declarado del arreglo a.
//Postcondición: Para numero_de_planta = 1 hasta ultimo_numero_de_planta:
//a[numero_de_planta-1] es la producción total de la planta numero_de_planta.

void obtener_total(int& sumatoria);
//Lee enteros no negativos del teclado y coloca
//su total en sumatoria.

int main ()
{
    using namespace std;
    int produccion[NUMERO_DE_PLANTAS];
    char respuesta;

    do
    {
        capturar_datos(produccion, NUMERO_DE_PLANTAS);
        cout << endl
             << "Produccion total de cada"
             << " planta de la 1 a la 4:\n";
        for (int numero = 1; numero <= NUMERO_DE_PLANTAS; numero++)
        cout << produccion[numero - 1] << " ";

        cout << endl
             << "Probar otra vez? (Escriba s o n y oprima Intro): ";
        cin >> respuesta;
    }while ( (respuesta != 'N') && (respuesta != 'n') );

    cout << endl;

    return 0;
}
```

CUADRO 10.6 ***Prueba de la función*** `capturar_datos` ***(parte 2 de 3)***

```
//Usa iostream:
void capturar_datos(int a[], int ultimo_numero_de_planta)
{
    using namespace std;
    for (int numero_de_planta = 1;
                    numero_de_planta <= ultimo_numero_de_planta; numero_de_planta++)
    {
        cout << endl
             << "Escriba los datos de produccion de la planta numero "
             << numero_de_planta << endl;
        obtener_total(a[numero_de_planta - 1]);
    }
}


//Usa iostream:
void obtener_total(int& sumatoria)
{
    using namespace std;
    cout << "Escriba las unidades producidas por cada departamento.\n"
         << "Anexe un numero negativo al final de la lista.\n";

    sumatoria = 0;
    int siguiente;
    cin >> siguiente;
    while (siguiente >= 0)
    {
       sumatoria = sumatoria + siguiente;
       cin >> siguiente;
    }

    cout << "Total = " << sumatoria << endl;
}
```

CUADRO 10.6 *Prueba de la función* `capturar_datos` *(parte 3 de 3)*

Diálogo de ejemplo

```
Escriba los datos de produccion de la planta numero 1
Escriba las unidades producidas por cada departamento.
Anexe un numero negativo al final de la lista.
1 2 3 -1
Total = 6
Escriba los datos de produccion de la planta numero 2
Escriba las unidades producidas por cada departamento.
Anexe un numero negativo al final de la lista.
0 2 3 -1
Total = 5
Escriba los datos de produccion de la planta numero 3
Escriba las unidades producidas por cada departamento.
Anexe un numero negativo al final de la lista.
2 -1
Total = 2
Escriba los datos de produccion de la planta numero 4
Escriba las unidades producidas por cada departamento.
Anexe un numero negativo al final de la lista.
-1
Total = 0
Produccion total de cada planta de la 1 a la 4:
6 5 2 0
Probar otra vez? (Escriba S o N y oprima Intro): n
```

El algoritmo de la función `escala` es:

```
for (int indice = 0; indice < tamanio; indice++)
```

Dividir el valor de `a[indice]` entre mil y redondear el resultado al entero más cercano; el resultado es el nuevo valor de `a[indice]`.

Codificación de `escala`

El algoritmo de `escala` se traduce en el código C++ que se da a continuación, donde `redondear` es una función que todavía no definimos. La función `redondear` recibe un argumento de tipo *`double`* y devuelve un valor de tipo *`int`* que es el entero más cercano a

su argumento, es decir, la función `redondear` redondea su argumento al entero más cercano:

```
void escala(int a[], int tamanio)
{
     for (int indice = 0; indice < tamanio; indice++)
         a[indice] = redondear(a[indice]/1000.0 );
}
```

Observe que dividimos entre `1000.0`, no entre `1000` (sin el punto decimal). Si hubiéramos dividido entre `1000`, habríamos efectuado una división entera. Por ejemplo, `2600/1000` daría la respuesta `2`, pero `2600/1000.0` da la respuesta `2.6`. es verdad que queremos un entero como respuesta final después de redondear, pero queremos que `2600` dividido entre mil produzca `3`, no `2`, cuando se redondee a un número entero.

redondear

Pasamos ahora a la definición de la función `redondear`, que redondea su argumento al entero más cercano. Por ejemplo, `redondear(2.3)` devuelve `2`, y `redondear(2.6)` devuelve `3`. El código de la función `redondear`, junto con el de `escala`, se da en el cuadro 10.7. El código de `redondear` tal vez requiera cierta explicación.

La función `redondear` usa la función predefinida `floor` de la biblioteca con archivo de encabezado `cmath`. La función `floor` devuelve el número entero inmediatamente menor que su argumento. Por ejemplo, tanto `floor(2.1)` como `floor(2.9)` devuelven `2`. Para constatar que `redondear` funciona correctamente, veamos algunos ejemplos. Consideremos `redondear(2.4)`. El valor devuelto es

```
floor(2.4 + 0.5)
```

que es `floor(2.9)`, o sea `2.0`. De hecho, cualquier número que sea mayor o igual que `2.0` y estrictamente menor que `2.5` dará un número menor que `3.0` si se le suma `0.5`. Por tanto, `floor` aplicada a ese número más `0.5` devolverá `2.0`. Entonces, la invocación de `redondear` con cualquier número que sea mayor o igual que `2.0` y menor que `2.5` devolverá `2`. (Puesto que la declaración de función de `redondear` especifica que el tipo del valor devuelto es `int`, el valor calculado de `2.0` se convertirá en el valor entero `2` sin punto decimal utilizando `static_cast<int>`.)

Consideremos ahora los números mayores o iguales que `2.5`; por ejemplo, `2.6`. El valor devuelto por la llamada `redondear(2.6)` es

```
floor(2.6 + 0.5)
```

que es `floor(3.1)`, o sea `3.0`. De hecho, cualquier número mayor o igual que `2.5` y menor o igual que `3.0` dará un número mayor que 3.0 si le sumamos `0.5`. Por tanto, la invocación de `redondear` con cualquier número mayor o igual que `2.5` y menor o igual que `3.0` devolverá `3`.

Así pues, `redondear` funciona correctamente con todos los argumentos entre `2.0` y `3.0`. Es evidente que los argumentos entre `2.0` y `3.0` nada tienen de especial. Un razonamiento similar aplica a todos los números no negativos. Por lo tanto, `redondear` funciona correctamente con cualquier argumento no negativo.

Prueba de `escala`

El cuadro 10.7 contiene un programa para demostrar la función `escala`, pero los programas que prueban las funciones `redondear` y `escala` deberán ser más completos que este sencillo programa. En particular, deberán permitirnos probar la función varias veces,

CUADRO 10.7 *La función* escala *(parte 1 de 2)*

```
//Programa que demuestra la función escala.
#include <iostream>
#include <cmath>

void escala(int a[], int tamanio);
//Precondición: a[0] a a[tamanio-1] tienen valores no negativos.
//Postcondición: a[i] se convirtió a los millares (redondeando a enteros)
//que había originalmente en a[i], para toda i tal que 0 <= i <= tamanio-1.

int redondear(double numero);
//Precondición: numero >= 0.
//Devuelve numero redondeado al entero más cercano.

int main()
{
    using namespace std;
    int un_arreglo[4], indice;

    cout <<"Escriba 4 numeros para cambiar su escala: ";
    for (indice = 0; indice < 4; indice++)
        cin >> un_arreglo[indice];

    escala(un_arreglo, 4);

    cout << "Los valores convertidos en millares son: ";
    for (indice = 0; indice < 4; indice++)
         cout << un_arreglo[indice] << " ";
    cout << endl;

    return 0;
}

void escala (int a[], int tamanio)
{
    for (int indice = 0; indice < tamanio; indice++)
        a[indice] = redondear(a[indice]/1000.0);
}

//Usa cmath:
int redondear(double numero)
{
    using namespace std;
    return static_cast<int>(floor(numero + 0.5));
}
```

CUADRO 10.7 *La función* `escala` *(parte 2 de 2)*

Diálogo de ejemplo

```
Escriba 4 numeros para cambiar su escala: 2600 999 465 3501
Los valores convertidos en millares son: 3 1 0 4
```

no sólo una. No daremos los programas de prueba completos, pero hay que probar primero `redondear` (el cual es usado por `escala`) en su propio programa controlador, y luego probar `escala` en cualquier otro programa controlador. El programa para probar `redondear` deberá probar argumentos que sean `0`, argumentos que se redondeen hacia arriba como `2.6` y argumentos que se redondeen hacia abajo como `2.3`. El programa que pruebe `escala` deberá probar una variedad similar de valores para los elementos del arreglo.

La función `graficar`

En el cuadro 10.8 se muestra el programa completo para producir la gráfica de barras deseada. No explicaremos paso a paso el diseño de la función `graficar` porque es muy sencilla.

Ejercicios de AUTOEVALUACIÓN

13. Escriba una definición para una función llamada `mas_uno` que tiene un parámetro formal para un arreglo de enteros e incrementa en uno el valor de cada elemento del arreglo. Añada cualesquier otros parámetros formales que se necesiten.

14. Considere la definición de función

```
void dos(int a[], int cuantos)
{
     for (int indice = 0; indice < cuantos; indice++)
          a[indice] = 2;
}
```

CUADRO 10.8 *Programa de gráfica de producción (parte 1 de 3)*

```
//Lee datos y muestra una gráfica de la productividad de cada planta.
#include <iostream>
#include <cmath>
const int NUMERO_DE_PLANTAS = 4;

void capturar_datos(int a[], int ultimo_numero_de_planta);
//Precondición: ultimo_numero_de_planta es el tamaño declarado del arreglo a.
//Postcondición: Para numero_de_planta = 1 hasta ultimo_numero_de_planta:
//a[numero_de_planta-1] es la producción total de la planta numero_de_planta.

void escala(int a[], int tamanio);
//Precondición: a[0] a a[tamanio-1] tienen valores no negativos.
//Postcondición: a[i] se convirtió a los millares (redondeando a enteros)
//que había originalmente en a[i], para toda i tal que 0 <= i <= tamanio-1.

void graficar(const int num_asteriscos[], int ultimo_numero_de_planta);
//Precondición: num_asteriscos[0] a num_asteriscos[ultimo_numero_de_planta-1]
// tienen valores no negativos.
//Postcondición: Se mostró una gráfica de barras que indica que la planta
//número N produjo num_asteriscos[N-1] millares de unidades, para cada N tal
//que 1 <= N <= ultimo_numero_de_planta.

void obtener_total(int& sumatoria);
//Lee enteros no negativos del teclado y coloca
//su total en sumatoria.

int redondear(double numero);
//Precondición: numero >= 0.
//Devuelve numero redondeado al entero más cercano.

void escribir_asteriscos(int n);
//Escribe n asteriscos en la pantalla.

int main()
{
    using namespace std;
    int produccion[NUMERO_DE_PLANTAS];

    cout << "Este programa muestra una grafica de la\n"
         << "produccion de cada planta de la compañía.\n";
```

CUADRO 10.8 *Programa de gráfica de producción (parte 2 de 3)*

```
    capturar_datos(produccion, NUMERO_DE_PLANTAS);
    escala(produccion, NUMERO_DE_PLANTAS);
    graficar(produccion, NUMERO_DE_PLANTAS);
    return 0;
}
//Usa iostream:
void capturar_datos(int a[], int ultimo_numero_de_planta)
<El resto de la definición de capturar_datos se da en el cuadro 10.6.>

//Usa iostream:
void obtener_total(int& sumatoria)
<El resto de la definición de obtener_total se da en el cuadro 10.6.>

void escala(int a[], int tamanio)
<El resto de la definición de escala se da en el cuadro 10.7.>

//Usa cmath:
int redondear(double numero)
<El resto de la definición de redondear se da en el cuadro 10.7.>

//Usa iostream:
void graficar(const int num_asteriscos[], int ultimo_numero_de_planta)
{
    using namespace std;
    cout << "\nUnidades producidas en millares de unidades:\n";
    for (int numero_de_planta = 1;
                numero_de_planta <= ultimo_numero_de_planta; numero_de_planta++)
    {
           cout << "Planta #" << numero_de_planta << " ";
           escribir_asteriscos(num_asteriscos[numero_de_planta - 1]);
           cout << endl;
    }
}
//Usa iostream:
void escribir_asteriscos(int n)
{
    using namespace std;
    for (int cuenta = 1; cuenta <= n; cuenta++)
        cout << "*";
}
```

CUADRO 10.8 *Programa de gráfica de producción (parte 3 de 3)*

Diálogo de ejemplo

```
Este programa muestra una grafica de la
produccion de cada planta de la compania.

Escriba los datos de produccion de la planta numero 1
Escriba las unidades producidas por cada departamento.
Anexe un numero negativo al final de la lista.
2000 3000 1000 -1
Total = 6000

Escriba los datos de produccion de la planta numero 2
Escriba las unidades producidas por cada departamento.
Anexe un numero negativo al final de la lista.
2050 3002 1300 -1
Total = 6352

Escriba los datos de produccion de la planta numero 3
Escriba las unidades producidas por cada departamento.
Anexe un numero negativo al final de la lista.
5000 4020 500 4348 -1
Total = 13868

Escriba los datos de produccion de la planta numero 4
Escriba las unidades producidas por cada departamento.
Anexe un numero negativo al final de la lista.
2507 6050 1809 -1
Total = 10366

Unidades producidas en millares de unidades:
Planta #1 ******
Planta #2 ******
Planta #3 **************
Planta #4 **********
```

¿Cuáles de las siguientes son llamadas de función aceptables?

```
int mi_arreglo[29];
dos(mi_arreglo, 29);
dos(mi_arreglo, 10);
dos(mi_arreglo, 55);
"Hey, dos. Ven por favor."
int tu_arreglo[100];
dos(tu_arreglo, 100);
dos(mi_arreglo[3], 29);
```

15. Inserte *const* antes de cualquiera de los siguientes parámetros de arreglo que se puedan convertir en parámetros de arreglo constantes:

```
void salida(double a[], int tamanio);
//Precondición: a[0] hasta a[tamanio - 1] tienen valores.
//Postcondición: Se enviaron a[0] hasta a[tamanio - 1] a la
//salida.
void omitir_impares(int a[], int tamanio);
//Precondición: a[0] hasta a[tamanio - 1] tienen valores.
//Postcondición: Todos los números impares en a[0] hasta
//a[tamanio - 1] se han convertido en 0.
```

16. Escriba una función *int* llamada `en_desorden` que reciba como parámetros un arreglo de tipo *double* y un *int* como tamaño. Esta función determinará si el arreglo está en desorden, es decir, si viola la condición

```
a[0] <= a[1] <= a[2] <= ...
```

La función devuelve `-1` si los elementos no están en desorden; de lo contrario, devolverá el índice del primer elemento del arreglo que esté en desorden. Por ejemplo, considere la declaración

```
double a[10] = {1.2, 2.1, 3.3, 2.5, 4.5,
                7.9, 5.4, 8.7, 9.9, 1.0};
```

En el arreglo anterior, `a[2]` y `a[3]` son el primer par que no está en orden, `a[3]` es el primer elemento en desorden, por lo que la función devuelve `3`. Si el arreglo estuviera ordenado, la función devolvería `-1`.

10.3 Programación con arreglos

Nunca confíes en las impresiones generales, muchacho;
concéntrate en los detalles.
Sir Arthur Conan Doyle, *Un caso de identidad* (Sherlock Holmes)

En esta sección discutiremos los arreglos parcialmente llenos y daremos una breve introducción tanto al ordenamiento como a la búsqueda de arreglos. Esta sección no incluye material nuevo acerca del lenguaje C++, pero incluye más prácticas con parámetros de arreglos.

Arreglos parcialmente llenos

Es común que en el momento de escribir un programa no se conozca el tamaño exacto que debe tener un arreglo, o que el tamaño varíe de una ejecución del programa a otra. Una forma común y fácil de manejar esta situación es declarar un arreglo del tamaño más grande que el programa podría necesitar. Así, el programa podrá usar tanto del arreglo como requiera.

arreglos parcialmente llenos

Los arreglos parcialmente llenos requieren cierto cuidado. El programa debe saber qué tanto del arreglo se ha usado y no debe hacer referencia a ninguna variable indizada que no haya recibido un valor. El programa del cuadro 10.9 ilustra este punto. Lee una lista de puntajes de golf e indica qué tanto difiere cada puntaje del promedio. Este programa funciona con listas que tengan entre uno y diez puntajes inclusive. Los puntajes se guardan en el arreglo `puntos`, que tiene 10 variables indizadas, pero el programa sólo usa tanto del arreglo como necesita. La variable `elem_usados` sigue la pista al número de elementos que se almacenan en el arreglo. Los elementos (o sea los puntajes) se guardan en las posiciones `puntos[0]` a `puntos[elem_usados - 1]`.

Los detalles son muy similares a lo que tendríamos si declaráramos `elem_usados` como tamaño del arreglo y usáramos todo el arreglo. En particular, la variable `elem_usados` siempre debe ser uno de los argumentos de cualquier función que manipule el arreglo parcialmente lleno. Puesto que el argumento `elem_usados` (si se usa correctamente) puede evitar que la función haga referencia a un índice de arreglo no válido, esto a veces (pero no siempre) hace innecesario un argumento que dé el tamaño declarado del arreglo. Por ejemplo, las funciones `mostrar_diferencia` y `calcular_promedio` usan el argumento `elem_usados` para asegurar que sólo se empleen índices de arreglo válidos. Sin embargo, la función `llenar_arreglo` necesita conocer el tamaño máximo declarado para el arreglo, a fin de no excederse al llenar el arreglo.

TIP DE PROGRAMACIÓN

No escatime en parámetros formales

Examine la función `llenar_arreglo` del cuadro 10.9. Cuando se invoca `llenar_arreglo`, el tamaño declarado del arreglo `MAXIMO_DE_PUNTAJES` se da como uno de los argumentos, como muestra la siguiente llamada de función del cuadro 10.9:

```
llenar_arreglo(puntos, MAXIMO_DE_PUNTAJES, elem_usados);
```

El lector podría protestar que `MAXIMO_DE_PUNTAJES` es una constante definida globalmente y por tanto podría usarse en la definición de `llenar_arreglo` sin convertirla en un argumento. El lector tendría razón, y si no usáramos `llenar_arreglo` en ningún otro programa fuera del cuadro 10.9 podríamos no incluir `MAXIMO_DE_PUNTAJES` como argumento de `llenar_arreglo`. Sin embargo, `llenar_arreglo` es una función de utilidad general que podríamos querer usar en varios programas distintos. De hecho, volvemos a usar la función `llenar_arreglo` en el programa del cuadro 10.10. En ese programa, el argumento para el tamaño declarado del arreglo es una constante global que lleva un nombre distinto. Si hubiéramos incluido la constante global `MAXIMO_DE_PUNTAJES` en el cuerpo de

CUADRO 10.9 *Arreglo parcialmente lleno (parte 1 de 3)*

```
//Muestra la diferencia entre cada puntaje de golf de una lista, y el promedio.
#include <iostream>
const int MAXIMO_DE_PUNTAJES = 10;

void llenar_arreglo(int a[], int tamanio, int& elem_usados);
//Precondición: tamanio es el tamaño declarado del arreglo a.
//Postcondición: elem_usados es el número de valores guardados en a.
//a[0] a a[elem_usados-1] se han llenado con
//enteros no negativos leídos del teclado.

double calcular_promedio(const int a[], int elem_usados);
//Precondición: a[0] a a[elem_usados-1] tienen valores; elem_usados > 0.
//Devuelve el promedio de los números a[0] a a[elem_usados-1].

void mostrar_diferencia(const int a[], int elem_usados);
//Precondición: Las primeras elem_usados variables indizadas de a tienen valores.
//Postcondición: Produce salidas en la pantalla que indican
//qué tanto difieren de su promedio los primeros elem_usados elementos de a.

int main()
{
    using namespace std;
    int puntos[MAXIMO_DE_PUNTAJES], elem_usados;

    cout << "Este programa lee puntajes de golf e indica\n"
         << "que tanto difiere cada uno del promedio.\n";

    cout << "Escriba puntajes de golf:\n";
    llenar_arreglo(puntos, MAXIMO_DE_PUNTAJES, elem_usados);
    mostrar_diferencia(puntos, elem_usados);

    return 0;
}
//Usa iostream:
void llenar_arreglo(int a[], int tamanio, int& elem_usados)
{
    using namespace std;
    cout << "Escriba hasta " << tamanio << " numeros enteros no negativos.\n"
         << "Marque el final de la lista con un numero negativo.\n";
```

CUADRO 10.9 *Arreglo parcialmente lleno (parte 2 de 3)*

```
    int siguiente, indice = 0;
    cin >> siguiente;
    while ((siguiente >= 0) && (indice < tamanio))
    {
            a[indice] = siguiente;
            indice++;
            cin >> siguiente;
    }
    elem_usados = indice;
}

double calcular_promedio(const int a[], int elem_usados)
{
    double total = 0;
    for (int indice = 0; indice < elem_usados; indice++)
        total = total + a[indice];
    if (elem_usados > 0)
    {
       return (total/elem_usados);
    }
    else
    {
        using namespace std;
        cout << "ERROR: el numero de elementos en calcular_promedio es 0.\n"
             << "calcular_promedio devuelve 0.\n";
        return 0;
    }
}

void mostrar_diferencia(const int a[], int elem_usados)
{
    using namespace std;
    double promedio = calcular_promedio(a, elem_usados);
    cout << "Promedio de los " << elem_usados
           << " puntajes = " << promedio << endl
           << "Los puntajes son:\n";
    for (int indice = 0; indice < elem_usados; indice++)
    cout << a[indice] << " difiere del promedio en "
         << (a[indice] - promedio) << endl;
}
```

CUADRO 10.9 *Arreglo parcialmente lleno (parte 3 de 3)*

Diálogo de ejemplo

```
Este programa lee puntajes de golf e indica
que tanto difiere cada uno del promedio.
Escriba puntajes de golf:
Escriba hasta 10 numeros enteros no negativos.
Marque el final de la lista con un numero negativo.
69 74 68 -1
Promedio de los 3 puntajes = 70.3333
Los puntajes son:
69 difiere del promedio en -1.33333
74 difiere del promedio en 3.66667
68 difiere del promedio en -2.33333
```

la función `llenar_arreglo`, no habríamos podido reutilizar la función en el programa del cuadro 10.10.

Incluso si sólo usáramos `llenar_arreglo` en un programa, sería recomendable hacer que el tamaño declarado del arreglo sea un argumento de `llenar_arreglo`. Al ver el tamaño declarado del arreglo como argumento recordaremos que esta información es crucial para la función.

EJEMPLO DE PROGRAMACIÓN
Búsqueda en un arreglo

Una tarea de programación común consiste en buscar un valor dado en un arreglo. Por ejemplo, el arreglo podría contener el número de estudiante de cada uno de los estudiantes de un curso dado. Para saber si cierto estudiante está inscrito, buscamos en el arreglo el número de ese estudiante. El sencillo programa del cuadro 10.10 llena un arreglo y luego busca en el arreglo valores especificados por el usuario. Un programa de aplicación real sería mucho más completo, pero éste muestra todos los componentes esenciales del algoritmo de *búsqueda secuencial*. La **búsqueda secuencial** es el algoritmo de búsqueda más sencillo imaginable: El programa examina los elementos del arreglo en orden desde el primero hasta el último para ver si el número buscado es igual a cualquiera de los valores contenidos en los elementos del arreglo.

búsqueda secuencial

En el cuadro 10.10 se usa la función `buscar` para examinar el arreglo. Al buscar en un arreglo, bien puede ser que nos interese averiguar algo más que si el valor buscado está o no en el arreglo. Si el valor buscado está en el arreglo, es común que queramos conocer el índice de la variable indizada que contiene dicho valor, pues ese índice podría servir como guía para obtener información adicional acerca del valor buscado. Por tanto, hemos diseñado la función `buscar` de modo que devuelva un índice que dé la ubicación del valor

`buscar`

CUADRO 10.10 *Búsqueda en un arreglo (parte 1 de 2)*

```
//Busca en un arreglo parcialmente lleno de enteros no negativos.
#include <iostream>
const int TAM_DECLARADO = 20;

void llenar_arreglo(int a[], int tamanio, int& elem_usados);
//Precondición: tamanio es el tamaño declarado del arreglo a.
//Postcondición: elem_usados es el número de valores guardados en a.
//a[0] a a[elem_usados-1] se han llenado con
//enteros no negativos leídos del teclado.

int buscar(const int a[], int elem_usados, int buscado);
//Precondición: elem_usados es <= el tamaño declarado de a.
//También a[0] a a[elem_usados-1] tienen valores
//Devuelve el primer índice tal que a[indice] == buscado,
//siempre que exista ese índice; si no, devuelve -1.

int main()
{
    using namespace std;
    int arr[TAM_DECLARADO], tamanio_lista, buscado;

    llenar_arreglo(arr, TAM_DECLARADO, tamanio_lista);

    char respuesta;
    int resultado;
    do
    {
        cout << "Escriba el numero que desea buscar: ";
        cin >> buscado;

        resultado = buscar(arr, tamanio_lista, buscado);
        if (resultado == -1)
            cout << buscado << " no esta en la lista.\n";
        else
            cout << buscado << " esta guardado en la posicion "
                 << resultado << endl
                 << "(Recuerde: La primera posicion es 0.)\n";

        cout << "Buscar otra vez? (S/N, seguido de Intro): ";
        cin >> respuesta;
    }while ((respuesta != 'n') && (respuesta != 'N'));

    cout << "Fin del programa.\n";
    return 0;
}
```

CUADRO 10.10 *Búsqueda en un arreglo (parte 2 de 2)*

```
//Usa iostream:
void llenar_arreglo(int a[], int tamanio, int& elem_usados)
<El resto de la definición de llenar_arreglo se da en el cuadro 10.9.>

int buscar(const int a[], int elem_usados, int buscado)
{

    int indice = 0;
    bool hallado = false;
    while ((!hallado) && (indice < elem_usados))
         if (buscado == a[indice])
             hallado = true;
         else
             indice++;

    if (hallado)
         return indice;
    else
         return -1;
}
```

Diálogo de ejemplo

```
Escriba hasta 20 numeros enteros no negativos.
Marque el final de la lista con un numero negativo.
10 20 30 40 50 60 70 80 -1
Escriba el numero que desea buscar: 10
10 esta guardado en la posicion 0
(Recuerde: La primera posicion es 0, no 1.)
Buscar otra vez? (S/N, seguido de Intro): s
Escriba el numero que desea buscar: 40
40 esta guardado en la posicion 3
(Recuerde: La primera posicion es 0, no 1.)
Buscar otra vez? (S/N, seguido de Intro): s
Escriba el numero que desea buscar: 42
42  no esta en la lista.
Buscar otra vez? (S/N, seguido de Intro): n
Fin del programa.
```

buscado en el arreglo, siempre que dicho valor esté en el arreglo. Si el valor buscado no está en el arreglo, `buscar` devuelve `-1`. Examinemos la función `buscar` con un poco más de detalle.

La función `buscar` usa un ciclo *`while`* para examinar los elementos del arreglo uno tras otro y ver si alguno de ellos contiene el valor buscado. Usamos la variable `hallado` como bandera para indicar si encontramos o no el valor buscado. Si el valor buscado se encuentra en el arreglo, se asigna *`true`* a `hallado`, y esto termina el ciclo *`while`*.

EJEMPLO DE PROGRAMACIÓN
Ordenamiento de un arreglo

Una de las tareas de programación más comunes, y ciertamente la que más exhaustivamente se ha estudiado, es ordenar una lista de valores, como una lista de cifras de ventas que se deben ordenar de menor a mayor o de mayor a menor, o una lista de palabras que se deben ordenar alfabéticamente. En esta sección describiremos una función llamada `ordenar` que ordenará de menor a mayor un arreglo que se ha llenado parcialmente con números.

El procedimiento `ordenar` tiene un parámetro de arreglo `a`. El arreglo `a` se llenará parcialmente, por lo que hay un parámetro formal adicional llamado `elem_usados`, que indica cuántas posiciones del arreglo están ocupadas. Así pues, la declaración y la precondición de la función `ordenar` serán:

```
void ordenar(int a[], int elem_usados);
//Precondición: elem_usados <= tamaño declarado del arreglo a.
//Los elementos del arreglo a[0] hasta a[elem_usados - 1] tienen
//valores.
```

La función `ordenar` reacomoda los elementos del arreglo `a` de modo que después de la llamada a la función los elementos queden ordenados así:

```
a[0] ≤ a[1] ≤ a[2] ≤ ... ≤ a[elem_usados - 1]
```

El algoritmo que usaremos para ordenar se denomina *ordenamiento por selección*. Éste es uno de los algoritmos de ordenamiento más fáciles de entender.

Una forma de diseñar un algoritmo es apoyarse en la definición del problema. En este caso el problema es ordenar un arreglo de menor a mayor. Esto implica reacomodar los valores de modo que `a[0]` contenga el más pequeño, `a[1]` contenga el siguiente más pequeño, etcétera. Esa definición produce un bosquejo del algoritmo de **ordenamiento por selección**:

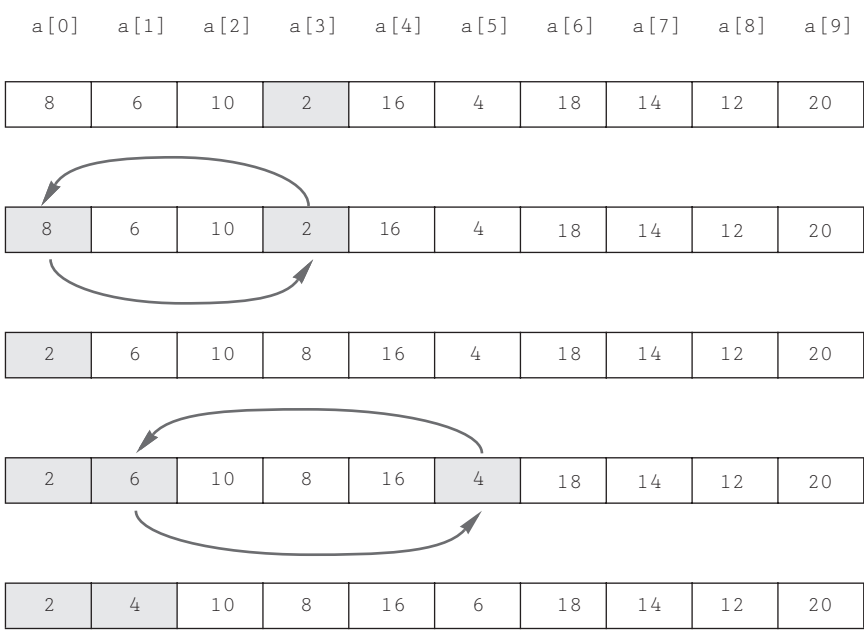
ordenamiento por selección

```
for (int indice = 0; indice < elem_usados; indice++)
    Colocar el indice-ésimo elemento más pequeño en a[indice]
```

Hay muchas formas de llevar a la práctica esta estrategia general. Una de ella podría usar dos arreglos y copiar los elementos de uno en el otro en orden de menor a mayor, pero deberá bastar un solo arreglo, lo cual sería más económico y adecuado. Por ello, la función `ordenar` sólo usa el arreglo que contiene los valores por ordenar. La función `ordenar` reacomoda los valores del arreglo `a` intercambiando pares de valores. Veamos un ejemplo concreto para entender cómo funciona el algoritmo.

Consideremos el arreglo que se muestra en el cuadro 10.11. El algoritmo coloca el valor más pequeño en `a[0]`. El valor más pequeño es el contenido en `a[3]`, así que el algoritmo intercambia los valores de `a[0]` y `a[3]`. Luego, el algoritmo busca el siguiente elemento más pequeño. El valor que está en `a[0]` ahora es el elemento más pequeño, por

CUADRO 10.11 *Ordenamiento por selección*



lo que el siguiente elemento más pequeño es el menor de los elementos restantes `a[1]`, `a[2]`, `a[3]`, ..., `a[9]`. En el ejemplo del cuadro 10.11, el siguiente elemento más pequeño está en `a[5]`, así que el algoritmo intercambia los valores de `a[1]` y `a[5]`. Esta ubicación del segundo elemento más pequeño se ilustra en la cuarta y la quinta representaciones del arreglo en el cuadro 10.11. El algoritmo acomoda después el tercer elemento más pequeño, y así sucesivamente.

Al continuar el ordenamiento, los primeros elementos del arreglo van recibiendo los valores ordenados correctos. La porción ordenada del arreglo crece por adición, uno por uno, de elementos del extremo no ordenado del arreglo. Observe que el algoritmo no tiene que hacer nada con el valor que queda en la última variable indizada `a[9]`. La razón es que una vez que los demás elementos están en la posición correcta, `a[9]` por fuerza debe tener también el valor correcto. Después de todo, el valor correcto para `a[9]` es el valor más pequeño que falta mover, y el único valor que falta mover es el que ya está en `a[9]`.

En el cuadro 10.12 se muestra la definición de la función `ordenar`, incluida en un programa de demostración. `ordenar` usa la función `indice_del_menor` para encontrar el índice del elemento más pequeño de la porción no ordenada del arreglo, y luego realiza un intercambio para pasar este siguiente elemento más pequeño a la parte ordenada del arreglo.

`indice_del_menor`

CUADRO 10.12 *Ordenamiento de un arreglo (parte 1 de 3)*

```
//Prueba el procedimiento ordenar.
#include <iostream>

void llenar_arreglo(int a[], int tamanio, int& elem_usados);
//Precondición: tamanio es el tamaño declarado del arreglo a.
//Postcondición: elem_usados es el número de valores guardados en a.
//a[0] a a[elem_usados-1] se han llenado con
//enteros no negativos leídos del teclado.

void ordenar(int a[], int elem_usados);
//Precondición: elem_usados <= tamaño declarado del arreglo a.
//Los elementos del arreglo a[0] hasta a[elem_usados] tienen valores
//Postcondición: Los valores a[0] a a[elem_usados-1] se han
//reacomodado de modo que a[0] <= a[1] <= ... <= a[elem_usados-1].

void intercambiar_valores(int& v1, int& v2);
//Intercambia los valores de v1 y v2.

int indice_del_menor(const int a[], int indice_inicio, int elem_usados);
//Precondición: 0 <= indice_inicio < elem_usados. Los elementos
//del arreglo a los que se hace referencia tienen valores.
//Devuelve el índice i tal que a[i] es el menor de los valores
//a[indice_inicio], a[indice_inicio + 1], ..., a[elem_usados-1].

int main()
{
    using namespace std;
    cout << "Este programa ordena numeros de menor a mayor.\n";

    int arreglo_muestra[10], elem_usados;
    llenar_arreglo(arreglo_muestra, 10, elem_usados);
    ordenar(arreglo_muestra, elem_usados);

    cout << "De menor a mayor los numeros son:\n";
    for (int indice = 0; indice < elem_usados; indice++)
        cout << arreglo_muestra[indice] << " ";
    cout << endl;

    return 0;
}
//Usa iostream:
void llenar_arreglo(int a[], int tamanio, int& elem_usados)
```

<El resto de la definición de `llenar_arreglo` se da en el cuadro 10.9.>

CUADRO 10.12 *Ordenamiento de un arreglo (parte 2 de 3)*

```
void ordenar(int a[], int elem_usados)
{
    int indice_del_siguiente_menor;
    for (int indice = 0; indice < elem_usados - 1; indice++)
    {//Colocar el valor correcto en a[indice]:
           indice_del_siguiente_menor =
                                indice_del_menor(a, indice, elem_usados);
           intercambiar_valores(a[indice], a[indice_del_siguiente_menor]);
           //a[0] <= a[1] <= ... <= a[indice] son los más pequeños de los
           //elementos originales del arreglo. Los demás elementos están en las
           //posiciones restantes.
    }
}

void intercambiar_valores(int& v1, int& v2)
{
    int temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}

int indice_del_menor(const int a[], int indice_inicio, int elem_usados)
{
    int min = a[indice_inicio],
        indice_de_min = indice_inicio;
    for (int indice = indice_inicio + 1; indice < elem_usados; indice++)
        if (a[indice] < min)
        {
             min = a[indice];
            indice_de_min = indice;
            //min es el menor de a[indice_inicio] a a[indice]
        }
    return indice_de_min;
}
```

CUADRO 10.12 *Ordenamiento de un arreglo (parte 3 de 3)*

Diálogo de ejemplo

```
Este programa ordena numeros de menor a mayor.
Escriba hasta 10 numeros enteros no negativos.
Marque el final de la lista con un numero negativo.
80 30 50 70 60 90 20 30 40 -1
De menor a mayor los numeros son:
20 30 30 40 50 60 70 80 90
```

intercambiar_valores

La función `intercambiar_valores`, que se muestra en el cuadro 10.12, sirve para intercambiar los valores de variables indizadas. Por ejemplo, la siguiente llamada intercambia los valores de `a[0]` y `a[3]`:

```
intercambiar_valores(a[0], a[3]);
```

La función `intercambiar_valores` se explicó en el capítulo 4.

Ejercicios de AUTOEVALUACIÓN

17. Escriba un programa que lea hasta 10 enteros no negativos y los coloque en un arreglo llamado `arr_numeros`, y luego escriba los enteros en la pantalla. No es necesario usar funciones para este ejercicio. Se trata de un programa de juguete y puede ser mínimo.

18. Escriba un programa que lea hasta 10 letras, las coloque en un arreglo y luego las escriba en la pantalla en orden inverso. Por ejemplo, si la entrada es

    `abcd.`

    la salida deberá ser

    `dcba`

    Use un punto como valor centinela para marcar el fin de las entradas. Llame al arreglo `caja_de_letras`. No es necesario usar funciones para este ejercicio. Se trata de un programa de juguete y puede ser mínimo.

19. A continuación se da una declaración de otra versión de la función `buscar` que definimos en el cuadro 10.12. Para usar esta versión de la función `buscar` necesitaríamos modificar un poco el programa, pero para este ejercicio lo único que se pide es escribir la definición de la función.

```
bool buscar(const int a[], int elem_usados,
                                int buscado, int& donde);
//Precondición: elem_usados es <= el tamaño declarado del
//arreglo a. También, a[0] hasta a[elem_usados-1] tienen valores.
//Postcondición: Si buscado es uno de los elementos a[0] hasta
//a[elem_usados-1], esta función devuelve true y establece el
//valor de donde de manera que a[donde] == buscado; de lo
//contrario la función devuelve false y el valor de
//donde no cambia.
```

10.4 Arreglos y clases

Podemos combinar arreglos, estructuras y clases para formar tipos intrincadamente estructurados como arreglos de estructuras, arreglos de clases y clases con arreglos como variables miembro. En esta sección trataremos unos cuantos ejemplos sencillos para darle una idea de las posibilidades.

Arreglos de clases

El tipo base de un arreglo puede ser cualquier tipo, incluso tipos definidos por nosotros, como los de estructuras y clases. Si queremos que cada elemento de un arreglo contenga cosas con diferentes tipos, podemos hacer que el arreglo sea un arreglo de estructuras. Por ejemplo, supongamos que queremos un arreglo para guardar 10 puntos datos meteorológicos, donde cada punto dato es una velocidad y una dirección del viento (norte, sur, este u oeste). Podríamos usar la siguiente definición de tipo y declaración de arreglo:

```
struct InfoViento
{
     double velocidad; //en kilómetros por hora
     char direccion;  //'N', 'S', 'E' u 'O'
};

InfoViento punto_dato[10];
```

Para llenar el arreglo `punto_dato`, podríamos usar el siguiente ciclo *for*:

```
int i;
for (i = 0; i < 10; i++)
{
    cout << "Escriba la velocidad para el punto dato "
         << i << " :";
    cin >> punto_dato[i].velocidad;
    cout << "Escriba la direccion para ese punto dato"
         << " (N, S, E u O): ";
    cin >> punto_dato[i].direccion;
}
```

La forma de leer una expresión como `punto_dato[i].velocidad` es de izquierda a derecha y con mucho cuidado. En primer lugar, `punto_dato` es un arreglo, así que `punto_dato[i]`

es la i-ésima variable indizada de ese arreglo. Las variables indizadas de ese arreglo son de tipo InfoViento, que es una estructura con dos variables miembro llamadas velocidad y direccion. Entonces, punto_dato[i].velocidad es la variable miembro llamada velocidad del i-ésimo elemento del arreglo. En términos menos formales, punto_dato[i].velocidad es la velocidad del viento para el i-ésimo punto dato. Así mismo, punto_dato[i].direccion es la dirección del viento para el i-ésimo punto dato.

Los 10 puntos dato del arreglo punto_dato se pueden desplegar en la pantalla con el siguiente ciclo *for*:

```
for (i = 0; i < 10; i++)
    cout << "Punto dato de viento numero " << i << ": \n"
         << punto_dato[i].velocidad
         << " kilometros por hora\n"
         << "direccion " << punto_dato[i].direccion
         << endl;
```

El cuadro 10.13. contiene el archivo de interfaz de una clase llamada Dinero. Los objetos de la clase Dinero sirven para representar cantidades de dinero en pesos. Las definiciones de las funciones miembro, operaciones miembro y funciones friend de esta clase se pueden encontrar en los cuadros del 8.3 al 8.8 y en la respuesta al ejercicio de autoevaluación 13 del capítulo 8. Estas definiciones deben reunirse en un archivo de implementación. Sin embargo, no mostraremos el archivo de implementación porque lo único que necesitamos saber para usar la clase Dinero se da en el archivo de interfaz.

Podemos tener arreglos cuyo tipo base sea el tipo Dinero. En el cuadro 10.14 se da un ejemplo sencillo. Ese programa lee una lista de cinco cantidades de dinero y calcula la diferencia entre cada cantidad y la más grande de las cinco cantidades. Observe que un arreglo cuyo tipo base es una clase se trata básicamente igual que cualquier otro arreglo. De hecho, el programa del cuadro 10.14 es muy similar al del cuadro 10.1, excepto que en el cuadro 10.14 el tipo base es una clase.

llamada a constructor

Cuando se declara un arreglo de clases, se invoca el constructor predeterminado para inicializar las variables indizadas, por lo que es importante tener un constructor predeterminado para cualquier clase que sea el tipo base de un arreglo.

Un arreglo de clases se manipula igual que un arreglo con un tipo base simple como *int* o *double*. Por ejemplo, la diferencia entre cada cantidad y la cantidad más grande se almacena en un arreglo llamado diferencia así:

```
Dinero diferencia[5];
for (i = 0; i < 5; i++)
    diferencia[i] = max - monto[i];
```

Ejercicios de AUTOEVALUACIÓN

20. Dé una definición de tipo para una estructura llamada Marcador que tenga dos variables miembro llamadas equipo_de_casa y oponente. Ambas variables miembro son de tipo int. Declare un arreglo llamado partido que sea un arreglo de 10 elementos de tipo Marcador. El arreglo partido podría servir para registrar los marcadores de 10 partidos de un equipo deportivo.

CUADRO 10.13 *Archivo de encabezado para la clase* Dinero *(parte 1 de 2)*

```
//Éste es el archivo de encabezado dinero.h; es la interfaz de la clase Dinero.
//Los valores de este tipo son cantidades de dinero en dólares.
#ifndef DINERO_H
#define DINERO_H
#include <iostream>
using namespace std;
namespace savitchdinero
{
    class Dinero
    {
    public:
       friend Dinero operator +(const Dinero& monto1, const Dinero& monto2);
       //Devuelve la suma de los valores de monto1 y monto2.

       friend Dinero operator -(const Dinero& monto1, const Dinero& monto2);
       //Devuelve monto1 menos monto2.

       friend Dinero operator -(const Dinero& monto);
       //Devuelve el negativo del valor de monto.

       friend bool operator ==(const Dinero& monto1, const Dinero& monto2);
       //Devuelve true si monto1 y monto2 tienen el mismo valor; si no,
       //devuelve false.

       friend bool operator < (const Dinero& monto1, const Dinero& monto2);
       //Devuelve true si monto1 es menor que monto2; si no, devuelve false.

       Dinero(long dolares, int centavos);
       //Inicializa el objeto de modo que su valor representa un monto
       //con los dólares y centavos dados por los argumentos. Si el monto
       //es negativo, dólares y centavos deben ser ambos negativos.

       Dinero(long dolares);
       //Inicializa el objeto de modo que su valor represente $dolares.00.

       Dinero();
       //Inicializa el objeto de modo que su valor represente $0.00

       double obtener_valor() const;
       //Devuelve el dinero que indica la porción de datos
       //del objeto invocador.

       friend istream& operator >>(istream& entra, Dinero& monto);
       //Sobrecarga el operador >> para poder introducir valores de tipo Dinero.
       //La notación para introducir montos negativos es como en -$100.00.
       //Precondición: Si entra es un flujo de archivo de entrada, entonces entra
       //ya se conectó a un archivo.
```

CUADRO 10.13 *Archivo de encabezado para la clase* `Dinero` *(parte 2 de 2)*

```
    friend ostream& operator <<(ostream& sale, const Dinero& monto);
    //Sobrecarga el operador << para enviar a la salida valores de tipo Dinero.
    //Antepone a cada valor de salida de tipo Dinero un signo de dólares.
    //Precondición: Si sale es un flujo de archivo de salida, entonces sale
    //ya se conectó a un archivo.
  private:
    long todo_centavos;
  };
}//namespace savitchdinero
#endif  //DINERO_H
```

21. Escriba un programa que lea cinco cantidades de dinero, duplique cada cantidad, y luego escriba los valores doblados en la pantalla. Use un arreglo con `Dinero` como tipo base. *Sugerencia*: Use el cuadro 10.14 como guía, pero este programa será más sencillo que el del cuadro 10.14.

Arreglos como miembros de clases

Podemos tener una estructura o clase que tenga un arreglo como variable miembro. Por ejemplo, suponga que es un nadador de velocidad y quiere un programa para registrar sus tiempos de práctica en diversas distancias. Podemos usar la estructura `mi_mejor` (del tipo `Datos` que se da a continuación) para registrar una distancia (en metros) y los tiempos (en segundos) para cada una de diez pruebas de práctica nadando esa distancia:

```
struct Datos
{
     double tiempo[10];
     int distancia;
};

Datos mi_mejor;
```

La estructura `mi_mejor` que acabamos de declarar tiene dos variables miembro: una, llamada `distancia`, es una variable de tipo *`int`* (para registrar una distancia); la otra, llamada `tiempo`, es un arreglo de diez valores de tipo *`double`* (para guardar los tiempos hechos en diez pruebas de práctica a la distancia especificada). Para hacer la distancia igual a `20` (metros), podemos usar lo siguiente:

```
mi_mejor.distancia = 20;
```

Podemos asignar valores a los 10 elementos del arreglo con el teclado así:

```
cout << "Escribe diez tiempos (en segundos):\n";
for (int i = 0; int < 10; i++)
    cin >> mi_mejor.tiempo[i];
```

CUADRO 10.14 *Programa que usa un arreglo de objetos (parte 1 de 2)*

```
//Lee cinco cantidades de dinero y muestra en cuánto
//cada cantidad difiere de la cantidad más grande.
#include <iostream>
#include "dinero.h"

int main()
{
    using namespace std;
    using namespace savitchdinero;
    Dinero monto[5], max;
    int i;

    cout << "Teclee 5 cantidades de dinero:\n";
    cin >> monto[0];
    max = monto[0];
    for (i = 1; i < 5; i++)
    {
        cin >> monto[i];
        if (max < monto[i])
            max = monto[i];
        //max es el mayor de monto[0],..., monto[i].
    }
    Dinero diferencia[5];
    for (i = 1; i < 5; i++)
        diferencia[i] = max - monto[i];

    cout << "La cantidad mas alta es " << max << endl;
    cout << "Las cantidades y sus\n "
         << "diferencias respecto a la mas grande son:\n";
    for (i = 0; i < 5; i++)
    {
        cout << monto[i] << " difiere en "
             << diferencia[i] << endl;
    }
    return 0;
}
```

CUADRO 10.14 *Programa que usa un arreglo de objetos (parte 2 de 2)*

Diálogo de ejemplo

```
Escriba 5 cantidades de dinero:
$5.00 $10.00 $19.99 $20.00 $12.79
La cantidad mas alta es $20.00
Las cantidades y sus
diferencias respecto a la mas grande son:
$5.00 difiere en $15.00
$10.00 difiere en $10.00
$19.99 difiere en $0.01
$20.00 difiere en $0.00
$12.79 difiere en $7.21
```

La expresión `mi_mejor.tiempo[i]` se lee de izquierda a derecha: `mi_mejor` es una estructura. `mi_mejor.tiempo` es la variable miembro llamada `tiempo`. Puesto que `mi_mejor.tiempo` es un arreglo, tiene sentido añadir un índice. Así, la expresión `mi_mejor.tiempo[i]` es la `i`-ésima variable indizada del arreglo `mi_mejor.tiempo`. Si usamos un tipo de clase en lugar de uno de estructura, podemos efectuar todas nuestras manipulaciones de arreglos con funciones miembro y evitar semejantes expresiones confusas. Esto se ilustra en el siguiente ejemplo de programación.

EJEMPLO DE PROGRAMACIÓN

Una clase para un arreglo parcialmente lleno

Los cuadros 10.15 y 10.16 muestran la definición de una clase llamada `ListaTemperaturas`, cuyos objetos son listas de temperaturas. Podríamos usar un objeto del tipo `ListaTemperaturas` en un programa que realiza análisis climatológicos. La lista de temperaturas se guarda en la variable miembro `lista`, que es un arreglo. Puesto que este arreglo normalmente sólo estará parcialmente lleno, se usa una segunda variable miembro, `tamanio`, para saber qué tanto del arreglo se usa. El valor de `tamanio` es el número de variables indizadas del arreglo `lista` que se están usando para almacenar valores.

En un programa que usa esta clase, el archivo de encabezado se debe mencionar en una directriz `include`, igual que cualquier otra clase que se coloque en un archivo aparte. Así pues, cualquier programa que use la clase `ListaTemperaturas` deberá contener la siguiente directriz `include`:

```
#include "listemp.h"
```

Los objetos de tipo `ListaTemperaturas` se declaran como los de cualquier otro tipo. Por ejemplo, lo que sigue declara `mis_datos` como un objeto de tipo `ListaTemperaturas`:

```
ListaTemperaturas mis_datos;
```

CUADRO 10.15 *Interfaz para una clase con un miembro arreglo*

```
//Éste es el archivo de encabezado listemp.h. Es la interfaz de la clase
//ListaTemperaturas. Los valores de este tipo son listas de temperaturas Fahrenheit.

#ifndef LISTEMP_H
#define LISTEMP_H
#include <iostream>
using namespace std;
namespace savitchlistat
{
  const int TAM_MAX_LISTA = 50;

  class ListaTemperaturas
  {
  public:
     ListaTemperaturas();
     //Inicializa el objeto con una lista vacía.

     void agregar_temperatura(double temperatura);
     //Precondición: La lista no está llena.
     //Postcondición: La temperatura se añadió a la lista.

     bool llena() const;
     //Devuelve true si la lista está llena, false en caso contrario.

     friend ostream& operator <<(ostream& sale,
                                 const ListaTemperaturas& el_objeto);
     //Sobrecarga el operador << para enviar a la salida valores de
     //tipo ListaTemperaturas. Se exhibe una temperatura por línea.
     //Precondición: Si sale es un flujo de archivo de salida, entonces
     //ya se conectó a un archivo.

    private:
      double lista[TAM_MAX_LISTA]; //de temperaturas Fahrenheit
      int tamanio; //número de posiciones del arreglo ocupadas
    };
}//namespace savitchlistat
#endif  //LISTEMP_H
```

CUADRO 10.16 *Implementación de una clase con un miembro arreglo*

```
//Éste es el archivo de implementación: listemp.cpp para la clase ListaTemperaturas.
//La interfaz de esta clase está en el archivo listemp.h.
#include <iostream>
#include <cstdlib>
#include "listemp.h"
using namespace std;
namespace savitchlistat
{
    ListaTemperaturas::ListaTemperaturas() : tamanio(0)
    {
          //cuerpo intencionalmente vacío
    }

    void ListaTemperaturas::agregar_temperatura(double temperatura)
    {//Usa iostream y cstdlib:
          if ( llena() )
          {
              cout << "Error: la lista ya esta llena.\n";
              exit(1);
          }
          else
          {
              lista[tamanio] = temperatura;
              tamanio = tamanio + 1;
          }
    }

    bool ListaTemperaturas::llena() const
    {
          return (tamanio == TAM_MAX_LISTA);
    }

    //Usa iostream:
    ostream& operator <<(ostream& sale, const ListaTemperaturas& el_objeto)
    {
          for (int i = 0; i < el_objeto.tamanio; i++)
              sale << el_objeto.lista[i] << " F\n";
          return sale;
    }
}//namespace savitchlistat
```

Esta declaración invoca el constructor predeterminado con el objeto nuevo `mis_datos`, y el objeto `mis_datos` se inicializa de modo que la variable miembro `tamanio` tenga el valor `0`, lo que indica que la lista está vacía.

Una vez declarado un objeto como `mis_datos`, podemos añadir un elemento a la lista de temperaturas (es decir, al arreglo miembro `lista`) con una llamada a la función miembro `agregar_temperatura` como sigue:

```
mis_datos.agregar_temperatura(77);
```

De hecho, esta es la única forma en que podemos añadir una temperatura a la lista `mis_datos`, pues el arreglo `lista` es una variable miembro privada. Tome nota de que cuando añadimos un elemento con una llamada a la función miembro `agregar_temperatura`, la función primero determina si el arreglo `lista` está lleno, y sólo añade el valor si el arreglo no está lleno.

La clase `ListaTemperaturas` es muy especializada. Las únicas cosas que podemos hacer con un objeto de la clase `ListaTemperaturas` son inicializar la lista de modo que esté vacía, añadir elementos a la lista, verificar si la lista está llena y enviar la lista a la salida. Para enviar a la pantalla las temperaturas almacenadas en el objeto `mis_datos` (que ya declaramos), la llamada sería:

```
cout << mis_datos;
```

Con la clase `ListaTemperaturas` no podemos eliminar una temperatura de la lista (arreglo) de temperaturas, pero sí podemos borrar toda la lista y volver a comenzar con una lista vacía invocando el constructor predeterminado así:

```
mis_datos = ListaTemperaturas();
```

El tipo `ListaTemperaturas` casi no usa propiedades de las temperaturas. Podríamos definir una clase similar para listas de presiones o listas de distancias o listas de cualesquier otros datos que se expresen como valores de tipo *`double`*. Para ahorrarnos el trabajo de definir todas estas clases distintas podríamos definir una sola clase que representa una lista arbitraria de valores de tipo *`double`* sin especificar qué representan los valores. Pediremos al lector que defina una clase así en el proyecto de programación 11.

Ejercicios de AUTOEVALUACIÓN

22. Modifique la clase `ListaTemperaturas` que se da en los cuadros 10.15 y 10.16 añadiéndole una función miembro `obtener_tamanio`, que no recibe argumentos y devuelve el número de temperaturas que hay en la lista.

23. Modifique el tipo `ListaTemperaturas` que se da en los cuadros 10.15 y 10.16 añadiéndole una función miembro llamada `obtener_temperatura`, que recibe un argumento *`int`* que es un entero mayor o igual que `0` y menor que TAM_MAX_LISTA. La función devuelve un valor de tipo *`double`*, que es la temperatura que está en esa posición de la lista. Así pues, con un argumento de `0` `obtener_temperatura` devuelve la primera temperatura; con un argumento de `1` `obtener_temperatura` devuelve la segunda temperatura, etcétera. Suponga que la función no se invocará con un argumento que especifique una posición de la lista que de momento no contiene una temperatura.

10.5 Arreglos multidimensionales

Dos índices son mejores que uno.

Encontrado en la pared de un baño del departamento de ciencias de la computación

C++ le permite declarar arreglos con más de un índice. En esta sección describimos estos arreglos multidimensionales.

Fundamentos de arreglos multidimensionales

declaraciones de arreglos – variables indizadas

A veces es útil tener un arreglo con más de un índice, y esto se permite en C++. Lo que sigue declara un arreglo de caracteres llamado `pagina`. El arreglo tiene dos índices: el primero va desde `0` hasta `29` y el segundo, desde `0` hasta `99`.

```
char pagina[30][100];
```

Las variables indizadas de este arreglo tienen dos índices. Por ejemplo, `pagina[0][0]`, `pagina[15][32]` y `pagina[29][99]` son tres de las variables indizadas para este arreglo. Observe que cada índice se debe encerrar en sus propios corchetes. Como en el caso de los arreglos unidimensionales que ya vimos, cada variable indizada de un arreglo multidimensional es una variable del tipo base.

Un arreglo puede tener cualquier cantidad de índices, pero el número más común de índices es dos. Un arreglo bidimensional se puede visualizar como un despliegue de dos dimensiones con el primer índice dice la fila y con el segundo, la columna. Por ejemplo las variables indizadas del arreglo bidimensional `pagina` se pueden visualizar de la siguiente manera:

```
pagina[0][0], pagina[0][1], ..., pagina[0][99]
pagina[1][0], pagina[1][1], ..., pagina[1][99]
pagina[2][0], pagina[2][1], ..., pagina[2][99]
          .
          .
          .
pagina[29][0], pagina[29][1], ..., pagina[29][99]
```

Podríamos usar el arreglo `pagina` para almacenar todos los caracteres de una página de texto que tiene 30 líneas (numeradas de la 0 a la 29) y 100 caracteres en cada línea (numerados del 0 al 99).

Un arreglo multidimensional es un arreglo de arreglos

En C++ un arreglo bidimensional, como `pagina`, en realidad es un arreglo de arreglos. El arreglo `pagina` es realmente un arreglo unidimensional de tamaño 30, cuyo tipo base es un arreglo unidimensional de caracteres de tamaño 100. Normalmente, esto no deberá preocuparnos, y podremos actuar como si `pagina` fuera en realidad un arreglo con dos índices (en lugar de un arreglo de arreglos, que es más difícil de manejar). Sin embargo, hay por lo menos una situación en la que un arreglo bidimensional se ve como un arreglo de arreglos: cuando tenemos una función con un parámetro de arreglo para un arreglo bidimensional, lo cual se discute en la siguiente subsección.

Parámetros de arreglos multidimensionales

La siguiente declaración de un arreglo bidimensional está actualmente declarando un arreglo unidimensional del tamaño 30, cuyo tipo base es un arreglo unidimensional de caracteres del tamaño 100

```
char pagina [30][100];
```

Declaración de arreglos multidimensionales

Sintaxis:

Tipo Nombre_del_Arreglo[*Tamanio_Dim_1*][*Tamanio_Dim_2*] ... [*Tamanio_Dim_Ultimo*];

Ejemplos:

```
char pagina[30][100];
int matriz[2][3];
double imagen_tridimensional[10][20][30];
```

Una declaración de arreglo, de la forma que se muestra aquí, define una variable indizada para cada combinación de índices del arreglo. Por ejemplo, la segunda de las declaraciones anteriores define las siguientes seis variables indizadas del arreglo `matriz`:

```
matriz[0][0], matriz[0][1], matriz[0][2],
matriz[1][0], matriz[1][1], matriz[1][2]
```

parámetros de arreglo multidimensional

Ver a un arreglo bidimensional como un arreglo de arreglos le ayudará a entender la manera en que C++ maneja los parámetros para un arreglo multidimensional.

Por ejemplo, la siguiente función recibe un arreglo, como `pagina`, y lo despliega en la pantalla:

```
void mostrar_pagina(const char p[][100], int tam_dimension_1)
{
     for (int indice1 = 0; indice1 < tam_dimension_1; indice1++)
     {//Se escribe una línea:
          for (int indice2 = 0; indice2 < 100; indice2++)
              cout << p[indice1][indice2];
          cout << endl;
     }
}
```

Observe que con un parámetro de arreglo bidimensional no se da el tamaño de la primera dimensión, por lo que es preciso incluir un parámetro `int` para dar el tamaño de esta primera dimensión. (Como con los arreglos ordinarios, el compilador nos permite especificar la primera dimensión colocando un número dentro del primer par de corchetes. Sin embargo, tal número es sólo un comentario; el compilador hace caso omiso de él.) El tamaño de la segunda dimensión (y de todas las demás dimensiones si hay más de dos) se da después del parámetro de arreglo, como se muestra para el parámetro

```
const char p[][100]
```

Parámetros de arreglo multidimensional

Cuando damos un parámetro de arreglo multidimensional en un encabezado o declaración de función, no damos el tamaño de la primera dimensión, pero debemos dar los tamaños de las dimensiones restantes entre corchetes. Puesto que no se da el tamaño de la primera dimensión, por lo regular se necesita un parámetro adicional de tipo `int` que da el tamaño de esta primera dimensión. He aquí un ejemplo de declaración de función con un parámetro de arreglo bidimensional `p`:

```
void obtener_pagina(char p[][100], int tamanio_dimension_1);
```

Si nos damos cuenta de que un arreglo multidimensional es un arreglo de arreglos, la regla resulta lógica. Puesto que el parámetro de arreglo multidimensional

```
const char p[][100]
```

es un parámetro para un arreglo de arreglos, la primera dimensión es en realidad el índice del arreglo y se trata igual que el índice de un arreglo unidimensional ordinario. La segunda dimensión es parte de la descripción del tipo base, que es un arreglo de caracteres de tamaño `100`.

EJEMPLO DE PROGRAMACIÓN
Programa calificador bidimensional

`calif`

El cuadro 10.17 contiene un programa que usa un arreglo bidimensional llamado `calif` para almacenar y luego mostrar las calificaciones de un grupo escolar pequeño. El grupo tiene cuatro estudiantes y sustenta tres exámenes. El cuadro 10.17 ilustra la forma de usar el arreglo `calif` para almacenar datos. El primer índice del arreglo sirve para designar un estudiante, y el segundo, para designar un examen. Puesto que los estudiantes y exámenes se numeran a partir de 1, no de 0, debemos restar uno al número de estudiantes y también al número de examenes para obtener la variable indizada que contiene una calificación de examen específica. Por ejemplo, la calificación que obtuvo el estudiante número 4 en el examen número 1 se registra en `calif[3][0]`.

`prom_est` **y** `prom_exam`

Nuestro programa también usa dos arreglos unidimensionales ordinarios. El arreglo `prom_est` sirve para registrar la calificación promedio de cada uno de los estudiantes. Por ejemplo, el programa hará `prom_est[0]` igual al promedio de las calificaciones de examen obtenidas por el estudiante 1, `prom_est[1]` será igual al promedio de las calificaciones obtenidas por el estudiante 2, etcétera. Usaremos el arreglo `prom_exam` para registrar la calificación promedio obtenida por el grupo en cada examen. Por ejemplo, el programa hará `prom_exam[0]` igual al promedio de las calificaciones de todos los estudiantes en el examen 1, `prom_exam[1]` será igual a la calificación promedio para el examen 2, etcétera. El cuadro 10.19 ilustra la relación entre los arreglos `calif`, `prom_est` y `prom_exam`. En ese cuadro hemos mostrado algunos datos de muestra para el arreglo `calif`. Estos datos, a su vez, determinan los valores que el programa almacena en `prom_est` y `prom_exam`.

CUADRO 10.17 *Arreglo bidimensional (parte 1 de 4)*

```
//Lee calificaciones de examen de cada estudiante y las guarda en el arreglo bidimensional
//calif. (No se muestra el código de entrada.) Calcula la calificación promedio
//para cada estudiante y para cada examen. Muestra las calificaciones y promedios.
#include <iostream>
#include <iomanip>
const int NUM_ESTUDIANTES = 4, NUM_EXAMENES = 3;
```

CUADRO 10.17 *Arreglo bidimensional (parte 2 de 4)*

```
void calcular_prom_est(const int calif[][NUM_EXAMENES], double prom_est[]);
//Precondición: Las constantes globales NUM_ESTUDIANTES y NUM_EXAMENES son las
//dimensiones del arreglo calif. Cada variable indizada calif[num_est-1,
//num_exam-1] contiene la calificación del estudiante num_est en el examen num_exam.
//Postcondición: Cada prom_est[num_est-1] contiene el promedio para el estudiante num_est.

void calcular_prom_exam(const int calif[][NUM_EXAMENES], double prom_exam[]);
//Precondición: Las constantes globales NUM_ESTUDIANTES y NUM_EXAMENES son las
//dimensiones del arreglo calif. Cada variable indizada calif[num_est-1,
//num_exam-1] contiene la calificación del estudiante num_est en el examen num_exam.
//Postcondición: Cada prom_exam[num_exam-1] contiene el promedio para el
//examen número num_exam.

void mostrar(const int calif[][NUM_EXAMENES],
                              const double prom_est[], const double prom_exam[]);
//Precondición: Las constantes globales NUM_ESTUDIANTES y NUM_EXAMENES son las
//dimensiones del arreglo calif. Cada variable indizada calif[num_est-1,
//num_exam-1] contiene la calificación del estudiante num_est en el examen num_exam.
//Cada prom_est[num_est-1] contiene el promedio para el estudiante num_est.
//Cada prom_exam[num_exam-1] contiene el promedio para el examen num_exam.
//Postcondición: Se enviaron a la salida todos los datos de calif, prom_est y prom_exam.

int main()
{
    using namespace std;
    int calif[NUM_ESTUDIANTES][NUM_EXAMENES];
    double prom_est[NUM_ESTUDIANTES];
    double prom_exam[NUM_EXAMENES];
```

<Aquí va el código para llenar el arreglo calif, pero no se muestra.>

```
    calcular_prom_est(calif, prom_est);
    calcular_prom_exam(calif, prom_exam);
    mostrar(calif, prom_est, prom_exam);
    return 0;
}
```

CUADRO 10.17 *Arreglo bidimensional (parte 3 de 4)*

```
void calcular_prom_est(const int calif[][NUM_EXAMENES], double prom_est[])
{
    for (int num_est = 1; num_est <= NUM_ESTUDIANTES; num_est++)
    { //Procesar un num_est:
          double suma = 0;
          for (int num_exam = 1; num_exam <= NUM_EXAMENES; num_exam++)
               suma = suma + calif[num_est-1][num_exam-1];
          //suma contiene la sumatoria de las calificaciones del estudiante num_est.
          prom_est[num_est-1] = suma/NUM_EXAMENES;
          //El promedio para el estudiante num_est es el valor de prom_est[num_est-1]
    }
}


//Usa iostream y iomanip:
void calcular_prom_exam(const int calif[][NUM_EXAMENES], double prom_exam[])
{
    for (int num_exam = 1; num_exam <= NUM_EXAMENES; num_exam++)
    {//Procesar un examen (para todos los estudiantes):
          double suma = 0;
          for (int num_est = 1; num_est <= NUM_ESTUDIANTES; num_est++)
              suma = suma + calif[num_est-1][num_exam-1];
          //suma contiene la sumatoria de las calificaciones para el examen num_exam.
          prom_exam[num_exam-1] = suma/NUM_ESTUDIANTES;
          //El promedio para el examen num_exam es el valor de prom_exam[num_exam-1]
    }
}


//Usa iostream y iomanip:
void mostrar(const int calif[][NUM_EXAMENES]
                           const double prom_est[], const double prom_exam[])
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
```

CUADRO 10.17 *Arreglo bidimensional (parte 4 de 4)*

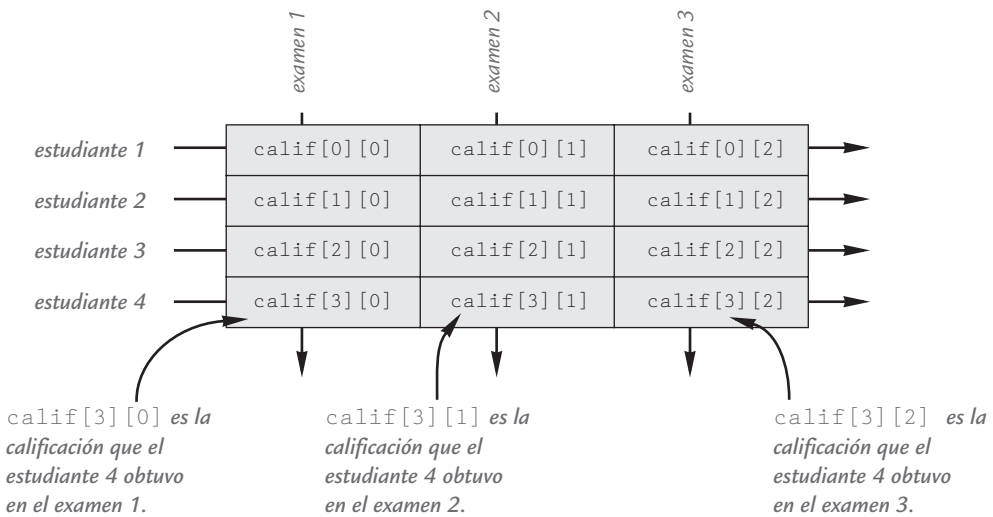
```
    cout << setw(10) << "Alumno "
         << setw(5) << "Prom"
         << setw(15) << "Examenes\n";
    for (int num_est = 1; num_est <= NUM_ESTUDIANTES; num_est++)
    {//Mostrar para un num_est:
         cout << setw(10) << num_est
              << setw(5) << prom_est[num_est-1] << " ";
         for (int num_exam = 1; num_exam <= NUM_EXAMENES; num_exam++)
             cout << setw(5) << calif[num_est-1][num_exam-1];
         cout << endl;
    }
    cout << "Promedios/exam= ";
    for (int num_exam = 1; num_exam <= NUM_EXAMENES; num_exam++)
         cout << setw(5) << prom_exam[num_exam-1];
         cout << endl;
}
```

Diálogo de ejemplo

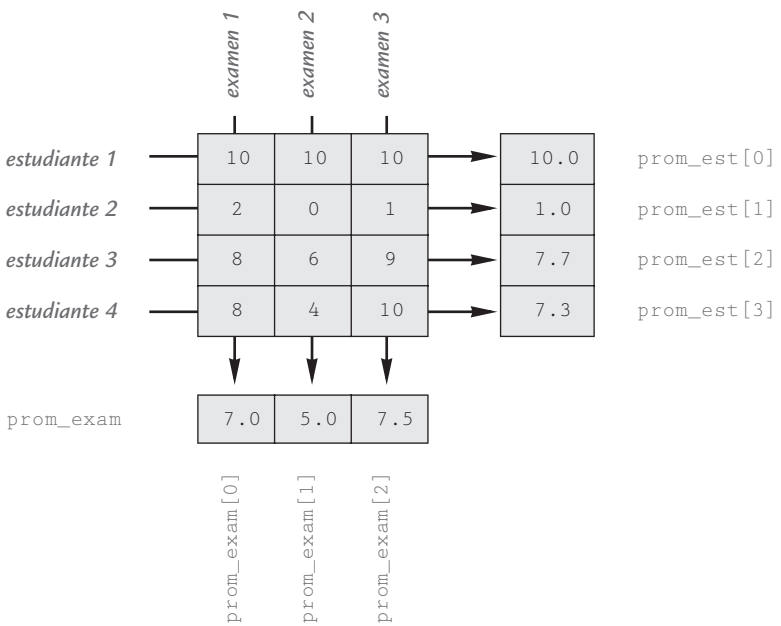
<No se muestra el diálogo para llenar el arreglo `calif`.>

```
Alumno:      Prom         Examenes
     1       10.0         10    10    10
     2        1.0          2     0     1
     3        7.7          8     6     9
     4        7.3          8     4    10
Promedios/exam =         7.0   5.0   7.5
```

CUADRO 10.18 *El arreglo bidimensional* `calif`



CUADRO 10.19 *El arreglo bidimensional* `calif` *(otra vista)*



El cuadro 10.19 también muestra estos valores, que el programa calcula para `prom_est` y `prom_exam`.

El programa completo para llenar el arreglo `calif` y luego calcular y mostrar tanto los promedios de los estudiantes como los promedios para cada examen se muestra en el cuadro 10.17. En ese programa hemos declarado las dimensiones de los arreglos como constantes globales. Puesto que todos los procedimientos son exclusivos para este programa y no podrían reutilizarse en otros, hemos usado tales constantes definidas globalmente en los cuerpos de los procedimientos, en lugar de incluir parámetros para el tamaño de las dimensiones de los arreglos. Puesto que el código que llena el arreglo es cosa de rutina, no se muestra en el cuadro.

RIESGO *Uso de comas entre arreglos indizados*

Observe que en el arreglo bidimensional del cuadro 10.17 escribimos una variable indizada utilizando dos pares de corchetes, es decir, calif[num_est−1][num_exam−1]. En algunos lenguajes de programación esto se debe escribir con un par de corchetes y comas, es decir: calif[num_est−1, num_exam−1]; esto es incorrecto en C++. Si utilizamos calif[num_est−1, num_exam−1] en C++ no obtendremos ningún mensaje de error pero su uso es incorrecto y provocará que su programa tenga un comportamiento inapropiado.

Ejercicios de AUTOEVALUACIÓN

24. ¿Qué salida produce el siguiente código?

```
int mi_arreglo[4][4], indice1, indice2;
for (indice1 = 0; indice1 < 4; indice1++)
    for (indice2 = 0; indice2 < 4; indice2++)
        mi_arreglo[indice1][indice2] = indice2;
for (indice1 = 0; indice1 < 4; indice1++)
{
    for (indice2 = 0; indice2 < 4; indice2++)
        cout << mi_arreglo[indice1][indice2] << " ";
    cout << endl;
}
```

25. Escriba un código que llene el arreglo `a` (que se declara enseguida) con números tecleados. Los números se introducirán cinco por línea, en cuatro líneas (aunque la solución no tiene que depender de cómo se dividen en líneas los números introducidos):

```
int a[4][5];
```

26. Escriba la definición de una función *void* llamada `eco` de manera que la siguiente llamada de función haga eco de las entradas descritas en el ejercicio de autoevaluación 25, y lo haga en el mismo formato que especificamos para la entrada (cuatro líneas de cinco números cada una):

```
eco(a, 4);
```

Resumen del capítulo

- Podemos utilizar un arreglo para almacenar y manipular un conjunto de datos, todos los cuales son del mismo tipo.
- Las variables indizadas de un arreglo se pueden usar como cualesquier otras variables del tipo base del arreglo.
- Un ciclo `for` es una buena forma de procesar los elementos de un arreglo, realizando alguna acción del programa con cada variable indizada.
- El error de programación más común que se comete al usar arreglos es tratar de acceder a un elemento inexistente del arreglo. Siempre revise la primera y última iteraciones de un ciclo que manipule un arreglo para asegurarse de que no use un índice no válido: demasiado pequeño o demasiado grande.
- Un parámetro formal de arreglo no es un parámetro de llamada por valor ni uno de llamada por referencia, sino un nuevo tipo de parámetro. Un parámetro de arreglo es similar a uno de llamada por referencia en cuanto a que cualquier cambio que se hace al parámetro formal en el cuerpo de la función se hace al argumento arreglo cuando se invoca la función.
- Las variables indizadas de un arreglo se almacenan una tras otra en la memoria de la computadora de modo que el arreglo ocupe una porción contigua de la memoria. Cuando el arreglo se pasa como argumento de una función, sólo se proporciona a la función invocadora la dirección de la primera variable indizada (la que tiene el índice `0`). Por tanto, una función con un parámetro de arreglo normalmente necesita otro parámetro formal de tipo `int` que dé el tamaño del arreglo.
- Al emplear un arreglo parcialmente lleno, un programa necesita una variable adicional de tipo `int` para saber qué tanto del arreglo está ocupado.
- Si queremos indicar al compilador que nuestra función no debe modificar un argumento arreglo, podemos insertar el modificador `const` antes del parámetro de arreglo correspondiente a ese argumento. Un parámetro de arreglo modificado con `const` es un **parámetro de arreglo constante**.
- El tipo base de un arreglo puede ser un tipo de estructura o de clase. Una estructura o una clase puede tener un arreglo como variable miembro.
- Si necesitamos un arreglo con más de un índice, podemos usar un arreglo multidimensional, que en realidad es un arreglo de arreglos.

Respuestas a los ejercicios de autoevaluación

1. La instrucción `int a[5];` es una declaración, donde `5` es el número de elementos del arreglo. La expresión `a[4]` es un acceso al arreglo definido por la instrucción anterior. El acceso es al elemento que tiene el índice `4`, o sea, al quinto (y último) elemento del arreglo.
2. a) `puntos`.
   b) `double`.
   c) `5`.
   d) `0` a `4`.
   e) Cualquiera de `puntos[0]`, `puntos[1]`, `puntos[2]`, `puntos[3]`, `puntos[4]`.

3. a) Un inicializador de más.

   b) Correcto. El tamaño del arreglo es 4.

   c) Correcto. El tamaño del arreglo es 4.

4. `abc`

5. `1.1 2.2 3.3`

   `1.1 3.3 3.3`

   (Recuerde que los índices comienzan con `0`, no con `1`.)

6. `0 2 4 6 8 10 12 14 16 18`

   `0 4 8 12 16`

7. Las variables indizadas de `arreglo_muestra` son `arreglo_muestra[0]` hasta `arreglo_muestra[9]`, pero este fragmento de código trata de llenar de `arreglo_muestra[1]` hasta `arreglo_muestra[10]`. El índice `10` en `arreglo_muestra[10]` está fuera de intervalo.

8. Hay un índice fuera de intervalo. Cuando `indice` es igual a `9`, `indice + 1` es igual a `10`, así que `a[indice + 1]`, que es lo mismo que `a[10]`, tiene un índice no válido. El ciclo debe terminar con una iteración menos. Para corregir el código, cambie la primera línea del ciclo *for* a:

   ```
   for (int indice = 0; indice < 9; indice++)
   ```

9. ```
   int i, a[20];
   cout << "Escriba 20 numeros:\n";
   for (i = 0; i < 20; i++)
   cin >> a[i];
   ```

10. El arreglo consumirá 14 bytes de memoria. La dirección de la variable indizada `tu_arreglo[3]` es 1006.

11. Las siguientes llamadas de función son aceptables:

    ```
    triplicador(numero);
    triplicador(a[2]);
    triplicador(a[numero]);
    ```

    Las siguientes llamadas de función son incorrectas.

    ```
    triplicador(a[3]);
    triplicador(a);
    ```

    La primera tiene un índice no válido; la segunda no tiene una expresión de índice. No podemos usar un arreglo entero como argumento de `triplicador`, como en la segunda de las siguientes llamadas. En la sección "Arreglos enteros como argumentos de funciones" se estudia una situación distinta en la que se puede usar todo un arreglo como argumento.

12. El ciclo procesa las variables indizadas `b[1]` a `b[5]`, pero `5` es un índice no válido para el arreglo `b`. Los índices son `0`, `1`, `2`, `3` y `4`. La versión correcta del código es:

    ```
    int b[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++)
       triplicador(b[i]);
    ```
    ```

13.
```
void mas_uno(int a[], int tamanio)
//Precondición: tamanio es el tamaño declarado del arreglo a.
//a[0] a a[tamanio21] han recibido valores.
//Postcondición: a[indice] se ha incrementado en 1
//para todas las variables indizadas de a.
{
for (int indice = 0; indice < tamanio; indice++)
     a[indice] = a[indice] + 1;
}
```

14. Las siguientes llamadas de función son aceptables:

```
dos(mi_arreglo, 29);
dos(mi_arreglo, 10);
dos(tu_arreglo, 100);
```

La llamada

```
dos(mi_arreglo, 10);
```

es válida, pero sólo llena las primeras 10 variables indizadas de `mi_arreglo`. Si eso es lo que desea, la llamada es aceptable.

Las siguientes llamadas de función son incorrectas:

```
dos(mi_arreglo, 55);
"Hey, dos. Ven por favor."
dos(mi_arreglo[3], 29);
```

La primera es incorrecta porque el segundo argumento es demasiado grande. La segunda porque no tiene signo de punto y coma al final (y por otras razones). La tercera es incorrecta porque usa una variable indizada como argumento, cuando debería usar todo el arreglo.

15. Podemos hacer que el parámetro de arreglo de `salida` sea un parámetro constante, ya que no hay necesidad de modificar los valores de las variables indizadas del parámetro de arreglo. No podemos hacer constante el parámetro de `omitir_impares` porque la función podría modificar los valores de algunas de sus variables indizadas.

```
void salida(const double a[], int tamanio);
//Precondición: a[0] a a[tamanio − 1] tienen valores.
//Postcondición: Se enviaron a[0] hasta a[tamanio − 1] a la
//salida.

void omitir_impares(int a[], int tamanio);
//Precondición: a[0] hasta a[tamanio − 1] tienen valores.
//Postcondición: Todos los números impares en a[0] hasta
//a[tamanio − 1] se han convertido en 0.
```

16.
```
int en_desorden (double arreglo[], int tamanio)
{
   for(int i = 0; i < tamanio−1; i++)
   if (arreglo[i] > arreglo[i+1] )//obtenemos a[i+1] para cada i.
       return i+1;
   return −1;
}
```

17.
```
#include <iostream>
using namespace std;
const int TAM_DECLARADO = 10;
int main()
{
cout << "Escriba hasta diez enteros no negativos.\n"
     << "Coloque un numero negativo al final.\n";
int arr_numeros[TAM_DECLARADO], siguiente, indice = 0;
cin >> siguiente;
while ( (siguiente >= 0) && (indice < TAM_DECLARADO) )
{
    arr_numeros[indice] = siguiente;
    indice++;
    cin >> siguiente;
}
int elem_usados = indice;
cout << "Te devuelvo los numeros:\n";
for (indice = 0; indice < elem_usados; indice++)
     cout << arr_numeros[indice] << " ";
cout << endl;
return 0;
}
```

18.
```
#include <iostream>
using namespace std;
const int TAM_DECLARADO = 10;

int main()
{
    cout << "Escribe hasta diez letras"
         << " seguidas de un punto:\n";
    char caja_de_letras[TAM_DECLARADO], siguiente;
    int indice = 0;
    cin >> siguiente;
    while ( (siguiente != '.') && (indice < TAM_DECLARADO) )
{
         caja_de_letras[indice] = siguiente;
         indice++;
         cin >> siguiente;
}
int elem_usados = indice;
cout << "He aqui las letras al reves:\n";
for (indice = elem_usados-1; indice >= 0; indice --)
    cout << caja_de_letras[indice];
cout << endl;
return 0;
}
```

19.
```
bool buscar(const int a[], int elem_usados,
                           int buscado, int& donde)
{
     int indice = 0;
     bool hallado = false;
     while ((!hallado) && (indice < elem_usados))
     if (buscado == a[indice])
```

```
            hallado = true;
        else
            indice++;
        //Si se halló buscado, entonces
        //hallado == true y a[indice] == buscado.
        if (hallado)
            donde = indice;
        return hallado;
    }
```

20.
```
    struct Marcador
    {
          int equipo_de_casa;
          int oponente;
    };
    Marcador partido[10];
```

21.
```
    //Lee 5 cantidades de dinero, duplica cada cantidad
    //y envía los resultados a la salida.
    #include <iostream>
    #include "dinero.h"

    int main()
    {
        using namespace std;
        Dinero monto[5];
        int i;
        cout << "Escriba 5 cantidades de dinero:\n";
        for (i = 0; i < 5; i++)
            cin >> monto[i];
        for (i = 0; i < 5; i++)
        monto[i] = monto[i] + monto[i];
        cout << "Despues de doblarlas, las cantidades son:\n";
        for (i = 0; i < 5; i++)
             cout << monto[i] << " ";
        cout << endl;

        return 0;

    }
```

(No podemos usar `2*monto[i]` porque no hemos sobrecargado `*` para operandos de tipo `Dinero`.)

22. Vea la respuesta 23.

23. Esta respuesta combina las respuestas a este ejercicio de autoevaluación y al anterior. La definición de clase cambiaría a lo siguiente. (Hemos eliminado algunos comentarios del cuadro 10.15 para ahorrar espacio, pero hay que incluirlos en la respuesta.)

```
    namespace savitchlistat
    {
        class ListaTemperaturas
        {
        public:
            ListaTemperaturas();
            int obtener_tamanio() const;
            //Devuelve el número de temperaturas en la lista.
```

```
        void agregar_temperatura(double temperatura);

        double obtener_temperatura(int posicion) const;
        //Precondición: 0 <= posicion < obtener_tamanio().
        //Devuelve la temperatura que se añadió en la posición
        //especificada. La primera temperatura agregada está
        //en la posición 0.

        bool llena() const;
        friend ostream& operator <<(ostream& sale,
                          const ListaTemperaturas& el_objeto);
    private:
        double lista[TAM_MAX_LISTA]; //de temperaturas Fahrenheit
        int tamanio; //número de posiciones del arreglo ocupadas
    };

}//namespace savitchlistat
```

También es necesario añadir las siguientes definiciones de funciones miembro:

```
int ListaTemperaturas::obtener_tamanio() const
{
    return tamanio;
}
//Usa iostream y cstdlib:
double ListaTemperaturas::obtener_temperatura(int posicion) const
{
     if ( (posicion >= tamanio) || (posicion < 0) )
     {
        cout << "Error: se esta leyendo "
             << "una posicion vacia de la lista.\n"
        exit(1);
     }
     else
     {
        return (lista[posicion]);
     }
}
```

24.
```
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
```

25.
```
int a[4][5];
int indice1, indice2;
for (indice1 = 0; indice1 < 4; indice1++)
     for (indice2 = 0; indice2 < 5; indice2++)
         cin >> a[indice1][indice2];
```

26.
```
void eco(const int a[][5], int tamanio_de_a)
//Despliega los valores del arreglo a en tamanio_de_a líneas
//con 5 números por línea.
{
    for (int indice1 = 0; indice1 < tamanio_de_a; indice1++)
```

```
    {
        for (int indice2 = 0; indice2 < 5; indice2++)
            cout << a[indice1][indice2] << " ";
        cout << endl;
    }
}
```

Proyectos de programación

Los proyectos 1 al 7 no requieren el uso de estructuras ni clases (aunque el proyecto 7 se puede resolver de forma más elegante usando estructuras). Los proyectos 8 al 11 se diseñaron para resolverse usando estructuras o clases. Los proyectos 12 a 15 se diseñaron para resolverse utilizando arreglos multidimensionales y no requieren ni estructuras ni clases (aunque en algunos casos se pueden resolver de forma más elegante usando clases o estructuras).

1. Hay tres versiones de este proyecto:

Versión 1 (todo interactivo): Escriba un programa que lea la precipitación pluvial media mensual en una ciudad para cada mes del año y luego lea la precipitación mensual real para cada uno de los 12 meses anteriores. El programa desplegará una tabla bien presentada que muestre la precipitación pluvial para cada uno de los doce meses anteriores e indique qué tanto por debajo o por arriba de la media estuvo la precipitación de cada mes. La precipitación media mensual se da para los meses de enero, febrero, etcétera, en orden. Para obtener la precipitación real en los doce meses anteriores, el programa primero pregunta en qué mes estamos y luego pide las cifras de precipitación para los doce meses anteriores. La salida deberá rotular correctamente los meses.

Hay varias formas de manejar los nombres de los meses. Un método sencillo consiste en codificar los meses como enteros y luego realizar una conversión antes de generar la salida. Una instrucción `switch` grande es aceptable en una función de salida. La introducción de meses se puede manejar de cualquier manera que se desee en tanto sea relativamente fácil y agradable para el usuario.

Después de terminar el programa anterior, produzca una versión mejorada que también despliegue un gráfico que muestre la precipitación media y la precipitación real para cada uno de los doce meses anteriores. El gráfico deberá ser similar a la que se muestra en el cuadro 10.8, excepto que deberá haber dos barras para cada mes y deberán rotularse como precipitación media y precipitación para el mes más reciente. El programa deberá preguntar al usuario si quiere ver la tabla o el gráfico de barras, y luego desplegar el formato solicitado. Incluya un ciclo que permita al usuario ver cualquiera de los formatos cuantas veces desee, hasta que pida al programa que termine.

Versión 2 (combina salidas interactivas y con archivos): Una versión más completa también permitirá al usuario solicitar que la tabla y el gráfico se envíen a un archivo. El usuario introduce el nombre del archivo. Este programa hace lo mismo que el de la Versión 1, pero tiene esta característica adicional. Para leer un nombre de archivo es preciso usar material presentado en la sección opcional del capítulo 5 titulada "Nombres de archivo como entradas".

Versión 3 (toda E/S con archivos): Esta versión es como la Versión 1 sólo que las entradas se toman de un archivo y las salidas se envían a un archivo. Puesto que no hay usuario con el cual interactuar, no hay ciclo que permita repetir la salida; tanto la tabla como el gráfico se envían al mismo archivo. Si este proyecto se deja de tarea, pida a su profesor instrucciones respecto a los nombres de archivo que usará.

2. Los números hexadecimales están escritos en base 16. Los 16 dígitos que se utilizan son del `'0'` al `'9'` y la `'a'` para el "dígito 10", la `'b'` para el "dígito 11", la `'c'` para el "dígito 12", la `'d'` para el "dígito 13", la `'e'` para el "dígito 14" y la `'f'` para el "dígito 15". Por ejemplo el número hexadecimal d es igual al número 14 en base decimal; y el número hexadecimal `1d` es `30` en base decimal. Escribir un programa en C++ que realice la suma de dos números hexadecimales con más de 10 dígitos. Si el resultado de la suma es más

de 10 dígitos de largo, entonces enviar el mensaje "Resultado muy grande" y no desplegar el resultado de la suma. Utilice arreglos para almacenar números decimales como arreglos de caracteres. Incluya un ciclo para repetir el cálculo para nuevos números hasta que el usuario decida terminar el programa.

3. 

Escriba una función llamada `eliminar_repetidos` que tenga un arreglo de caracteres parcialmente lleno como parámetro y que elimine todas las letras repetidas del arreglo. Puesto que un arreglo parcialmente lleno requiere dos argumentos, la función tendrá en realidad dos parámetros formales: un parámetro de arreglo y un parámetro formal de tipo `int` que dé el número de posiciones ocupadas del arreglo. Cuando se elimina una letra, las letras restantes se desplazan hacia el principio del arreglo para cerrar el hueco. Esto creará posiciones vacías al final del arreglo, y la porción ocupada del arreglo será menor. Puesto que el parámetro formal es un arreglo parcialmente lleno, un segundo parámetro formal de tipo `int` indicará cuántas posiciones del arreglo están ocupadas. Este segundo parámetro formal será un parámetro de llamada por referencia y se modificará a modo de indicar qué tanto del arreglo está ocupado después de eliminarse las letras repetidas.

Por ejemplo, considere el siguiente código:

```
char a[10];
a[0] = 'a';
a[1] = 'b';
a[2] = 'a';
a[3] = 'c';
int tamanio = 4;
eliminar_repetidas(a, tamanio);
```

Después de ejecutarse este código, el valor de `a[0]` es `'a'`, el valor de `a[1]` es `'b'`, el valor de `a[2]` es `'c'` y el valor de `tamanio` es `3`. (El valor de `a[3]` ya no nos interesa, pues el arreglo parcialmente lleno ya no usa esta variable indizada.

Podemos suponer que el arreglo parcialmente lleno sólo contiene letras minúsculas. Incorpore su función en un programa de prueba apropiado.

4. La desviación estándar de una lista de números es una medida de qué tanto los números difieren del promedio. Si la desviación estándar es pequeña, los números están agrupados cerca del promedio. Si la desviación estándar es grande, los números están dispersos lejos del promedio. La desviación estándar, S, de una lista de N números x_i se define así:

$$S = \sqrt{\frac{\sum_{i=1}^{N} (x_i - \overline{x})^2}{N}}$$

donde $\overline{x}$ es la media de los N números x_1, x_2, ... Defina una función que reciba un arreglo de números parcialmente lleno como argumento y devuelva la desviación estándar de los números contenidos en el arreglo. Puesto que un arreglo parcialmente lleno requiere dos argumentos, la función en realidad tendrá dos parámetros formales, un parámetro de arreglo y uno de tipo `int` que dé el número de posiciones ocupadas del arreglo. Los números contenidos en el arreglo son de tipo `double`. Incorpore su función en un programa de prueba adecuado.

5. Escriba un programa que lea un arreglo de tipo `int`. Contemple la lectura de este arreglo tanto del teclado como de un archivo, como indique el usuario. Si el usuario escoge entrada de archivo, el programa deberá solicitar un nombre de archivo. Puede suponer que hay menos de 50 entradas en el arreglo. El programa determina el número de entradas. La salida debe ser una lista de dos columnas. La primera es una lista de los distintos elementos del arreglo; la segunda es un conteo del número de ocurrencias de cada elemento. La lista deberá ordenarse según las entradas de la primera columna, de mayor a menor.

Por ejemplo, para la entrada

−12 3 −12 4 1 1 −12 1 −1 1 2 3 4 2 3 −12

la salida deberá ser

```
N       Cuenta
4       2
3       3
2       2
1       4
−1      1
−12     4
```

6 En el texto se habla del ordenamiento por selección. Proponemos una rutina distinta, el ordenamiento por inserción. Esta rutina es en cierto sentido lo contrario del ordenamiento por selección en cuanto a que toma elementos sucesivos del arreglo y los *inserta* en la posición correcta de un subarreglo que ya está ordenado (en un extremo del arreglo que estamos ordenando).

El arreglo por ordenar se divide en un subarreglo ordenado y un subarreglo no examinado. Inicialmente, el subarreglo ordenado está vacío. Cada elemento del subarreglo no examinado se toma y se inserta en su posición correcta en el subarreglo ordenado.

Escriba una función y un programa de prueba que implemente el ordenamiento por inserción. Pruebe exhaustivamente su programa.

Ejemplo y sugerencias: La implementación implica un ciclo exterior que selecciona elementos sucesivos del subarreglo no ordenado y un ciclo anidado que inserta cada elemento en su posición correcta en el subarreglo ordenado.

Inicialmente, el subarreglo ordenado está vacío, y el subarreglo no ordenado es todo el arreglo:

```
a[0]   a[1]   a[2]    a[3]    a[4]    a[5]    a[6]    a[7]    a[8]    a[9]
  8      6     10       2      16       4      18      14      12      10
```

Tomamos el primer elemento, `a[0]`, (o sea `8`) y lo colocamos en la primera posición. El ciclo interior no tiene nada que hacer en este primer caso. El arreglo y los subarreglos quedan así:

ordenado no ordenado

```
a[0]   a[1]   a[2]    a[3]    a[4]    a[5]    a[6]    a[7]    a[8]    a[9]
  8      6     10       2      16       4      18      14      12      10
```

Ahora tomamos el primer elemento del subarreglo no ordenado (`a[1]`, que tiene el valor `6`). Lo insertamos en el subarreglo ordenado en su posición correcta. Esto está fuera de orden, de manera que el ciclo interior intercambia los valores de la posición `0` y la posición `1`. El resultado es:

ordenado no ordenado

```
a[0]   a[1]   a[2]    a[3]    a[4]    a[5]    a[6]    a[7]    a[8]    a[9]
  6      8     10       2      16       4      18      14      10      12
```

Observe que el subarreglo ordenado ya creció en una entrada.

Repita el proceso con la primera entrada del subarreglo no ordenado, `a[2]`, encontrando un lugar donde colocar `a[2]` de modo que el primer subarreglo siga ordenado. Puesto que `a[2]` ya está en el lugar correcto, es decir, su valor es mayor que el del elemento más grande del subarreglo ordenado, el ciclo interior no tiene que hacer nada. El resultado es el siguiente:

```
ordenado                  no ordenado

a[0]   a[1]   a[2]    a[3]    a[4]    a[5]    a[6]    a[7]    a[8]    a[9]
  6      8      10       2      16       4      18      14      10      12
```

Una vez más, tomamos el primer elemento del subarreglo no ordenado, `a[3]`, pero esta vez el ciclo interior tiene que intercambiar valores hasta que el valor de `a[3]` queda en su posición correcta. Esto implica varios intercambios:

```
ordenado                       no ordenado

a[0]   a[1]   a[2]    a[3]    a[4]    a[5]    a[6]    a[7]    a[8]    a[9]
  6      8       10<-->2              16       4      18      14      10      12

ordenado                       no ordenado

a[0]   a[1]   a[2]    a[3]    a[4]    a[5]    a[6]    a[7]    a[8]    a[9]
  6       8<--->2              10      16       4      18      14      10      12

ordenado                       no ordenado

a[0]   a[1]   a[2]    a[3]    a[4]    a[5]    a[6]    a[7]    a[8]    a[9]
  6<--->2              8       10      16       4      18      14      10      12
```

El resultado de colocar el 2 en el subarreglo ordenado es

```
ordenado                       no ordenado

a[0]   a[1]   a[2]    a[3]    a[4]    a[5]    a[6]    a[7]    a[8]    a[9]
  2      6      8       10      16       4      18      14      10      12
```

El algoritmo continúa de este modo hasta que el subarreglo no ordenado está vacío, y el subarreglo ordenado tiene todos los elementos del arreglo original.

7. Podemos usar un arreglo para almacenar enteros grandes dígito por dígito. Por ejemplo, el entero `1234` podría almacenarse en el arreglo `a` asignando `1` a `a[0]`, `2` a `a[1]`, `3` a `a[2]` y `4` a `a[3]`. Sin embargo, para este ejercicio podría ser más útil almacenar los dígitos al revés, es decir, `4` en `a[0]`, `3` en `a[1]`, `2` en `a[2]` y `1` en `a[3]`.

En este ejercicio escribirá un programa que lee dos enteros positivos con 20 o menos dígitos de longitud y produce la suma de los dos números. El programa leerá los dígitos como valores de tipo *char* de modo que el número 1234 se lea como los cuatro caracteres '1', '2', '3' y '4'. Una vez introducidos en el programa, los caracteres se convertirán en valores de tipo *int*. Los dígitos leídos se colocarán en un arreglo parcialmente lleno, y podría ser útil invertir el orden de los elementos del arreglo después de llenarlo con los datos del teclado. (Usted decide si invierte o no el orden de los elementos. El proyecto puede hacerse de las dos maneras y ambas tienen sus ventajas y sus desventajas.)

El programa efectuará la suma implementando el algoritmo acostumbrado de suma con lápiz y papel. El resultado de la suma se almacena en un arreglo de tamaño 20 y luego el resultado se despliega en la pantalla. Si el resultado de la suma es un entero con más del número máximo de dígitos (20), el programa deberá desplegar un mensaje indicando que hubo "desbordamiento de enteros". Deberá ser posible cambiar la longitud máxima de los enteros modificando sólo una constante definida globalmente. Incluya un ciclo que permita al usuario seguir efectuando sumas hasta que diga que el programa debe terminar.

8. Escriba un programa que lea una línea de texto y produzca una lista de todas las letras que ocurren en el texto junto con el número de veces que cada letra ocurre en la línea. Termine la línea con un punto que funcione como valor centinela. Las letras deberán aparecer en orden, desde la más frecuente hasta la menos frecuente. Use un arreglo con un tipo *struct* como tipo base de modo que cada elemento del arreglo pueda contener tanto una letra como un entero. Puede suponer que la entrada consiste únicamente en letras minúsculas. Por ejemplo, la entrada

do be do bo

deberá producir una salida similar a la siguiente:

```
Letra:      Número de ocurrencias
    o       3
    d       2
    b       2
    e       1
```

El programa tendrá que ordenar el arreglo según los miembros enteros de las estructuras del arreglo. Esto requerirá modificar la función `ordenar` que se dio en el cuadro 10.12. No puede usarse `ordenar` para resolver este problema sin modificarla. Si este proyecto se deja como tarea, pregunte a su profesor si la entrada/salida se debe efectuar con el teclado y la pantalla o con archivos. Si va a hacerse con archivos, pida a su profesor instrucciones respecto a los nombres de archivo.

9. Escriba un programa que juzgue manos de póquer de cinco cartas, usando las siguientes categorías: nada, un par, dos pares, tercia, corrida (en orden sin huecos), flor (todas del mismo palo, por ejemplo, todas espadas), full (un par y una tercia), póquer (cuatro cartas del mismo valor), flor imperial (tanto flor como corrida). Use un arreglo de estructuras para almacenar la mano. La estructura tendrá dos variables miembro: una para el valor de la carta y una para el palo. Incluya un ciclo que permita al usuario seguir juzgando más manos hasta que indique que el programa debe terminar.

10. Escriba un programa para balancear una chequera. El programa leerá lo siguiente para todos los cheques que no se habían cobrado en la última ocasión que se balanceó la chequera: el número de cada cheque, el importe del cheque, y si ya se cobró o no. Use un arreglo con un tipo base de clase. La clase deberá ser específica para un cheque. Deberá haber tres variables miembro para registrar el número de cheque, el importe y si el cheque se cobró o no. La clase Cheque tendrá una variable miembro de tipo `Dinero` (como se define en el cuadro 10.13) para registrar el importe del cheque. Así pues, tendremos una clase dentro de otra. La clase Cheque deberá tener funciones de acceso además de constructores y funciones de entrada/salida.

Además de los cheques, el programa lee todos los depósitos, así como el saldo anterior y el saldo nuevo de la cuenta. Podría usarse otro arreglo para guardar los depósitos. El nuevo saldo de la cuenta deberá ser el saldo anterior más todos los depósitos menos todos los cheques cobrados. El programa deberá desplegar el total de los cheques cobrados, el total de los depósitos, el nuevo saldo según el balance y la diferencia de éste respecto al saldo que da el banco. También se generan dos listas de cheques: los que se cobraron desde la última vez que se balanceó la chequera y los que no se han cobrado. Muestre ambas listas de cheques en orden de menor a mayor número de cheque. Si el proyecto se deja como tarea, pregunte a su profesor si la entrada/salida se debe efectuar con el teclado y la pantalla o con archivos. Si va a hacerse con archivos, pida a su profesor instrucciones respecto a los nombres de archivo.

11. Defina una clase llamada `Lista` que pueda contener una lista de valores de tipo *`double`*. Modele su definición de clase según la clase `ListaTemperaturas` que se dio en los cuadros 10.15 y 10.16, pero la clase `Lista` no hará referencia a temperaturas cuando despliega los valores. Los valores pueden representar datos de cualquier especie en tanto sean de tipo *`double`*. Incluya las características adicionales que se especificaron en los ejercicios de autoevaluación 22 y 23. Cambie los nombres de las funciones miembro de modo que no tengan que ver con temperaturas. Añada una función miembro llamada `obtener_ultimo` que no reciba argumentos y devuelva el último elemento de la lista. La función miembro `obtener_ultimo` no modifica la lista, y no se deberá invocar cuando la lista esté vacía. Añada otra función miembro llamada `eliminar_ultimo` que elimine el último elemento de la lista. La función miembro `eliminar_ultimo` es una función `void`. Cabe señalar que, cuando se elimina el último elemento, hay que ajustar la variable miembro `tamanio`. Si `eliminar_ultimo` se invoca con una lista vacía como objeto invocador, la llamada de función no tiene ningún efecto. Se debe colocar la definición de la clase en un archivo de interfaz y en un archivo de implementación como hicimos con el tipo `ListaTemperaturas` en los cuadros 10.15 y 10.16. Diseñe un programa que pruebe exhaustivamente la definición de la clase `Lista`.

12. Escriba un programa que permita a dos usuarios jugar "gato". El programa deberá pedir jugadas de forma alternada al jugador X y al jugador O. El programa muestra las posiciones del juego así:

```
1   2   3
4   5   6
7   8   9
```

Los jugadores introducen sus jugadas indicando el número de la posición que desean marcar. Después de cada jugada, el programa desplegará el tablero modificado. Un ejemplo de configuración del tablero es:

```
X   X   0
4   5   6
0   8   9
```

13. Escriba un programa para asignar asientos a los pasajeros de un avión. Suponga que el avión es pequeño y tiene la siguiente numeración de asientos:

```
1   A B   C  D
2   A B   C  D
3   A B   C  D
4   A B   C  D
5   A B   C  D
6   A B   C  D
7   A B   C  D
```

El programa deberá mostrar el patrón de asientos marcando con una `'X'` los asientos que ya se asignaron. Por ejemplo, después de ocuparse los asientos 1A, 2B y 4C, la pantalla deberá ser:

```
1   X  B   C  D
2   A  X   C  D
3   A  B   C  D
4   A  B   X  D
5   A  B   C  D
6   A  B   C  D
7   A  B   C  D
```

Después de mostrar los asientos disponibles, el programa pide el asiento deseado, el usuario teclea un asiento y luego se actualiza la presentación de asientos disponibles. Esto continúa hasta que todos los asientos se ocupan o hasta que el usuario indica que el programa debe terminar. Si el usuario teclea un asiento que ya se asignó, el programa deberá decir que el asiento está ocupado y pedir otra selección.

14. Escriba un programa que acepte entradas como el programa del cuadro 10.8 y produzca una gráfica de barras como la del cuadro, sólo que este programa las presentará verticalmente. Podría ser útil un arreglo bidimensional.

15. El matemático John Horton Conway inventó el "Juego de la Vida". Aunque no es un juego tradicional, produce una conducta interesante que se especifica con unas cuantas reglas. Se debe escribir un programa que permita especificar una configuración inicial. El programa seguirá las reglas de la Vida (para mostrar la conducta de la configuración.

    VIDA es un organismo que vive en un mundo bidimensional discreto. Aunque en realidad este mundo es ilimitado, no podemos darnos ese lujo, por lo que limitaremos el arreglo a 80 posiciones a lo ancho y 22 posiciones a lo alto. Si tiene acceso a una pantalla más grande, úsela.

    Este mundo es un areglo y cada celda puede contener una célula de VIDA. Las generaciones marcan el paso del tiempo. Cada generación trae nacimientos y decesos a la comunidad de VIDA.

    Los nacimientos y decesos siguen este conjunto de reglas:

- Cada célula puede tener hasta ocho células *vecinas*. Las vecinas de una célula son las que está directamente arriba, abajo, a la izquierda, a la derecha, diagonalmente arriba a la izquierda y la derecha, y diagonalmente abajo a la izquierda y la derecha.

- Si una célula tiene 0 o 1 vecinas, muere de *soledad*; si tiene más de 3 vecinas, muere por *sobrepoblación*.

- Si una celda vacía tiene exactamente tres celdas vecinas ocupadas, *nace* ahí una nueva célula.

- Los nacimientos y muertes son instantáneos, y ocurren al cambio de generación. Una célula que muere por cualquier razón puede ayudar a causar un nacimiento, pero una célula recién nacida no puede revivir a una célula que está muriendo, y tampoco la muerte de una célula puede evitar la muerte de otra (al reducir la población local).

Notas: algunas configuraciones crecen a partir de configuraciones iniciales relativamente pequeñas. Otras se desplazan por la región. Se recomienda usar para salidas de texto un arreglo rectangular de `char` con 80 columnas y 22 filas para almacenar las generaciones sucesivas del mundo de VIDA. Use un `*` para indicar una célula viva y un espacio en blanco para indicar una celda vacía. Si su pantalla tiene más filas, use toda la pantalla.

Ejemplos:

```
***
```

se vuelve

```
*
*
*
```

y luego

```
***
```

otra vez, y así.

Sugerencias: Busque configuraciones estables. Es decir, busque comunidades que repitan patrones continuamente. El número de configuraciones de la repetición se llama *periodo*. Hay configuraciones fijas, que no cambian. Un posible proyecto es encontrar tales configuraciones.

Consejo: Defina una función `void` llamada `generacion` que reciba un arreglo llamado `mundo`: un arreglo de `char` de 80 columnas por 22 filas que contiene la configuración inicial. La función examina el arreglo y modifica las celdas, marcándolas con nacimientos o muertes según las reglas dadas. Esto implica examinar cada celda por turno y decidir si se matará o dejará vivir a la célula que la ocupa; o, si la celda está vacía, decidir si nacerá una célula. Deberá haber una función `mostrar` que acepte el arreglo `mundo` y lo despliegue en la pantalla. Se necesita cierto retraso entre las llamadas a `generacion` y `mostrar`. Para ello, el programa deberá generar y desplegar la siguiente generación cuando el usuario oprima Intro. Se puede automatizar esto, pero no es necesario.

11 Cadenas y vectores

11 Cadenas y vectores

Polonio: ¿Qué está leyendo mi señor?
Hamlet: Palabras, palabras, palabras.

WILLIAM SHAKESPEARE, *Hamlet*

Introducción

En este capítulo hablaremos sobre dos temas en los que se utilizan arreglos, o que están relacionados con los arreglos: cadenas y vectores. Aunque estos dos temas están muy relacionados entre sí, esta relación no siempre es obvia y no dependen uno del otro. Los temas de cadenas y vectores se pueden cubrir en cualquier orden.

En las secciones 11.1 y 11.2 presentaremos dos tipos cuyos valores representan cadenas de caracteres, como `"Hola"`. Uno de esos tipos, que veremos en la sección 11.1, es sólo un arreglo con el tipo base *`char`* en el que se almacenan cadenas de caracteres y se marca el final de la cadena con el carácter nulo `'\0'`. Esta es la manera antigua de representar cadenas, que C++ heredó del lenguaje de programación C. Estos tipos de cadenas se llaman cadenas tipo C. Aunque las cadenas tipo C son una manera antigua de representar las cadenas, es difícil realizar cualquier tipo de procesamiento en C++ sin tener aunque sea un mínimo contacto con las cadenas tipo C. Por ejemplo, las cadenas entre comillas tales como `"Hola"` se implementan en C++ como cadenas tipo C.

El estándar ANSI/ISO de C++ incluye una herramienta para manejo de cadenas más moderna en la forma de la clase `string`. Esta clase es el segundo tipo de cadenas que veremos en este capítulo, el cual cubriremos en la sección 11.2.

Los vectores pueden considerarse como arreglos que pueden crecer (y reducirse) en longitud mientras un programa se está ejecutando. En C++, una vez que su programa crea un arreglo, no se puede cambiar su longitud. Los vectores sirven para el mismo propósito que los arreglos, sólo que pueden cambiar su longitud mientras el programa se está ejecutando.

Prerrequisitos

Las secciones 11.1 y 11.2, en las que se cubren las cadenas, y la sección 11.3 que cubre los vectores, son independientes unas de otras. Si desea cubrir el tema de los vectores antes de las cadenas, no hay problema.

En la sección 11.1 sobre cadenas tipo C se utiliza material de los capítulos del 2 al 5, del capítulo 7 y de las secciones 10.1, 10.2 y 10.3 del capítulo 10; no se utiliza nada de material sobre clases de los capítulos 6, 8 o 9.

En la sección 11.2 que trata acerca de la clase `string` se utiliza la sección 11.1 y material de los capítulos del 2 al 7, junto con las secciones 10.1, 10.2 y 10.3 del capítulo 10.

En la sección 11.3 que trata sobre vectores se utiliza material de los capítulos del 2 al 7 y de las secciones 10.1, 10.2 y 10.3 del capítulo 10.

11.1 Un tipo de arreglo para las cadenas

En todo se debe considerar el final.
Jean de La Fontaine, *Fábulas*, Libro III (1668)

En esta sección describiremos una forma de representar cadenas de caracteres que C++ ha heredado del lenguaje C. En la sección 11.2 describiremos una clase llamada string, la cual representa una forma más moderna de representar las cadenas. Aunque el tipo de cadena que describiremos aquí pueda estar un poco "fuera de moda", aún se utiliza mucho y constituye una parte integral del lenguaje C++.

Valores y variables de cadena tipo C

Una manera de representar una cadena es como un arreglo con el tipo base `char`. Si la cadena es `"Hola"`, es útil representarla como un arreglo de caracteres con seis variables indizadas: cuatro para las cuatro letras en `"Hola"` más una para el carácter `'\0'`, que sirve como marcador final. El carácter `'\0'` se llama **carácter nulo** y se utiliza como marcador final porque es distinto de todos los caracteres "reales". El marcador final permite que su programa lea el arreglo un carácter a la vez y sabe que debe dejar de leer cuando lea el marcador final `'\0'`. Una cadena que se almacena de esta forma (como un arreglo de caracteres con la terminación `'\0'`) se llama **cadena tipo C**.

el carácter nulo `'\0'`

cadena tipo C

Escribimos `'\0'` con dos símbolos cuando lo escribimos en un programa, pero al igual que el carácter de nueva línea `'\n'`, el carácter `'\0'` es en realidad un valor de carácter individual. Al igual que cualquier otro valor de carácter, `'\0'` puede almacenarse en una variable de tipo `char` o en una variable indizada de un arreglo de caracteres.

El carácter nulo, `'\0'`

Este carácter se utiliza para marcar el final de una `cadena tipo C` que se almacena en un arreglo de caracteres. Cuando se utiliza un arreglo de caracteres de esta forma, a menudo se le conoce como variable de cadena tipo C. Aunque el carácter nulo `'\0'` se escribe con dos símbolos, es un solo carácter que puede almacenarse en una variable de tipo `char` o en una variable indizada de un arreglo de caracteres.

En este libro ya hemos utilizado cadenas tipo C. En C++, una cadena literal tal como `"Hola"` se almacena como una cadena tipo C, aunque en pocas ocasiones se necesita conocer este detalle.

variable de cadena tipo C

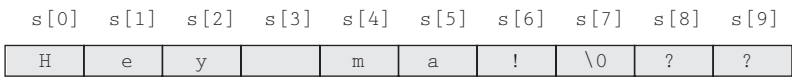
Una **variable de cadena tipo C** es sólo un arreglo de caracteres. Por lo tanto, la siguiente declaración de un arreglo nos proporciona una variable de cadena tipo C capaz de almacenar un valor de cadena tipo C con nueve o menos caracteres:

```
char s[10];
```

El `10` es para las nueve letras en la cadena, más el carácter nulo '\0' para marcar el final de la cadena.

Una variable de cadena tipo C es un arreglo de caracteres que se llena sólo en forma parcial. Al igual que cualquier otro arreglo lleno en forma parcial, una variable de cadena tipo C utiliza posiciones que empiezan en la variable indizada 0 y llegan hasta el número que sea necesario. No obstante, una variable de cadena tipo C no utiliza una variable `int` para llevar la cuenta de cuánto espacio del arreglo se está utilizando. *En vez de ello, una variable de cadena coloca el símbolo especial* `'\0'` *en el arreglo, justo después del último carácter de la*

cadena tipo C. Por lo tanto, si `s` contiene la cadena `"Hey ma!"`, entonces los elementos del arreglo se llenan como se muestra a continuación:



El carácter `'\0'` se utiliza como valor centinela para marcar el final de la cadena tipo C. Si leemos los caracteres de la cadena tipo C empezando con la variable indizada `s[0]`, después `s[1]`, luego `s[2]` y así sucesivamente, sabremos que cuando si encontramos el símbolo `'\0'` habremos llegado al final de la cadena tipo C. Como el símbolo `'\0'` siempre ocupa un elemento del arreglo, la longitud de la cadena más extensa que puede almacenar el arreglo es uno menos que el tamaño del mismo.

comparación entre variables de cadena tipo C y arreglos de caracteres

Lo que distingue a una variable de cadena tipo C de un arreglo ordinario de caracteres es que una variable de cadena tipo C debe contener el carácter nulo `'\0'` al final del valor de cadena tipo C. Ésta es una distinción en cuanto a la forma en que se utiliza el arreglo, más que una distinción en cuanto a lo que es el arreglo en sí. *Una variable de cadena tipo C es un arreglo de caracteres, pero se utiliza de una manera distinta*.

Declaración de variables de cadena tipo C

Una **variable de cadena tipo C** es lo mismo que un arreglo de caracteres, sólo que se usa en forma diferente. Una variable de cadena tipo C se declara como un arreglo de caracteres en la forma usual.

Sintaxis

char Nombre_arreglo[*Tamanio_maximo_cadena_tipo_C* + 1];

Ejemplo

```
char mi_cadena_c[11];
```

El +1 deja un espacio para el carácter nulo `'\0'`, el cual termina cualquier `cadena tipo C` almacenada en el arreglo. Por ejemplo, la variable de `cadena tipo C` llamada `mi_cadena_c` en el ejemplo puede almacenar una `cadena tipo C` que tenga hasta diez caracteres de longitud.

inicialización de variables de cadena tipo C

Podemos inicializar una variable de cadena tipo C al declararla, como se muestra en el siguiente ejemplo:

```
char mi_mensaje[20] = "Hola a todos.";
```

Observe que la cadena tipo C que se asigna a la variable de cadena tipo C no necesita llenar todo el arreglo.

Cuando inicializamos una variable de cadena tipo C se puede omitir el tamaño del arreglo. C++ se encargará de manera automática que el tamaño de la variable de cadena tipo C sea uno más que la longitud de la cadena entre comillas. (La variable indizada extra es para el símbolo `'\0'`.) Por ejemplo,

```
char cadena_corta[] = "abc";
```

es equivalente a

```
char cadena_corta[4] = "abc";
```

Asegúrese de no confundir las siguientes inicializaciones:

```
char cadena_corta[] = "abc";
```

y

```
char cadena_corta[] = {'a', 'b', 'c'};
```

Estas dos inicializaciones *no son equivalentes*. La primera coloca el carácter nulo `'\0'` en el arreglo después de los caracteres `'a'`, `'b'` y `'c'`. La segunda no coloca el `'\0'` en ninguna parte del arreglo.

Inicialización de una variable de cadena tipo C

Una variable de cadena tipo C puede inicializarse al momento de declararla, como se muestra en el siguiente ejemplo:

```
char su_cadena[11] = "Do Be Do"
```

Esta forma de inicialización coloca de manera automática el carácter nulo `'\0'` en el arreglo al final de la cadena tipo C especificada.

Si omitimos el número dentro de los corchetes `[]`, entonces el tamaño de la variable de cadena tipo C será un carácter más que la longitud de la cadena tipo C. Por ejemplo, en el siguiente ejemplo se declara `mi_cadena` para que tenga nueve variables indizadas (ocho para los caracteres de la cadena tipo `C "Do Be Do"` y uno para el carácter nulo `'\0'`:

```
char mi_cadena[] = "Do Be Do";
```

variables indizadas para las variables de cadena tipo C

Una variable de cadena tipo C es un arreglo, por lo cual tiene variables indizadas que pueden utilizarse de la misma forma que en cualquier otro arreglo. Por ejemplo, supongamos que nuestro programa contiene la siguiente declaración de una variable de cadena tipo C:

```
char nuestra_cadena[5] = "Hey";
```

Al declarar `nuestra_cadena` como se muestra arriba, nuestro programa tiene las siguientes variables indizadas: `nuestra_cadena[0]`, `nuestra_cadena[1]`, `nuestra_cadena[2]`, `nuestra_cadena[3]` y `nuestra_cadena[4]`. Por ejemplo, el siguiente código modificará el valor de la cadena tipo C en `nuestra_cadena` para convertirla en una cadena tipo C de la misma longitud en donde todos sus caracteres sean `'X'`:

```
int indice = 0;
while (nuestra_cadena[indice] != '\0')
{
    nuestra_cadena[indice] = 'X';
    indice++;
}
```

No destruya el `'\0'`.

Al manipular estas variables indizadas, hay que tener cuidado de no sustituir el carácter nulo `'\0'` con cualquier otro valor. Si el arreglo pierde el valor `'\0'`, ya no se comportará como una variable de cadena tipo C. Por ejemplo, el siguiente código modificará el arreglo `cadena_feliz` para que ya no contenga una cadena tipo C:

```
char cadena_feliz[7] = "DoBeDo";
cadena_feliz[6] = 'Z';
```

Después de ejecutar este código, el arreglo `cadena_feliz` todavía contendrá las seis letras de la cadena tipo C `"DoBeDo"`, pero `cadena_feliz` ya no contendrá el carácter nulo `'\0'` para marcar el final de la cadena tipo C. Muchas funciones de manipulación de cadenas dependen de una manera considerable de `'\0'` para marcar el final del valor de cadena tipo C.

Como otro ejemplo, considere el ciclo *while* anterior en el que se modifican los caracteres en la variable de cadena tipo C `nuestra_cadena`. Ese ciclo *while* modifica caracteres hasta que se encuentra un `'\0'`. Si el ciclo nunca encuentra el `'\0'`, entonces podría modificar una gran parte de la memoria con valores no deseados, lo cual podría ocasionar que su programa hiciera cosas extrañas. Como medida de seguridad, es conveniente reescribir ese ciclo *while* como se muestra a continuación, de tal forma que si se pierde el carácter nulo `'\0'`, el ciclo no modifique involuntariamente las posiciones de memoria que están más allá del final del arreglo:

```
int indice = 0;
while ( (nuestra_cadena[indice] != '\0') && (indice < TAMANIO) )
{
    nuestra_cadena[indice] = 'X';
    indice++;
}
```

`TAMANIO` es una constante definida, igual al tamaño declarado del arreglo `nuestra_cadena`.

RIESGO *Uso de = y == con cadenas tipo C*

Los valores y las variables de cadena tipo C no son como los valores y las variables de otros tipos de datos; además muchas de las operaciones usuales no funcionan en las cadenas tipo C. No podemos usar una variable de cadena tipo C en una instrucción de asignación que contenga =. Si utilizamos == para evaluar la igualdad de las cadenas tipo C, no obtendremos el resultado esperado. La razón de estos problemas es que las cadenas tipo C y las variables de cadena tipo C son arreglos.

Asignación de un valor a una cadena tipo C.
No es tan simple asignar un valor a una variable de cadena tipo C como con otros tipos de variables. Lo siguiente es ilegal:

```
char una_cadena[10];
una_cadena = "Hola";
```
(¡Ilegal!)

Aunque podemos usar el signo de igual para asignar un valor a una variable de cadena tipo C cuando la declaramos, no podemos usarla en ninguna otra parte del programa. En términos técnicos, el uso del signo igual en una declaración tal como:

```
char cadena_feliz[7] = "DoBeDo";
```

es una inicialización, no una asignación. Si deseamos asignar un valor a una variable de cadena tipo C, debemos hacer otra cosa.

Existen diversas formas para asignar un valor a una variable de cadena tipo C. La más sencilla es mediante el uso de la función predefinida `strcpy`, como se muestra a continuación:

```
strcpy(una_cadena, "Hola");
```

Esto hace que el valor de una_cadena sea igual a "Hola". Por desgracia, esta versión de la función strcpy no comprueba para asegurarse que la copia no se exceda del tamaño de la variable de cadena que es el primer argumento.

Muchas (pero no todas) versiones de C++ también tienen una versión más segura de strcpy. Esta versión más segura se escribe como strncpy (con una n). La función strncpy toma un tercer argumento que proporciona el máximo número de caracteres que se pueden copiar. Por ejemplo:

```
char otra_cadena[10];
strncpy(otra_cadena, una_variable_de_cadena, 9);
```

Con esta función strncpy se copian cuando mucho nueve caracteres (se deja un espacio para '\0') de la variable de cadena tipo C llamada una_variable_de_cadena, sin importar qué tan larga pueda ser la cadena en esta variable.

evaluación de la igualdad de cadenas tipo C

Tampoco podemos utilizar el operador == en una expresión para evaluar si dos cadenas tipo C son iguales. (En realidad es mucho peor que eso. Podemos utilizar el operador == con cadenas tipo C, pero éste no evalúa si las cadenas tipo C son iguales. Por consecuencia, si utilizamos == para evaluar la igualdad de dos cadenas tipo C es muy probable que obtengamos resultados incorrectos, ¡sin ningún mensaje de error!) Para evaluar si dos cadenas tipo C son iguales, podemos usar la función predefinida strcmp. Por ejemplo:

```
if (strcmp(cadena_c1, cadena_c2))
    cout << "Las cadenas NO son iguales.";
else
    cout << "Las cadenas son iguales.";
```

Observe que la función strcmp funciona de manera distinta a la esperada. La comparación es verdadera si las cadenas no concuerdan. La función strcmp compara las cadenas en los argumentos de cadena tipo C un carácter a la vez. Si en cualquier punto la codificación numérica del carácter de cadena_c1 es menor que la codificación numérica del carácter correspondiente de cadena_c2, la evaluación termina y se regresa un número negativo. Si el carácter de cadena_c1 es mayor que el carácter de cadena_c2, entonces se devuelve un número positivo. (Algunas implementaciones de strcmp devuelven la diferencia de las codificaciones de los caracteres, pero no hay que depender de eso.) Si las cadenas tipo C son iguales se devuelve un 0. La relación de orden que se utiliza para comparar caracteres se conoce como **orden lexicográfico**. El punto importante a observar es que si ambas cadenas se encuentran en mayúsculas o en minúsculas, el orden lexicográfico es sólo orden alfabético.

orden lexicográfico

Podemos ver que strcmp devuelve un valor negativo, un valor positivo o cero, dependiendo de si las cadenas tipo C se comparan en forma lexicográfica como menores, mayores o iguales. Si utilizamos strcmp como expresión Booleana en una instrucción *if* o en un ciclo para evaluar la igualdad de cadenas tipo C, entonces el valor distinto de cero se convertirá en verdadero si las cadenas son distintas, y el cero se convertirá en falso. Asegúrese de recordar esta lógica invertida en sus pruebas de igualdad de cadenas tipo C.

Los compiladores de C++ que están en conformidad con el estándar tienen una versión más segura de `strcmp`, la cual tiene un tercer argumento que proporciona el número máximo de caracteres a comparar.

Las funciones `strcpy` y `strcmp` están en la biblioteca que tiene el archivo de encabezado `<cstring>`, por lo que para usarlas necesita insertar la siguiente línea cerca de la parte superior del archivo:

```
#include <cstring>
```

Las funciones `strcpy` y `strcmp` no requieren la siguiente línea ni nada similar (aunque es muy probable que otras partes de su programa lo requieran):[1]

```
using namespace std;
```

La biblioteca `<cstring>`

No necesita ninguna directiva `include` o `using` para poder declarar e inicializar cadenas tipo C. No obstante, al procesar cadenas tipo C tendrá que utilizar algunas de las funciones de cadena predefinidas en la biblioteca `<cstring>`. Por lo tanto, cuando utilice cadenas tipo C por lo general deberá agregar la siguiente directiva include cerca del principio del archivo que contenga su código:

```
#include <cstring>
```

Otras funciones en `<cstring>`

El cuadro 11.1 contiene unas cuantas de las funciones comunes más utilizadas de la biblioteca que tiene el archivo de encabezado `<cstring>`. Para utilizarlas debemos insertar la siguiente línea cerca del principio del archivo:

```
#include <cstring>
```

Al igual que las funciones `strcpy` y `strcmp`, todas las demás funciones en `<cstring>` tampoco requieren la siguiente línea ni nada similar (aunque es probable que otras partes de su programa la requieran):[1]

```
using namespace std;
```

Ya hemos hablado sobre `strcpy` y `strcmp`. La función `strlen` es fácil de comprender y usar. Por ejemplo, `strlen("dobedo")` devuelve `6` ya que hay seis caracteres en `"dobedo"`.

La función `strcat` se utiliza para concatenar dos cadenas tipo C; es decir, para formar una cadena más larga al colocar las dos cadenas tipo C más cortas una después de la

[1] Si no ha leído el capítulo 9, debe ignorar esta nota al pie. Si ya leyó el capítulo 9, los detalles son los siguientes: las definiciones de `strcpy` y `strcmp`, y todas las demás funciones de cadena en `<cstring>` se colocan en el espacio de nombres global no en el espacio de nombres `std`, por lo cual no se requiere una directiva *`using`*.

otra. El primer argumento debe ser una variable de cadena tipo C. El segundo argumento puede ser cualquier cosa que se evalúe como valor de cadena tipo C, como una cadena entre comillas. El resultado se coloca en la variable de cadena tipo C que es el primer argumento. Por ejemplo, considere lo siguiente:

```
char var_cadena[20] = "La lluvia";
strcat(var_cadena, "en Espania");
```

Este código modificará el valor de `var_cadena` para que quede como `"La lluvia en Espania"`. Como lo muestra este ejemplo, hay que tener cuidado de tomar en cuenta los espacios en blanco al concatenar cadenas tipo C.

Si analizamos la tabla del cuadro 11.1 podremos ver que hay disponibles versiones más seguras de tres argumentos de las funciones `strcpy`, `strcat` y `strcmp` en muchas, pero no en todas las versiones de C++. Además, estas versiones de tres argumentos se escriben con una letra `n` adicional: `strncpy`, `strncat` y `strncmp`.

CUADRO 11.1 ***Algunas funciones de cadena tipo C predefinidas en*** `<cstring>` ***(parte 1 de 2)***

Función	Descripción	Precauciones
`strcpy`(*Var_cadena_destino, Cadena_origen*)	Copia el valor de cadena tipo C *Cadena_origen* hacia la variable de cadena tipo C *Var_cadena_destino*.	No verifica que *Var_cadena_destino* sea lo bastante grande como para almacenar el valor de *Cadena_origen*.
`strncpy`(*Var_cadena_destino, Cadena_origen, Límite*)	Igual que la función `strcpy` de dos argumentos, sólo que se copian cuando mucho *Límite* caracteres.	Si *Límite* se elige con cuidado, esta función es más segura que la versión de dos argumentos de `strcpy`. No se implementa en todas las versiones de C++.
`strcat`(*Var_cadena_destino, Cadena_origen*)	Concatena el valor de cadena tipo C *Cadena_origen* con el final de la cadena tipo C que se encuentra en la variable de cadena tipo C *Var_cadena_destino*.	No verifica que *Var_cadena_destino* sea lo bastante grande como para almacenar el resultado de la concatenación.
`strncat`(*Var_cadena_destino, Cadena_origen, Límite*)	Igual que la función `strcat` de dos argumentos, sólo que se anexan cuando mucho *Límite* caracteres.	Si *Límite* se elige con cuidado, esta función es más segura que la versión de dos argumentos de `strcat`. No se implementa en todas las versiones de C++.
`strlen`(*Cadena_origen*)	Devuelve un entero igual a la longitud de *Cadena_origen*. (El carácter nulo `'\0'` no se cuenta en la longitud.)	

CUADRO 11.1 *Algunas funciones de cadena tipo C predefinidas en* `<cstring>` *(parte 2 de 2)*

Función	Descripción	Precauciones
`strcmp(`*Cadena_1, Cadena_2*`)`	Devuelve `0` si *Cadena_1* y *Cadena_2* son iguales. Devuelve un valor `< 0` si *Cadena_1* es menor que *Cadena_2*. Devuelve un valor `> 0` si *Cadena_1* es mayor que *Cadena_2* (es decir, devuelve un valor distinto de cero si *Cadena_1* y *Cadena_2* son distintas.) El orden es lexicográfico.	Si *Cadena_1* es igual a *Cadena_2* esta función devuelve `0`, lo que la convierte en `falsa`. Esto es lo inverso de lo que podríamos esperar que devolviera si las cadenas son iguales.
`strncmp(`*Cadena_1*, *Cadena_2*, *Límite*`)`	Igual que la función `strcat` de dos argumentos, sólo que se comparan cuando mucho *Límite* caracteres.	Si *Límite* se elige con cuidado, esta función es más segura que la versión de dos argumentos de `strcmp`. No se implementa en todas las versiones de C++.

Argumentos y parámetros de cadenas tipo C

Una variable de cadena tipo C es un arreglo, por lo que un parámetro de cadena tipo C para una función es sólo un parámetro tipo arreglo.

Al igual que con cualquier parámetro tipo arreglo, cada vez que una función modifica el valor de un parámetro de cadena tipo C, es más seguro incluir un parámetro `int` adicional que contenga el tamaño declarado de la variable de cadena tipo C.

Por otro lado, si una función sólo utiliza el valor en un argumento de cadena tipo C pero no modifica ese valor, entonces no hay necesidad de incluir otro parámetro para proporcionar el tamaño declarado de la variable de cadena tipo C o la cantidad del arreglo de la variable de cadena tipo C que está lleno. El carácter nulo `'\0'` puede utilizarse para detectar el final del valor de cadena tipo C que se almacena en la variable de cadena tipo C.

Ejercicios de AUTOEVALUACIÓN

1. ¿Cuáles de las siguientes declaraciones son equivalentes?

```
char var_cadena[10] = "Hola";
char var_cadena[10] = {'H', 'o', 'l', 'a', '\0'};
char var_cadena[10] = {'H', 'o', 'l', 'a'};
char var_cadena[6] = "Hola";
char var_cadena[] = "Hola";
```

2. ¿Qué cadena tipo C se almacenará en `cadena_canto` después de ejecutar el siguiente código?

```
char cadena_canto[20] = "DoBeDo";
strcat(cadena_canto, " para ti");
```

Suponga que el código está incrustado en un programa completo y correcto, y que hay una directiva `include` para `<cstring>` en el archivo del programa.

3. ¿Cuál es el error (si lo hay) en el siguiente código?

```
char var_cadena[] = "Hola";
strcat(var_cadena, " y adios.");
cout << var_cadena;
```

Suponga que el código está incrustado en un programa completo, y que hay una directiva `include` para `<string>` en el archivo del programa.

4. Suponga que no se ha definido todavía la función `strlen` (la cual devuelve la longitud de su argumento tipo cadena). Proporcione una definición para la función `strlen`. Observe que `strlen` sólo tiene un argumento, que es una cadena tipo C. No agregue argumentos adicionales; no son necesarios.

5. ¿Cuál es la máxima longitud de una cadena que puede colocarse en la variable de cadena declarada como sigue? Explique.

```
char s[6];
```

6. ¿Cuántos caracteres hay en cada una de las siguientes constantes tipo carácter y tipo cadena?

a) `'\n'`
b) `'n'`
c) `"Mary"`
d) `"M"`
e) `"Mary\n"`

7. Como las cadenas de caracteres son sólo arreglos de tipo `char`, ¿por qué el libro le recomienda que no confunda la siguiente declaración e inicialización?

```
char cadena_corta[] = "abc";
char cadena_corta[] = {'a', 'b', 'c'};
```

8. Dada la siguiente declaración e inicialización de la variable de cadena, escriba un ciclo para asignar `'X'` a todas las posiciones de esta variable de cadena; mantenga la misma longitud.

```
char nuestra_cadena[15] = "Hola a todos!";
```

9. Dada la declaración de una variable de cadena tipo C, en donde `TAMANIO` es una constante definida:

```
char nuestra_cadena[TAMANIO];
```

La variable de cadena tipo C `nuestra_cadena` se ha asignado en cierto código que no se muestra aquí. Para variables de cadena tipo C correctas, el siguiente ciclo reasigna a todas las posiciones de

nuestra_cadena el valor 'X' y deja la misma longitud que antes. Suponga que el siguiente fragmento de código está incrustado en un programa que de otra manera estaría completo y correcto. Responda a las preguntas siguientes a este fragmento de código:

```
int indice = 0;
while (nuestra_cadena[indice] != '\0')
{
    nuestra_cadena[indice] = 'X';
    indice++;
}
```

a) Explique cómo este código puede destruir el contenido de la memoria que está más allá del final del arreglo.

b) Modifique este ciclo para protegerse contra la modificación involuntaria de la memoria que está más allá del final del arreglo.

10. Escriba código mediante el uso de una función de biblioteca para copiar la constante de cadena "Hola" en la variable de cadena que se declara a continuación. Asegúrese de incluir (mediante #include) el archivo de encabezado necesario para obtener la declaración de la función que utilice.

```
char una_cadena[10];
```

11. ¿Qué cadena se desplegará cuando se ejecute el siguiente código? (Suponga, como siempre, que este código está integrado en un programa completo y correcto.)

```
char cancion[10] = "Lo hice ";
char cancion_franks[20];
strcpy ( cancion_franks, cancion );
strcat ( cancion_franks, "a mi manera!" );
cout << cancion_franks << endl;
```

12. ¿Cuál es el problema (si es que lo hay) con este código?

```
char una_cadena[20] = "Como estas? ";
strcat(una_cadena, "Bien, espero.");
```

Entrada y salida de cadenas tipo C

Las cadenas tipo C pueden mostrarse mediante el operador de inserción <<. De hecho, ya lo hemos hecho con las cadenas entre comillas. Podemos usar una variable de cadena tipo C de la misma forma; por ejemplo,

```
cout << noticias << " Wow.\n";
```

en donde noticias es una variable de cadena tipo C.

Es posible llenar una variable de cadena tipo C mediante el uso del operador >>, pero hay que considerar algo. Al igual que para los demás tipos de datos, todo el espacio en blanco (espacios, tabulaciones y retornos de línea) se omite cuando se leen las cadenas tipo C de esta forma. Lo que es más, cada lectura de entrada se detiene en el siguiente espacio o retorno de línea. Por ejemplo, considere el siguiente código:

```
char a[80], b[80];
cout << "Escriba algo de entrada:\n";
cin >> a >> b;
cout << a << b << "FIN DE SALIDA\n";
```

Cuando se incluye en un programa completo, este código produce un diálogo como el siguiente:

```
Escriba algo de entrada:
Do be do para ti!
DobeFIN DE SALIDA
```

Cada una de las variables de cadena tipo C `a` y `b` reciben sólo una palabra de la entrada: `a` recibe el valor de cadena tipo C `"Do"`, ya que el carácter de entrada que va después de `Do` es un espacio en blanco; `b` recibe `"be"`, ya que el carácter de entrada que va después de `be` es un espacio en blanco.

Si desea que su programa lea una línea completa de entrada, puede usar el operador de extracción `>>` para leer la línea una palabra a la vez. Esto puede ser tedioso y de todas formas no leerá los espacios en blanco que haya en la línea. Hay una manera sencilla de leer toda una línea completa de entrada y de colocar la cadena tipo C resultante en una variable de cadena tipo C. Sólo debe usar la función miembro predefinida `getline`, la cual es una función miembro de todo flujo de entrada (como `cin` o un flujo de entrada de archivo). La función `getline` tiene dos argumentos. El primero es una variable de cadena tipo C para recibir la entrada y el segundo es un entero que por lo general es el tamaño declarado de la variable de cadena tipo C. El segundo argumento indica el número máximo de elementos del arreglo en la variable de cadena tipo C que `getline` podrá llenar con caracteres. Por ejemplo, considere el siguiente código:

getline

```
char a[80];
cout << "Escriba algo de entrada:\n";
cin.getline(a, 80);
cout << a << "FIN DE SALIDA\n";
```

Cuando se incluye en un programa completo, este código produce un diálogo como el siguiente:

```
Escriba algo de entrada:
Do be do para ti!
Do be do para ti!FIN DE SALIDA
```

Con la función `cin.getline` se lee toda la línea completa. La lectura termina cuando termina la línea, aún y cuando la cadena tipo C resultante pueda ser más corta que el número máximo de caracteres especificados por el segundo argumento.

Cuando se ejecuta `getline`, la lectura se detiene después de que se llena el número de caracteres dado por el segundo argumento en el arreglo de cadena tipo C, incluso aunque no se haya llegado al final de la línea. Por ejemplo, considere el siguiente código:

```
char cadena_corta[5];
cout << "Escriba algo de entrada:\n";
cin.getline(cadena_corta, 5);
cout << cadena_corta << "FIN DE SALIDA\n";
```

Cuando se incluye en un programa completo, este código produce un diálogo como el siguiente:

```
Escriba algo de entrada:
dobedowap
dobeFIN DE SALIDA
```

Observe que se leen cuatro y no cinco caracteres en la variable de cadena tipo C `cadena_corta`, aún y cuando el segundo argumento es `5`. Esto se debe a que el carácter nulo `'\0'` llena una posición del arreglo. Cada cadena tipo C se termina con el carácter nulo cuando se almacena en una variable de cadena tipo C, y esto siempre consume una posición del arreglo.

entrada/salida con archivos

Las técnicas de entrada y salida para las cadenas tipo C que presentamos para `cout` y `cin` funcionan de la misma forma para la entrada y salida con archivos. El flujo de entrada `cin` puede sustituirse por un flujo de entrada que esté conectado a un archivo. El flujo de salida `cout` puede sustituirse por un flujo de salida que esté conectado a un archivo. (En el capítulo 5 hablamos sobre la E/S de archivos.)

`getline`

La función miembro `getline` puede usarse para leer una línea de entrada y colocar la cadena tipo C de caracteres de esa línea, en una variable de cadena tipo C.

Sintaxis

```
cin.getline(Var_cadena, Caracteres_Máx + 1);
```

Se lee una línea de entrada del flujo ***Flujo_entrada*** y la cadena tipo C resultante se coloca en ***Var_cadena***. Si la línea es más larga que ***Caracteres_Max***, entonces sólo se leen los primeros ***Caracteres_Max*** en la línea. (El +1 se necesita ya que cada cadena tipo C tiene el carácter nulo `'\0'` que se agrega al final de la cadena tipo C y por lo tanto, la cadena almacenada en ***Var_cadena*** es una posición más grande que el número de caracteres que se leen de entrada.)

Ejemplo

```
char una_linea[80];
cin.getline(una_linea, 80);
```

(Puede usar un flujo de entrada conectado a un archivo de texto en lugar de `cin`.)

Ejercicios de AUTOEVALUACIÓN

13. Considere el siguiente código (y asuma que está integrado en un programa completo y correcto, el cual se ejecuta a continuación):

```
char a[80], b[80];
cout << "Escriba algo de entrada:\n";
cin >> a >> b;
cout << a << '-' << b << "FIN DE SALIDA\n";
```

Si el diálogo comienza como se muestra a continuación, ¿cuál será la siguiente línea de salida?

Escriba algo de entrada:

```
El
tiempo es ahora.
```

14. Considere el siguiente código (y asuma que está integrado en un programa completo y correcto, que se ejecuta a continuación):

```
char mi_cadena[80];
cout << "Escriba una linea de entrada:\n";
cin.getline(mi_cadena, 6);
cout << mi_cadena << "<FIN DE SALIDA".
```

Si el diálogo empieza como se muestra a continuación, ¿cuál será la siguiente línea de salida?

```
Escriba una linea de entrada:
Que el vello de los dedos de tus pies crezca largo y ondulado.
```

Conversiones de cadena tipo C a número y entrada robusta

La cadena tipo C "1234" y el número 1234 no son lo mismo. El primer valor es una secuencia de caracteres, el segundo es un número. En nuestro uso diario las escribimos de la misma forma y la distinción no es clara, pero en un programa en C++ no se puede ignorar. Si queremos realizar operaciones aritméticas necesitamos 1234, no "1234". Si queremos agregar una coma al número para tener mil doscientos treinta y cuatro, entonces convertiremos la cadena tipo C "1234" en "1,234". Al diseñar entradas numérica a menudo es útil leerla como una cadena de caracteres, después se edita la cadena y luego se convierte en un número. Por ejemplo, si queremos que nuestro programa lea una cantidad de dinero, la entrada podría empezar o no con un signo de pesos. Si nuestro programa lee porcentajes, la entrada podría tener o no un signo de por ciento al final. Si nuestro programa lee la entrada como una cadena de caracteres, podrá almacenar la cadena en una variable de cadena tipo C y eliminar cualquier carácter no deseado, dejando sólo una cadena tipo C de dígitos. Entonces nuestro programa tendría que convertir esta cadena tipo C de dígitos en un número, lo cual puede realizarse fácilmente con la función predefinida atoi.

Funciones para convertir cadenas tipo C en números

Las funciones atoi, atol y atof pueden usarse para convertir una cadena tipo C de dígitos en el valor numérico correspondiente. Las funciones atoi y atol convierten cadenas tipo C en enteros. La única diferencia entre atoi y atol es que atoi devuelve un valor de tipo *int*, mientras que atol devuelve un valor de tipo *long*. La función atof convierte una cadena tipo C en un valor de tipo *double*. Si el argumento de cadena tipo C (para cualquier función) es tal que no pueda realizarse la conversión, entonces la función devolverá cero. Por ejemplo

```
int x = atoi("657");
```

establece el valor de x en 657, y

```
double y = atof("12.37");
```

establece el valor de y en 12.37.

Cualquier programa que utilice atoi o atof deberá contener la siguiente directiva:

```
#include <cstdlib>
```

atoi

La función `atoi` toma un argumento que es una cadena tipo C y devuelve el valor *`int`* que corresponde a esa cadena tipo C. Por ejemplo, `atoi("1234")` devuelve el entero `1234`. Si el argumento no corresponde a un valor *`int`*, entonces `atoi` devuelve `0`. Por ejemplo, `atoi("#37")` devuelve `0` ya que el carácter `"#"` no es un dígito. El nombre de la función `atoi` es una abreviación de la frase en inglés "alfabético a entero". La función `atoi` está en la biblioteca que tiene el archivo de encabezado `cstdlib`, por lo que cualquier programa que la utilice debe contener la siguiente directiva:

```
#include <cstdlib>
```

atol

Si nuestros números son demasiado grandes como para ser valores de tipo *`int`*, podemos convertirlos de cadenas tipo C a valores de tipo *`long`*. La función `atol` permite la misma conversión que `atoi`, sólo que `atol` devuelve valores de tipo *`long`* y, por lo tanto, puede acomodar valores enteros más grandes (en sistemas en los que esto es importante).

lee_y_limpia

El cuadro 11.2 contiene la definición de una función llamada `lee_y_limpia` que lee una línea de entrada y descarta todos los caracteres que no sean los dígitos del `'0'` al `'9'`. Después, esta función utiliza a la función `atoi` para convertir la cadena de dígitos tipo C "limpia" en un valor entero. Como lo indica el programa de demostración, podemos usar esta función para leer cantidades monetarias sin importar si el usuario incluye el signo de pesos o no. De manera similar, puede leer porcentajes y no importa si el usuario escribe un signo de por ciento o no. Aunque la salida aparenta como si la función `lee_y_limpia` sólo eliminara algunos símbolos, hay algo más que eso. El valor que se produce es un valor *`int`* verdadero que puede usarse en un programa como un número; no es una cadena tipo C de caracteres.

La función `lee_y_limpia` que se muestra en el cuadro 11.2 eliminará cualquier valor que no sea dígito de la cadena que se introduzca, pero no puede comprobar que los dígitos restantes vayan a producir el número que el usuario tenía en mente. Se debe dar la oportunidad al usuario de que vea el valor final y compruebe si es correcto. Si no es así, se debe permitir al usuario que vuelva a introducir la cadena. En el cuadro 11.3 hemos usado la función `lee_y_limpia` en otra función llamada `obtiene_int`, la cual aceptará cualquier cosa que el usuario escriba y le permitirá volver a introducir los datos hasta que esté satisfecho(a) con el número resultante de la cadena de entrada. Es un procedimiento de entrada muy robusto (La función `obtiene_int` es una versión mejorada de la función con el mismo nombre que se muestra en el cuadro 5.7.)

obtiene_int

Las funciones `lee_y_limpia` del cuadro 11.2 y `obtiene_int` del cuadro 11.3 son ejemplos de las diversas funciones de entrada que podemos diseñar mediante la lectura de entrada numérica en forma de valor de cadena. En el proyecto de programación 3 al final de este capítulo usted tendrá que definir una función similar a `obtiene_int` que lea un número de tipo *`double`* en vez de un número de tipo *`int`*. Para escribir esa función sería conveniente tener una función predefinida que convierta un valor de cadena en un número de tipo *`double`*. Por fortuna, la función predefinida `atof` (que también se encuentra en la biblioteca con el archivo de encabezado `cstdlib`) hace justo eso. Por ejemplo, `atof("9.99")` devuelve el valor `9.99` de tipo *`double`*. Si el argumento no corresponde a un número de tipo *`double`*, entonces `atof` devuelve `0.0`. El nombre de la función `atof` es una abreviación de la frase en inglés "alfabético a punto flotante". Recuerde que los números con un punto decimal por lo general se llaman números de *punto flotante*, debido a la manera en que la computadora maneja el punto decimal cuando almacena estos números en la memoria.

atof

CUADRO 11.2 *Cadenas de tipo C a enteros (parte 1 de 2)*

```
//Demuestra el uso de la función lee_y_limpia.
#include <iostream>
#include <cstdlib>
#include <cctype>

void lee_y_limpia(int& n);
//Lee una línea de entrada. Descarta todos los símbolos excepto los dígitos. Convierte
//la cadena tipo C en un entero y hace que n sea igual al valor de este entero.

void nueva_linea( );
//Descarta toda la entrada restante en la línea de entrada actual.
//También descarta la '\n' al final de la línea.

int main( )
{
    using namespace std;
    int n;
    char resp;
    do
    {
          cout << "Escriba un entero y oprima Intro: ";
          lee_y_limpia(n);
           cout << "Esa cadena se convierte en el entero " << n << endl;
          cout << "De nuevo? (si/no): ";
          cin >> resp;
          nueva_linea( );
    } while ( (resp != 'n') && (resp != 'N') );
    return 0;
}
```

CUADRO 11.2 *Cadenas de tipo C a enteros (parte 2 de 2)*

```
//Usa iostream, cstdlib y cctype:
void lee_y_limpia(int& n)
{
    using namespace std;
    const int TAMANIO_ARREGLO = 6;
    char cadena_digito[TAMANIO_ARREGLO];

    char siguiente;
    cin.get(siguiente);
    int indice = 0;
    while (siguiente != '\n')
    {
        if ( (isdigit(siguiente)) && (indice < TAMANIO_ARREGLO - 1) )
        {
            cadena_digito[indice] = siguiente;
            indice++;
         }
         cin.get(siguiente);
    }
    cadena_digito[indice] = '\0';
    n = atoi(cadena_digito);
}
//Usa iostream:
void nueva_linea( )
{
using namespace std;
```

<El resto de la definición de `nueva_linea` se proporciona en el cuadro 5.7.>

Diálogo de ejemplo

```
Escriba un entero y oprima intro: $ 100
Esa cadena se convierte en el entero 100
De nuevo? (si/no): si
Escriba un entero y oprima intro: 100
Esa cadena se convierte en el entero 100
De nuevo? (si/no): si
Escriba un entero y oprima intro: 99%
Esa cadena se convierte en el entero 99
De nuevo? (si/no): si
Escriba un entero y oprima intro: 23% &&5 *12
Esa cadena se convierte en el entero 23512
De nuevo? (si/no): no
```

CUADRO 11.3 *Función de entrada robusta (parte 1 de 2)*

```
//Programa de demostración para la versión mejorada de obtiene_int.
#include <iostream>
#include <cstdlib>
#include <cctype>

void lee_y_limpia(int& n);
//Lee una línea de entrada. Descarta todos los símbolos excepto los dígitos. Convierte
//la cadena tipo C en un entero y hace que n sea igual al valor de este entero.

void nueva_linea( );
//Descarta toda la entrada restante en la línea de entrada actual.
//También descarta la '\n' al final de la línea.

void obtiene_int(int& numero_entrada);
//Proporciona a numero_entrada un valor que apruebe el usuario.

int main( )
{
    using namespace std;
    int numero_entrada;
    obtiene_int(numero_entrada);
    cout << "Valor final leido = " << numero_entrada << endl;
    return 0;
}

//Usa iostream y lee_y_limpia:
void obtiene_int(int& numero_entrada)
{
    using namespace std;
    char resp;
    do
    {
       cout << "Escriba el numero de entrada: ";
       lee_y_limpia(numero_entrada);
       cout << "Usted escribio " << numero_entrada
            << " Es correcto? (si/no): ";
       cin >> resp;
       nueva_linea( );
    } while ((resp != 's') && (resp != 'S'));
}
```

CUADRO 11.3 *Función de entrada robusta (parte 2 de 2)*

```
//Usa iostream, cstdlib y cctype:
void lee_y_limpia(int& n)
```

<El resto de la definición de `lee_y_limpia` se proporciona en el cuadro 11.2.>

```
//Usa iostream:
void nueva_linea( )
```

<El resto de la definición de `nueva_linea` se proporciona en el cuadro 11.2.>

Diálogo de ejemplo

```
Escriba el numero de entrada: $57
Usted escribio 57 Es correcto? (si/no): no
Escriba el numero de entrada: $77*5xa
Usted escribio 775 Es correcto? (si/no): no
Escriba el numero de entrada: 77
Usted escribio 77 Es correcto? (si/no): no
Escriba el numero de entrada: $75
Usted escribio 75 Es correcto? (si/no): si
Valor final leido = 75
```

11.2 La clase `string` estándar

Trato de atrapar cada oración, cada palabra que tú y yo decimos, y encierro con rapidez todas esas oraciones y palabras en mi almacén literario porque podrían ser útiles.

Anton Chekhov, *The Seagull*

En la sección 11.1 presentamos las cadenas tipo C. Estas cadenas son sólo arreglos de caracteres que terminan con el carácter nulo `'\0'`. Para poder manipular estas cadenas tipo C, hay que preocuparse por todos los detalles sobre el manejo de arreglos. Por ejemplo, si deseamos agregar caracteres a una cadena tipo C y no hay suficiente espacio en el arreglo, debe crear otro arreglo para almacenar esta cadena más larga de caracteres. En resumen, las cadenas tipo C requieren que el programador lleve la cuenta de todos los detalles de bajo nivel que tratan acerca de cómo se almacenan las cadenas tipo C en la memoria. Esto es mucho trabajo adicional y es además una fuente de errores del programador. El estándar ANSI/ISO más reciente para C++ especifica que este lenguaje ahora debe tener también una clase `string` que permita al programador tratar las cadenas como un tipo de datos bási-

co, sin necesidad de preocuparse por los detalles de implementación. En esta sección presentaremos este tipo `string`.

Introducción a la clase estándar `string`

La clase `string` se define en la biblioteca cuyo nombre también es `<string>`, y las definiciones se colocan en el espacio de nombres `std`. Por lo tanto, para poder usar la clase `string` nuestro código debe contener lo siguiente (o algo que sea más o menos equivalente):

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

La clase `string` nos permite tratar valores y expresiones `string` en forma muy parecida a los valores de un tipo simple. Podemos usar el operador = para asignar un valor a una variable `string` y podemos usar el signo + para concatenar dos cadenas. Por ejemplo, supongamos que `s1`, `s2` y `s3` son objetos de tipo `string` y que tanto `s1` como `s2` tienen valores de cadena. Entonces `s3` puede hacerse igual a la concatenación del valor de cadena en `s1`, seguido por el valor de cadena en `s2`, como se muestra a continuación:

+ realiza la concatenación

```
s3 = s1 + s2;
```

No hay peligro de que `s3` sea demasiado pequeña para este nuevo valor de cadena. Si la suma de las longitudes de `s1` y `s2` excede la capacidad de `s3`, se asigna más espacio de manera automática para `s3`.

Como se vio antes en este capítulo, las cadenas entre comillas en realidad son cadenas tipo C, por lo que literalmente no son de tipo `string`. No obstante, C++ proporciona la conversión de tipo automática de las cadenas entre comillas a valores de tipo `string`. Por ende, podemos utilizar cadenas entre comillas como si fueran valores literales de tipo `string`, y con frecuencia nos referiremos (junto con la mayoría de los programadores) a las cadenas entre comillas como si fueran valores de tipo `string`. Por ejemplo,

```
s3 = "Hola, mama";
```

asigna el valor de la variable `s3` tipo `string` a un objeto `string` con los mismos caracteres que en la cadena tipo C `"Hola, mama!"`.

constructores

La clase `string` tiene un constructor predeterminado que inicializa un objeto `string` con la cadena vacía. La clase `string` también cuenta con un segundo constructor que toma un argumento, el cual es una cadena tipo C estándar y por ende puede ser una cadena entre comillas. Este segundo constructor inicializa el objeto `string` con un valor que representa la misma cadena que su argumento de cadena tipo C. Por ejemplo,

```
string frase;
string sustantivo("hormigas");
```

La primera línea declara la variable de cadena `frase` y la inicializa con la cadena vacía. La segunda línea declara `sustantivo` como de tipo `string` y la inicializa con un valor de cadena equivalente a la cadena tipo C `"hormigas"`. La mayoría de los programadores dirían de manera informal que "`sustantivo` se inicializa con `"hormigas"`", pero en realidad aquí hay una conversión de tipo. La cadena entre comillas `"hormigas"` es una cadena tipo C, no un valor de tipo `string`. La variable `sustantivo` recibe un valor `string` que tie-

ne los mismos caracteres que "hormigas" en el mismo orden que "hormigas", pero el valor string no se termina con el carácter nulo '\0'. De hecho y al menos en teoría, no se sabe ni importa si el valor string de sustantivo se guarda en un arreglo o no, a diferencia de cualquier otra estructura de datos.

Hay una notación alterna para declarar una variable string e invocar a un constructor. Las siguientes dos líneas son equivalentes de manera exacta:

```
string sustantivo("hormigas");
string sustantivo = "hormigas";
```

En el cuadro 11.4 ilustramos estos detalles básicos sobre la clase string. Observe que, como se muestra en este cuadro, se pueden mostrar valores string mediante el operador <<.

Considere la siguiente línea del cuadro 11.4:

```
frase = "Me encantan las " + sustantivo + " " + adjetivo + "!";
```

conversión de constantes de cadena tipo C al tipo string

C++ debe trabajar mucho para permitirnos concatenar cadenas en esta forma simple y natural. La constante de cadena "Me encantan las " no es un objeto de tipo string. Una constante de cadena como "Me encantan las " se almacena como una cadena

CUADRO 11.4 *Programa que utiliza la clase* string

```
//Demuestra el uso de la clase estándar string.
#include <iostream>
#include <string>
using namespace std;

int main( )
{
    string frase;
    string adjetivo("fritas"), sustantivo("hormigas");
    string deseo = "Provecho!";

    frase = "Me encantan las " + sustantivo + " " + adjetivo + "!";
    cout << frase << endl
         << deseo << endl;

    return 0;
}
```

Diálogo de ejemplo

```
Me encantan las hormigas fritas!
Provecho!
```

tipo C (en otras palabras, como un arreglo de caracteres con terminación nula). Cuando C++ ve "Me encantan las " como un argumento para +, busca la definición (o sobrecarga) de + que se aplique a un valor tal como "Me encantan las ". Hay sobrecargas del operador + que tienen una cadena tipo C a la izquierda y un valor string a la derecha, así como el inverso de estas posiciones. Hay incluso una versión que tiene una cadena tipo C en ambos lados del + y que produce un objeto string como el valor devuelto. Desde luego que también está la sobrecarga que esperamos, con el tipo string para ambos operandos.

En realidad C++ no necesita proporcionar todos esos casos de sobrecarga para +. Si no se proporcionaran estas sobrecargas, C++ buscaría un constructor que pudiera realizar una conversión de tipos para convertir la cadena tipo C "Me encantan las " en un valor que se pudiera aplicar a +. En este caso, el constructor que tiene sólo un parámetro de cadena tipo C realizaría ese tipo de conversión. No obstante, las sobrecargas adicionales son mucho más eficientes.

A menudo la clase string se considera como un sustituto moderno para las cadenas tipo C. Sin embargo, en C++ no se puede evitar con facilidad el uso de cadenas tipo C cuando se programa con la clase string.

La clase `string`

Esta clase puede utilizarse para representar valores que son cadenas de caracteres. La clase string proporciona una representación de las cadenas más versátil que las cadenas tipo C que vimos en la sección 11.1.

La clase string se define en la biblioteca que también se llama <string>, y su definición se coloca en el espacio de nombres std. Así, los programas que utilicen la clase string deberán contener lo siguiente (o algo más o menos equivalente):

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

La clase string tiene un constructor predeterminado que inicializa el objeto string con la cadena vacía y un constructor que toma una cadena tipo C como argumento e inicializa el objeto string con un valor que representa a la cadena que se proporciona como argumento. Por ejemplo:

```
string s1, s2("Hola");
```

E/S con la clase `string`

Puede utilizar el operador de inserción << y cout para mostrar objetos string de igual forma que con los datos de otros tipos. Esto se muestra en el cuadro 11.4. La entrada con la clase string es un poco más sutil.

El operador de extracción >> y cin funcionan igual para los objetos string que para los demás datos, pero hay que recordar que el operador de extracción ignora el espacio en blanco inicial y deja de leer cuando encuentra más espacio en blanco. Esto se aplica tanto para las cadenas como para los demás tipos de datos. Por ejemplo, considere el siguiente código:

```
string s1, s2;
cin >> s1;
cin >> s2;
```

Si el usuario escribe

```
Que el vello de los dedos de tus pies crezca largo y rizado!
```

entonces `s1` recibirá el valor `"Que"` y se eliminará todo el espacio en blanco antes (o después) de esta palabra. La variable `s2` recibirá la cadena `"el"`. Si se utiliza el operador de extracción `>>` y `cin`, sólo se puede leer en palabras; no podemos leer una línea u otra cadena que contenga un espacio en blanco. Algunas veces esto es lo que queremos, pero otras es todo lo contrario.

Si desea que su programa lea toda una línea de entrada en una variable de tipo `string`, puede utilizar la función `getline`. La sintaxis para usar `getline` con objetos `string` es un poco distinta de lo que describimos para las cadenas tipo C en la sección 11.1. En vez de usar `cin.getline;` hay que hacer de `cin` el primer argumento para `getline`.[2] (Por lo tanto, esta versión de `getline` no es una función miembro.)

getline

```
string linea;
cout << "Escriba una linea de entrada:\n";
getline(cin, linea);
cout << linea << "FIN DE SALIDA\n";
```

Cuando se integra en un programa completo, este código produce un diálogo como el siguiente:

```
Escriba una linea de entrada:
Do be do para ti!
Do be do para ti!FIN DE SALIDA
```

Si hubiera espacios en blanco a la izquierda o a la derecha en la línea, entonces también serían parte del valor de cadena que lee `getline`. Esta versión de `getline` está en la biblioteca `<string>`. Puede utilizar un objeto de flujo conectado a un archivo de texto en vez de `cin` para obtener la entrada desde un archivo mediante `getline`.

No puede utilizar `cin` y `>>` para leer un carácter en blanco. SI desea leer un carácter a la vez puede usar `cin.get`, que vimos en el capítulo 5. La función `cin.get` lee valores de tipo *`char`*, no de tipo `string`, pero puede ser útil cuando se maneja entrada tipo `string`. El cuadro 11.5 contiene un programa que ilustra el uso de `getline` y de `cin.get` para introducir valores `string`. El significado de la función `nueva_linea` se explica en la subsección Riesgo titulada “Mezcla de `cin >> variable;` y `getline`.”

[2] Esto es un poco irónico, ya que la clase `string` se diseñó en base a técnicas orientadas a objetos más modernas, y la notación que utiliza para `getline` es la antigua, menos orientada a objetos. Esto es un accidente de la historia. Esta función `getline` se definió después de que ya se estaba utilizando la bibilioteca `iostream`, por lo que los diseñadores no tuvieron otra opción más que hacer de `getline` una función independiente.

CUADRO 11.5 ***Programa que utiliza la clase*** `string` ***(parte 1 de 2)***

```
//Demuestra el uso de getline y de cin.get.
#include <iostream>
#include <string>

void nueva_linea( );

int main( )
{
    using namespace std;

    string primer_nombre, apellido, nombre_registro;
    string lema = "Sus registros son nuestros registros.";

    cout << "Escriba su primer nombre y apellido:\n";
    cin >> primer_nombre >> apellido;
    nueva_linea( );

    nombre_registro = apellido + ", " + primer_nombre;
    cout << "Su nombre en nuestros registros es: ";
    cout << nombre_registro << endl;

    cout << "Nuestro lema es\n"
         << lema << endl;
    cout << "Por favor sugiera un mejor lema (una linea):\n";
    getline(cin, lema);
    cout << "Nuestro nuevo lema sera:\n";
    cout << lema << endl;

    return 0;
}

//Usa iostream:
void nueva_linea( )
{
    using namespace std;

    char siguiente_char;
    do
    {
        cin.get(siguiente_char);
    } while (siguiente_char != '\n');
}
```

CUADRO 11.5 *Programa que utiliza la clase* `string` *(parte 2 de 2)*

Diálogo de ejemplo

```
Escriba su primer nombre y apellido:
  B'Elanna Torres
Su nombre en nuestros registros es: Torres, B'Elanna
Nuestro lema es
Sus registros son nuestros registros.
Por favor sugiera un mejor lema (una linea):
Nuestros registros llegan donde ninguno lo ha hecho antes
Nuestro nuevo lema sera:
Nuestros registros llegan donde ninguno lo ha hecho antes
```

E/S con objetos `string`

Puede utilizar el operador de inserción << con `cout` para mostrar objetos `string`. Puede introducir un objeto `string` mediante el operador de extracción >> y `cin`. Cuando se utiliza >> para introducir datos, el código lee una cadena delimitada con espacio en blanco. Puede usar la función `getline` para introducir toda una línea de texto en un objeto `string`.

Ejemplos

```
string saludo("Hola"), respuesta, siguiente_palabra;
cout << saludo << endl;
getline(cin, respuesta);
cin >> siguiente_palabra;
```

Ejercicios de AUTOEVALUACIÓN

15. Considere el siguiente código (y suponga que está integrado en un programa completo y correcto que después se ejecuta):

```
string s1, s2;
cout << "Escriba una linea de entrada:\n";
cin >> s1 >> s2;
cout << s1 << "*" << s2 << "<FIN DE SALIDA";
```

Si el diálogo empieza como se muestra a continuación, ¿cuál será la siguiente línea de salida?

```
Escriba una línea de entrada:
Una cadena es una alegria eterna!
```

16. Considere el siguiente código (y suponga que está integrado en un programa completo y correcto que después se ejecuta):

```
string s;
cout << "Escriba una linea de entrada:\n";
getline(cin, s);
cout << s << "<FIN DE SALIDA";
```

Si el diálogo comienza como se muestra a continuación, ¿cuál será la siguiente línea de salida?

```
Escriba una linea de entrada:
Una cadena es una alegria eterna!
```

TIP DE PROGRAMACIÓN

Más versiones de `getline`

Hasta ahora hemos descrito la siguiente manera de utilizar `getline`:

```
string linea;
cout << "Escriba una linea de entrada:\n";
getline(cin, linea);
```

Esta versión deja de leer cuando encuentra el marcador de fin de línea `'\n'`. Hay una versión que nos permite especificar un carácter distinto para usarlo como señal de fin de línea. Por ejemplo, el siguiente código se detendrá cuando encuentre el primer signo de interrogación:

```
string linea;
cout << “Escriba algo de entrada:\n”;
getline(cin, linea, '?');
```

Tiene sentido utilizar `getline` como si fuera una función *`void`*, pero en realidad devuelve una referencia a su primer argumento, el cual es `cin` en el código anterior. Por lo tanto, el siguiente código leerá una línea de texto en `s1` y una cadena de caracteres en `s2` que no sean espacios en blanco:

```
string s1, s2;
getline(cin, s1) >> s2;
```

La invocación `getline(cin, s1)` devuelve una referencia a `cin`, por lo que después de la invocación de `getline`, lo que ocurre después es equivalente a

```
cin >> s2;
```

Este tipo de uso de `getline` parece haber sido diseñado para usarse en un examen de C++ en vez de cumplir alguna necesidad de programación real, pero puede ser útil algunas veces.

`getline` para objetos de la clase `string`

La función `getline` para objetos `string` tiene dos versiones:

```
istream& getline(istream& ins, string& var_str,
                         char delimitador);
```

y

```
istream& getline(istream& ins, string& var_str);
```

La primera versión de esta función lee caracteres desde el objeto `istream` que se proporciona como el primer argumento (siempre será `cin` en este capítulo), e inserta los caracteres en la variable `string` llamada `var_str` hasta que se encuentra una instancia del carácter `delimitador`. Este carácter se extrae de la entrada y se descarta. La segunda versión utiliza `'\n'` como valor predeterminado de `delimitador`; en cualquier otro caso funciona igual.

Estas funciones `getline` devuelven su primer argumento (siempre será `cin` en este capítulo), pero por lo general se utilizan como si fueran funciones *`void`*.

RIESGO *Mezcla de* `cin >> variable`*; y* `getline`

Tenga cuidado al mezclar entrada mediante el uso de `cin >> variable;` con entrada mediante el uso de `getline`. Por ejemplo, considere el siguiente código:

```
int n;
string linea;
cin >> n;
getline(cin, linea);
```

Cuando este código lee la siguiente entrada, es de esperarse que a `n` se le asigne el valor `42` y que a `linea` se le asigne un valor de cadena que represente `"Hola vagabundo."`:

```
42
Hola vagabundo.
```

No obstante, mientras que a `n` sin duda se le asigna el valor de `42`, a `linea` se le asigna la cadena vacía. ¿Qué ocurrió?

Al usar `cin >> n` se omite el espacio en blanco a la izquierda en la entrada, pero se deja el resto de la línea (en este caso sólo `'\n'`) para la siguiente entrada. Una instrucción tal como

```
cin >> n;
```

siempre deja algo en la línea para que la siguiente función `getline` lo lea (aún si sólo es el carácter `'\n'`). En este caso, la función `getline` ve el `'\n'` y deja de leer, por lo que recibe una cadena vacía. Si su programa parece ignorar datos de entrada en forma misteriosa, revise si no ha mezcla-

do estos dos tipos de entrada. Tal vez necesite utilizar la nueva función `nueva_linea` del cuadro 11.5 o la función `ignore` de la biblioteca `iostream`. Por ejemplo:

```
cin.ignore(1000, '\n');
```

Con estos argumentos, una llamada a la función miembro `ignore` leerá y descartará todo el resto de la línea, hasta e incluyendo el `'\n'` (o hasta que descarte 1,000 caracteres si no encuentra el final de la línea después de los 1,000 caracteres).

Puede haber otros problemas desconcertantes en programas que utilicen `cin` con `>>` y con `getline`. Lo que es más, estos problemas pueden aparecer y desaparecer a medida que cambiemos de un compilador de C++ a otro. Cuando todo lo demás falle, o si desea estar seguro de la portabilidad, puede recurrir a la técnica de introducir datos un carácter a la vez mediante el uso de `cin.get`.

Estos problemas pueden ocurrir con cualquiera de las versiones de `getline` que describimos en este capítulo.

Procesamiento de cadenas con la clase `string`

La clase `string` nos permite realizar las mismas operaciones que con las cadenas tipo C que vimos en la sección 11.1, y más.

Podemos acceder a los caracteres en un objeto `string` de la misma forma que se accede a los elementos de un arreglo, por lo que los objetos `string` tienen todas las ventajas de los arreglos de caracteres, además de varias ventajas más que los arreglos no tienen, como el incremento automático de su capacidad.

Si `apellido` es el nombre de un objeto `string`, entonces `apellido[i]` nos da acceso al `i`-ésimo carácter en la cadena representada por `apellido`. Este uso de los corchetes en los arreglos se muestra en el cuadro 11.6.

`length`

En este cuadro también se muestra el uso de la función miembro `length`. Cada objeto `string` tiene una función miembro llamada `length`, la cual no tiene argumentos y devuelve la longitud de la cadena representada por el objeto `string`. Por lo tanto, un objeto `string` no sólo puede utilizarse como un arreglo, sino que la función miembro `length` hace que se comporte como un arreglo lleno en forma parcial que lleva la cuenta de manera automática sobre cuántas posiciones están ocupadas.

Cuando los corchetes de los arreglos se utilizan con un objeto de la clase `string`, no se comprueban los índices ilegales. Si usted utiliza un índice ilegal (es decir, un índice mayor o igual a la longitud de la cadena en el objeto) los resultados serán impredecibles, pero lo más probable es que sean malos. Tal vez sólo obtenga un comportamiento extraño sin ningún mensaje de error que le indique que el problema es un valor de índice ilegal.

Existe una función miembro llamada `at` que comprueba los valores de índices ilegales. En sí, esta función miembro se comporta casi igual que los corchetes, sólo por dos cosas: con la función `at` se utiliza la notación de función, por lo que en vez de `a[i]` se utiliza `a.at(i)`; y la función miembro `at` comprueba si `i` se evalúa como un índice ilegal. Si el valor de `i` en `a.at(i)` es un índice ilegal deberá producirse un mensaje de error en tiempo de ejecución que le indique cuál es el problema. En los dos fragmentos de código de ejemplo siguientes, el acceso que se intenta está fuera de rango, pero aún así es probable que el primero de éstos fragmentos no produzca un mensaje de error, aunque estaría accediendo a una variable indizada no existente:

```
string cad("Mary");
cout << cad[6] << endl;
```

CUADRO 11.6 *Un objeto* `string` *puede comportarse como un arreglo*

```
//Demuestra el uso de un objeto string como si fuera un arreglo.
#include <iostream>
#include <string>
using namespace std;

int main( )
{
    string primer_nombre, apellido;

    cout << "Escriba su primer nombre y apellido:\n";
    cin >> primer_nombre >> apellido;

    cout << "Su apellido se escribe:\n";
    int i;
    for (i = 0; i < apellido.length( ); i++)
    {
        cout << apellido[i] << " ";
        apellido[i] = '-';
    }
    cout << endl;
    for (i = 0; i < apellido.length( ); i++)
        cout << apellido[i] << " "; //Coloca un "-" debajo de cada letra.
    cout << endl;

    cout << "Que pase un buen dia, " << primer_nombre << endl;
    return 0;
}
```

Diálogo de ejemplo

```
Escriba su primer nombre y apellido:
John Crichton
Su apellido se escribe:
C r i c h t o n
- - - - - - - -
Que pase un buen dia, John
```

Sin embargo, el segundo ejemplo hará que el programa termine en forma anormal, de manera que por lo menos se puede saber que algo está mal:

```
string cad("Mary");
cout << cad.at(6) << endl;
```

Es importante que sepa que algunos sistemas proporcionan mensajes de error muy pobres cuando `a.at(i)` tiene un índice `i` ilegal.

Puede modificar un carácter individual en la cadena si asigna un valor *`char`* a la variable indizada, como `cad[i]`. Esto también puede hacerse con la función miembro `at`. Por ejemplo, para modificar el tercer carácter en el objeto `string` llamado `cad`, de manera que sea `'X'`, puede usar cualquiera de los siguientes fragmentos de código:

```
cad.at(2) = 'X';
```

o

```
cad[2] = 'X';
```

Al igual que en un arreglo ordinario de caracteres, las posiciones para los objetos de tipo `string` se indizan comenzando desde `0` por lo que el tercer carácter en un objeto `string` está en la posición de índice `2`.

El cuadro 11.7 muestra una lista parcial de las funciones miembro de la clase `string`. En muchos casos, los objetos de la clase `string` se comportan mejor que las cadenas tipo C que presentamos en la sección 11.1. En especial, el operador `==` en los objetos de la clase `string` devuelve un resultado que corresponde a nuestra noción intuitiva de la igualdad de dos cadenas; es decir, devuelve *`true`* si las dos cadenas contienen los mismos caracteres en el mismo orden y devuelve *`false`* en caso contrario. De manera similar, los operadores de comparación `<`, `>`, `<=`, `>=` comparan objetos `string` mediante el uso del ordenamiento lexicográfico. (Éste es un ordenamiento alfabético que utiliza el orden de los símbolos que se proporcionan en el conjunto de caracteres ASCII del apéndice 3. Si las cadenas consisten sólo de letras y éstas son todas mayúsculas o minúsculas, entonces para este caso el ordenamiento lexicográfico es el mismo que el ordenamiento alfabético común.)

EJEMPLO DE PROGRAMACIÓN
Prueba del palíndromo

Un palíndromo es una cadena que se lee igual tanto de adelante hacia atrás, como de atrás hacia delante. El programa del cuadro 11.8 evalúa una cadena de entrada para ver si es un palíndromo. Nuestra prueba del palíndromo descartará todos los espacios y signos de puntuación, y considerará que las versiones mayúscula y minúscula de una letra son iguales al decidir si una frase es un palíndromo. Algunos ejemplos de palíndromos son:

```
Dábale arroz a la zorra el abad
Anita lava la tina
Somos o no somos
Echele leche
Luz azul
La ruta natural
radar
No traces en ese cartón
Yo haré cera hoy
```

CUADRO 11.7 *Funciones miembro de la clase estándar* `string`

Ejemplo	Comentarios
Constructores	
`string cad;`	Constructor predeterminado que crea el objeto `string` vacío `cad`.
`string cad("ejemplo");`	Crea un objeto `string` con los datos "`ejemplo`".
`string cad(una_cadena);`	Crea un objeto `string` llamado `cad`, el cual es una copia de `una_cadena`; `una_cadena` es un objeto de la clase `string`.
Acceso a los elementos	
`cad[i]`	Devuelve una referencia de lectura/escritura a un carácter en `cad`, en el índice `i`. No comprueba si el índice es ilegal.
`cad.at(i)`	Devuelve una referencia de lectura/escritura al carácter en `cad`, en el índice `i`. Igual que `cad[i]`, pero esta versión comprueba índices ilegales.
`cad.substr(posicion, longitud)`	Devuelve la subcadena del objeto que hace la llamada, empezando en `posicion` y con `longitud` caracteres.
Asignación/modificadores	
`cad1 = cad2;`	Inicializa `cad1` con los datos de `cad2`.
`cad1 += cad2;`	Los datos tipo carácter de `cad2` se concatenan al final de `cad2`.
`cad.emtpy( )`	Devuelve *`true`* si `cad` es una cadena vacía; *`false`* en caso contrario.
`cad1 + cad2`	Devuelve una cadena en la cual se concatenan los datos de `cad2` al final de los datos de `cad1`.
`cad.insert(pos, cad2);`	Inserta `cad2` en `cad`, empezando en la posición `pos`.
`cad.remove(pos, longitud);`	Elimina la subcadena de tamaño `longitud`, empezando en la posición `pos`.
Comparación	
`cad1 == cad2  cad1 != cad2`	Compara la igualdad o desigualdad; devuelve un valor Boleano.
`cad1 < cad2  cad1 > cad2` `cad1 <= cad2  cad1>= cad2`	Cuatro comparaciones. Todas son comparaciones lexicográficas.
Búsquedas	
`cad.find(cad1)`	Devuelve el índice de la primera ocurrencia de `cad1` en `cad`.
`cad.find(cad1, pos)`	Devuelve el índice de la primera ocurrencia de la cadena `cad1` en `cad`; la búsqueda empieza en la posición `pos`.
`cad.find_first_of(cad1, pos)`	Devuelve el índice de la primera instancia en `cad` de cualquier carácter en `cad1`; la búsqueda empieza en la posición `pos`.
`cad.find_first_not_of(cad1, pos)`	Devuelve el índice de la primera instancia en `cad` de cualquier carácter que no esté en `cad1`; la búsqueda empieza en la posición `pos`.

CUADRO 11.8 *Programa de prueba del palíndromo (parte 1 de 3)*

```
//Prueba la propiedad de un palíndromo.
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

void intercambia(char& v1, char& v2);
//Intercambia los valores de v1 y v2.

string inverso(const string& s);
//Devuelve una copia de s pero con los caracteres en orden inverso.

string elimina_punt(const string& s, const string& punt);
//Devuelve una copia de s en la que se elimina cualquier ocurrencia
//de caracteres en la cadena punt.

string hacer_minusculas(const string& s);
//Devuelve una copia de s en la que se cambian todos los caracteres
//de mayúsculas a minúsculas, los demás caracteres no se cambian.

bool es_pal(const string& s);
//Devuelve verdadero si s es un palíndromo, falso si no lo es.

int main( )
{
     string cad;
     cout << "Escriba un candidato para la prueba del palindromo,\n"
          << "al terminar oprima Intro.\n";
     getline(cin, cad);

     if (es_pal(cad))
          cout << """ << cad + """ es un palindromo.";
     else
          cout << """ << cad + """ no es un palindromo.";
     cout << endl;

     return 0;
}

void intercambia(char& v1, char& v2)
{
     char temp = v1;
     v1 = v2;
     v2 = temp;
}
```

CUADRO 11.8 *Programa de prueba del palíndromo (parte 2 de 3)*

```
string inverso(const string& s)
{
     int inicio = 0;
     int final = s.length( );
     string temp(s);

     while (inicio < final)
     {
        final–;
        intercambia(temp[inicio], temp[final]);
        inicio++;
     }

     return temp;
}


//Usa <cctype> y <string>
string hacer_minusculas(const string& s)
{
     string temp(s);
     for (int i = 0; i < s.length( ); i++)
         temp[i] = tolower(s[i]);

     return temp;
}

string elimina_punt(const string& s, const string& punt)
{
     string no_punt; //se inicializa con la cadena vacía
     int longitud_s = s.length( );
     int longitud_punt = punt.length( );

     for (int i = 0; i < longitud_s; i++)
     {
          string un_car = s.substr(i,1); //Una cadena de un carácter
          int posicion = punt.find(un_car, 0);
          //Encuentra la posición de los caracteres sucesivos
          //de s en punt.

       if (posicion < 0 || posicion >= longitud_punt)
          no_punt = no_punt + un_car; //un_car no está en punt, por lo que se guarda
     }

     return no_punt;
}
```

CUADRO 11.8 *Programa de prueba del palíndromo (parte 3 de 3)*

```
//utiliza las funciones hacer_minusculas, elimina_punt.
bool es_pal(const string& s)
{
    string punt(",;:.?!'"" "); //incluye un espacio en blanco
    string cad(s);
    cad = hacer_minusculas(cad);
    string cad_inferior = elimina_punt(cad, punt);

    return (cad_inferior == inverso(cad_inferior));
}
```

Diálogo de ejemplo 1

```
Escriba un candidato para la prueba del palindromo,
al terminar oprima Intro.
Somos, o no somos.
"Somos, o no somos." es un palindromo.
```

Diálogo de ejemplo 2

```
Escriba un candidato para la prueba del palindromo,
al terminar oprima Intro.
Radar
"Radar" es un palindromo.
```

Diálogo de ejemplo 3

```
Escriba un candidato para la prueba del palindromo,
al terminar oprima Intro.
Soy un palindromo?
"Soy un palindromo?" no es un palindromo.
```

= y == son distintos para los objetos `string` y las cadenas tipo C

Cuando los operadores =, ==, !=, <, >, <=, >= se usan con el tipo `string` estándar de C++, producen resultados que corresponden a nuestra noción intuitiva de la manera en que se comparan las cadenas. No se comportan en forma errónea como con las cadenas tipo C, como vimos en la sección 11.1.

La función `elimina_punt` es interesante en cuanto a que utiliza las funciones miembro `substr` y `find` de `string`. La función miembro `substr` extrae una subcadena del objeto que hace la llamada, dadas la posición y la longitud de la subcadena deseada. Las primeras tres líneas de `elimina_punt` declaran las variables que se van a usar en la función. El ciclo *for* pasa por los caracteres del parámetro `s` uno a la vez y trata de buscarlos mediante `find` en la cadena `punt`. Para hacer esto se extrae en cada posición de carácter un objeto `string` que sea subcadena de `s` con longitud `1`. La posición de esta subcadena en la cadena `punt` se determina mediante el uso de la función miembro `find`. Si este objeto `string` de un carácter no se encuentra en la cadena `punt`, entonces el objeto `string` de un carácter se concatena con la cadena `no_punt` que se va a devolver.

Ejercicios de AUTOEVALUACIÓN

17. Considere el siguiente código:

```
string s1, s2("Hola");
cout << "Escriba una linea de entrada:\n";
cin >> s1;
if (s1 == s2)
   cout << "Es igual\n";
else
   cout << "No es igual\n";
```

Si el diálogo empieza como se muestra a continuación, ¿cuál será la siguiente línea de salida?

```
Escriba una linea de entrada:
Hola amigo!
```

18. ¿Cuál es la salida que produce el siguiente código?

```
string s1, s2("Hola");
s1 = s2;
s2[0] = 'L';
cout << s1 << " " << s2;
```

Conversión entre objetos `string` y cadenas tipo C

Ya hemos visto que C++ realiza una conversión de tipo automática para permitirnos almacenar una cadena tipo C en una variable de tipo `string`. Por ejemplo, el siguiente fragmento de código funciona sin problemas:

```
char una_cadena_c[] = "Esta es mi cadena tipo C.";
string variable_string;
variable_string = una_cadena_c;
```

No obstante, el siguiente código producirá un mensaje de error del compilador:

```
una_cadena_c = variable_string; //ILEGAL
```

Lo siguiente también es ilegal:

```
Strcpy(una_cadena_c, variable_string); //ILEGAL
```

`strcpy` no puede tomar un objeto `string` como su segundo argumento; además no hay conversión automática de objetos `string` a cadenas tipo C, lo cual parece ser el problema que no podemos evitar.

`c_str( )`

Para obtener la cadena tipo C correspondiente a un objeto `string`, debemos realizar una conversión explícita. Esto puede hacerse con la función miembro `c_str( )` de la clase `string`. La versión correcta de la operación de copia que hemos tratado de realizar es la siguiente:

```
strcpy(una_cadena_c, variable_string.c_str( )); //Legal;
```

Observe que necesitamos usar la función `strcpy` para realizar la operación de copia. La función miembro `c_str( )` devuelve la cadena tipo C que corresponde al objeto `string` que hace la llamada. Como vimos antes en este capítulo, el operador de asignación no trabaja con cadenas tipo C. Por consecuencia y antes de que piense que el siguiente código podría funcionar, debemos recalcar que también es ilegal.

```
una_cadena_c = variable_string.c_str( ); //ILEGAL
```

11.3 Vectores

> *"Bueno, lo comeré" dijo Alicia, "y si me hace más grande, podré alcanzar la llave; si me hace más pequeña, podré meterme debajo de la puerta, así que de cualquier forma entraré al jardín..."*
>
> Lewis Caroll, *Alicia en el país de las maravillas*

Podemos considerar a los vectores como arreglos que pueden aumentar (y reducir) su longitud mientras nuestro programa se está ejecutando. En C++, una vez que nuestro programa crea un arreglo no se puede modificar la longitud del mismo. Los vectores tienen la misma función que los arreglos, con la excepción de que pueden modificar su longitud mientras el programa se está ejecutando. Los vectores forman parte de una biblioteca estándar de C++ conocida como STL (Biblioteca de plantillas estándar), la cual veremos con más detalle en el capítulo 18.

No necesita leer las secciones anteriores de este capítulo para poder cubrir esta sección.

Fundamentos de vectores

Al igual que un arreglo, un vector tiene un tipo base y almacena una colección de valores de su tipo base. No obstante, la sintaxis para la declaración de un tipo de vector y de una variable de vector es distinta de la sintaxis para los arreglos.

declaración de una variable tipo vector

La declaración de una variable `v` para un vector con el tipo base *`int`* es la siguiente:

```
vector<int> v;
```

clase de plantilla

La notación `vector<Tipo_base>` es una **clase de plantilla**, lo cual significa que puede utilizar cualquier tipo para *`Tipo_base`* y se producirá una clase para los vectores con ese tipo de base. Podemos considerar esto como si se especificara el tipo base para un vector de la misma forma en que se especifica un tipo base para un arreglo. Puede usar cualquier tipo (incluso tipos de clases) como tipo base para un vector. La notación `vector<int>` es un nombre de clase, por lo que la declaración anterior de `v` como vector de tipo `vector<int>` incluye una llamada al constructor predeterminado para la clase `vector<int>`, el cual crea un objeto vector que está vacío (no tiene elementos).

`v[i]`

Los elementos de un vector se indizan empezando desde 0, al igual que los arreglos. Puede usarse la notación de corchetes de los arreglos para leer o modificar estos elementos, al igual que en un arreglo. Por ejemplo, el siguiente código modifica el valor del `i`-ésimo elemento del vector `v` y después muestra ese valor modificado. (`i` es una variable *`int`*.)

```
v[i] = 42;
cout << "La respuesta es " << v[i];
```

No obstante, existe una restricción en cuanto a este uso de la notación de los corchetes con los vectores, la cual es distinta a la misma notación que se utiliza con los arreglos. Podemos usar `v[i]` para modificar el valor del `i`-ésimo elemento, pero no podemos inicializar el `i`-ésimo elemento mediante el uso de `v[i]`; sólo podemos modificar un elemento que ya haya recibido algún valor. Para agregar un elemento a una posición de índice de un vector por primera vez, por lo general se utiliza la función miembro `push_back`.

`push_back`

Los elementos de un vector se agregan en orden de posiciones; primero en la posición `0`, después en la posición `1`, luego `2`, y así sucesivamente. La función miembro `push_back` agrega un elemento en la siguiente posición disponible. Por ejemplo, el siguiente código proporciona valores iniciales a los elementos `0`, `1` y `2` del vector llamado `ejemplo`:

```
vector<double> ejemplo;
ejemplo.push_back(0.0);
ejemplo.push_back(1.1);
ejemplo.push_back(2.2);
```

size

El número de elementos en un vector se conoce como el **tamaño** del vector. La función miembro `size` puede utilizarse para determinar cuántos elementos hay en un vector. Por ejemplo, una vez que se ejecuta el código anterior, `ejemplo.size( )` devuelve `3`. Podemos escribir todos los elementos que tiene el vector `ejemplo` en cualquier momento, como se muestra a continuación:

```
for (int i = 0; i < ejemplo.size( ); i++)
    cout << muestra[i] << endl;
```

tamaño *`unsigned int`*

La función `size` devuelve un valor de tipo *`unsigned int`*, no un valor de tipo *`int`*. (El tipo *`unsigned int`* sólo permite valores enteros no negativos.) Este valor devuelto debe convertirse de manera automática al tipo *`int`* cuando necesita ser de tipo *`int`*, pero

algunos compiladores podrían advertir que se está utilizando un `unsigned int` en donde se requiere un `int`. Si desea estar muy seguro, siempre puede aplicar una conversión de tipos para convertir el `unsigned int` devuelto en un `int` o, en casos como este ciclo `for`, puede utilizar una variable de control de ciclo de tipo `unsigned int`, como se muestra a continuación:

```
for (unsigned int i = 0; i < ejemplo.size( ); i++)
    cout << ejemplo[i] << endl;
```

En el cuadro 11.9 se proporciona una demostración simple en la que se ilustran algunas técnicas básicas de vectores.

Hay un constructor de vectores que toma un argumento tipo entero y que inicializa el número de posiciones que recibe como argumento. Por ejemplo, si declara `v` de la siguiente manera:

```
vector<int> v(10);
```

entonces los primeros diez elementos se inicializan con `0`, por lo que `v.size( )` devolvería `10`. De esta forma podemos establecer el valor del `i`-ésimo elemento, mediante el uso de `v[i]` para valores de i desde el 0 hasta el 9. En especial, podría colocarse el siguiente código después de la declaración:

```
for (unsigned int i = 0; i < 10; i++)
    v[i] = i;
```

Para establecer el `i`-ésimo elemento cuando `i` es mayor o igual a `10`, utilizaríamos `push_back`.

Cuando se utiliza el constructor con un argumento tipo entero, los vectores de números se inicializan con el cero del tipo de número. Si el tipo base del vector es un tipo de clase, se utiliza el constructor predeterminado para la inicialización.

La definición del vector se proporciona en la biblioteca `vector`, la cual lo coloca en el espacio de nombres `std`. Por ende, un archivo que utilice vectores debería incluir lo siguiente (o algo similar):

```
#include <vector>
using namespace std;
```

RIESGO *Uso de corchetes más allá del tamaño del vector*

Si `v` es un vector e `i` es mayor o igual a `v.size( )`, entonces el elemento `v[i]` no existe todavía y necesita crearse mediante el uso de `push_back` para agregar elementos hasta, e incluyendo la posición `i`. Si trata de establecer `v[i]` para `i` mayor o igual que `v.size( )`, como en

```
v[i] = n;
```

tal vez obtenga un mensaje de error o tal vez no, pero sin duda su programa funcionará de manera incorrecta en cierto momento.

CUADRO 11.9 *Uso de un vector*

```
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
    vector<int> v;
    cout << "Escriba una lista de numeros positivos.\n"
         << "Coloque un numero negativo al final.\n";

    int siguiente;
    cin >> siguiente;
    while (siguiente > 0)
    {
        v.push_back(siguiente);
        cout << siguiente << " agregado. ";
        cout << "v.size( ) = " << v.size( ) << endl;
        cin >> siguiente;
    }

    cout << "Usted escribio:\n";
    for (unsigned int i = 0; i < v.size( ); i++)
        cout << v[i] << " ";
    cout << endl;

    return 0;
}
```

Diálogo de ejemplo

```
Escriba una lista de numeros positivos.
Coloque un numero negativo al final.
2 4 6 8 -1
2 agregado. v.size( ) = 1
4 agregado. v.size( ) = 2
6 agregado. v.size( ) = 3
8 agregado. v.size( ) = 4
Usted escribio:
2 4 6 8
```

Vectores

Los vectores se utilizan en forma muy parecida a los arreglos, sólo que un vector no tiene un tamaño fijo. Si necesita más capacidad para almacenar otro elemento, su capacidad se incrementa de manera automática. Los vectores están definidos en la biblioteca `<vector>`, la cual los coloca en el espacio de nombres `std`. Por lo tanto, un archivo que utilice vectores debe incluir lo siguiente (o algo similar):

```
#include <vector>
using namespace std;
```

La clase de vector para un ***Tipo_base*** dado se escribe así: `vector<`***Tipo_base***`>`. Dos ejemplos de declaraciones de vectores son

```
vector<int> v;  //constructor predeterminado
                //que produce un vector vacío.
vector<UnaClase> registro(20);  //constructor de vector
//para UnaClase que inicializa 20 elementos.
```

Para agregar elementos a un vector se utiliza la función miembro `push_back`, como se muestra a continuación:

```
v.push_back(42);
```

Una vez que una posición de elemento recibe su primer elemento, ya sea mediante `push_back` o con la inicialización de un constructor, se podrá acceder a esa posición de elemento mediante el uso de la notación de los corchetes, de igual manera que con el elemento de un arreglo.

TIP DE PROGRAMACIÓN

La asignación de vectores tiene buen comportamiento

Cuando el operador de asignación se usa con vectores, realiza una asignación elemento por elemento en el vector que se encuentra del lado izquierdo del operador de asignación (incrementa la capacidad si es necesario y restablece el tamaño del vector que está del lado izquierdo del operador de asignación). Por lo tanto, si el operador de asignación en el tipo base realiza una copia independiente del elemento del tipo base, entonces el operador de asignación en el vector realizará una copia independiente.

Para que el operador de asignación produzca una copia completamente independiente del vector que está del lado derecho del operador de asignación, se requiere que el operador de asignación en el tipo base realice copias completamente independientes. El operador de asignación en un vector es sólo tan eficiente (o deficiente) como el operador de asignación en su tipo base. (En el capítulo 12 proporcionaremos los detalles acerca de cómo sobrecargar el operador de asignación para las clases que lo requieren.)

Cuestiones de eficiencia

capacidad

En cualquier punto en el tiempo un vector tiene una **capacidad**, la cual es el número de elementos para los cuales tiene asignada la memoria necesaria en ese momento. La función miembro `capacity( )` puede usarse para conocer la capacidad de un vector. No debemos confundir la capacidad de un vector con su tamaño. El *tamaño* es el número de elemen-

tos en un vector, mientras que la *capacidad* es el número de elementos para los cuales se tiene asignada la memoria necesaria. Por lo general, la capacidad es mayor que el tamaño, y la capacidad siempre es mayor o igual que el tamaño.

Cada vez que se agota la capacidad de un vector y requiere más espacio para un miembro adicional, la capacidad se incrementa en forma automática. La cantidad exacta del incremento depende de la implementación, pero siempre permite más capacidad de la que se necesita de manera inmediata. Un esquema de implementación que se utiliza con frecuencia es duplicar la capacidad cada vez que se necesite incrementar. Como el incremento de la capacidad es una tarea compleja, este método de reasignar la capacidad en piezas grandes es más eficiente que asignar muchas piezas pequeñas.

Tamaño y capacidad

El **tamaño** de un vector es el número de elementos que contiene. La **capacidad** de un vector es el número de elementos para los cuales tiene memoria asignada. Para conocer el tamaño y la capacidad de un vector `v` se utilizan las funciones miembro `v.size( )` y `v.capacity( )`.

Podemos ignorar por completo la capacidad de un vector sin que esto afecte el funcionamiento de nuestro programa. No obstante, si hay que tener en cuenta la eficiencia podría ser conveniente manejar la capacidad por su cuenta y no sólo aceptar el comportamiento predeterminado de duplicar la capacidad cada vez que se necesite más. Puede usar la función miembro `reserve` para incrementar en forma explícita la capacidad de un vector. Por ejemplo,

```
v.reserve(32);
```

establece la capacidad a cuando menos 32 elementos, y

```
v.reserve(v.size( ) + 10);
```

establece la capacidad a cuando menos 10 elementos más que el número de elementos actuales en el vector. Podemos depender de `v.reserve` para incrementar la capacidad de un vector, pero si el argumento es menor que la capacidad actual, esto no implica que se vaya a reducir la capacidad del vector.

Puede modificar el tamaño de un vector mediante el uso de la función miembro `resize`. Por ejemplo, el siguiente código cambia el tamaño de un vector a 24 elementos:

```
v.resize(24);
```

Si el tamaño anterior era menor que 24, entonces los nuevos elementos se inicializan como vimos en el caso para el constructor con un argumento tipo entero. Si el tamaño anterior era mayor que 24, entonces se pierden todos los elementos excepto los primeros 24. La capacidad se incrementa en forma automática si es necesario. Mediante el uso de `resize` y `reserve` podemos reducir el tamaño y la capacidad de un vector cuando ya no se requieran algunos elementos o cierta capacidad.

Ejercicios de AUTOEVALUACIÓN

19. ¿Es legal el siguiente programa? Si es así, ¿cuál es la salida?

```
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
    vector<int> v(10);
    int i;

    for (i = 0; i < v.size( ); i++)
        v[i] = i;

    vector<int> copia;
    copia = v;
    v[0] = 42;

    for (i = 0; i < copia.size( ); i++)
        cout << copia[i] << " ";
    cout << endl;

    return 0;
}
```

20. ¿Cuál es la diferencia entre el tamaño y la capacidad de un vector?

Resumen del capítulo

- Una variable tipo C es lo mismo que un arreglo de caracteres, pero se utiliza de una manera un poco distinta. Una variable de cadena utiliza el carácter nulo `'\0'` para marcar el final de la cadena que se almacena en el arreglo.

- Por lo general, las variables de cadena tipo C deben tratarse como arreglos en vez de tratarse como variables simples del tipo que utilizamos para los números y caracteres individuales. En especial, no podemos asignar un valor de cadena tipo C a una variable de cadena tipo C mediante el uso del signo de igual (=) y no podemos comparar los valores en dos variables de cadena tipo C mediante el operador ==. En vez de ello debemos utilizar funciones especiales para cadenas tipo C para realizar estas tareas.

- La biblioteca `<string>` del estándar ANSI/ISO proporciona una clase con muchas herramientas llamada `string`, la cual puede usarse para representar cadenas de caracteres.

- Los objetos de la clase `string` se comportan mejor que las cadenas tipo C. En especial, los operadores de asignación e igualdad (= y ==) tienen su significado intuitivo cuando se utilizan con objetos de la clase `string`.

- Los vectores pueden considerarse como arreglos que pueden aumentar (y disminuir) su longitud mientras un programa se está ejecutando.

Respuestas a los ejercicios de autoevaluación

1. Las siguientes dos líneas son equivalentes entre sí (pero no son equivalentes con ninguna otra):

```
char var_cadena[10] = "Hola";
char var_cadena[10] = {'H', 'o', 'l', 'a', '\0'};
```

Las siguientes dos líneas son equivalentes entre sí (pero no equivalentes con ninguna otra):

```
char var_cadena[5] = "Hola";
char var_cadena[] = "Hola";
```

La siguiente línea no es equivalente con ninguna de las otras:

```
char var_cadena[10] = {'H', 'o', 'l', 'a'};
```

2. `"DoBeDo para ti"`

3. La declaración significa que `var_cadena` sólo tiene espacio para seis caracteres (incluyendo el carácter nulo `'\0'`). La función `strcat` no comprueba que haya espacio para agregar más caracteres a `var_cadena`, por lo que `strcat` escribirá todos los caracteres en la cadena `" y adios."` en la memoria, aún y cuando se requiera más memoria que la asignada a `var_cadena`. Esto significa que se modificará la memoria que no debería modificarse. El efecto neto es impredecible, pero malo.

4. Si `strlen` no estuviera ya definida, podría utiliza la siguiente definición:

```
int strlen(const char cad[])
//Precondición: cad contiene un valor de cadena que se
//termina con '\0'.
//Devuelve el número de caracteres en la cadena cad (sin
//contar '\0').
{
     int indice = 0;
     while (cad[indice] != '\0')
         indice++;
     return indice;
}
```

5. El número máximo de caracteres es de cinco, ya que se necesita la sexta posición para el terminador nulo (`'\0'`).

6. a) 1
   b) 1
   c) 5 (incluyendo el `'\0'`)
   d) 2 (incluyendo el `'\0'`)
   e) 6 (incluyendo el `'\0'`)

7. Porque *no son equivalentes*. La primera coloca el carácter nulo `'\0'` en el arreglo después de los caracteres `'a'`, `'b'` y `'c'`. La segunda sólo asigna las posiciones sucesivas `'a'`, `'b'` y `'c'` pero *no coloca un* `'\0'` *en ninguna parte*.

8.
```
int indice = 0;
while ( nuestra_cadena[indice] != '\0')
{
        nuestra_cadena[indice] = 'X';
        indice++;
}
```

9. a) Si la variable de cadena tipo C no tiene un terminador nulo `'\0'`, el ciclo puede ir más allá de la memoria asignada para la cadena tipo C, con lo que destruiría el contenido de esa memoria. Para proteger la memoria más allá del final del arreglo, modifique la condición *while* como se muestra en (b).

   b) `while ( nuestra_cadena[indice] != '\0' && indice < TAMANIO )`

10.
```
#include <cstring>
//necesario para obtener la declaración de strcpy
...
strcpy(una_cadena, "Hola");
```

11. `Lo hice a mi manera!`

12. La cadena `"Bien, espero."` es demasiado larga para `una_cadena`. Se sobreescribirá una pieza de memoria que no pertenezca al arreglo `una_cadena`.

13. Escriba algo de entrada:

```
El
  tiempo es ahora.
El-tiempoFIN DE SALIDA
```

14. El diálogo completo se muestra a continuación:

```
Escriba una linea de entrada:
Que el vello de los dedos de tus pies crezca largo y ondulado.
Que e<FIN DE SALIDA
```

15. `Una*cadena<FIN DE SALIDA`

16. `Una cadena es una alegria eterna!`

17. El diálogo completo es

```
Escriba una linea de entrada:
Hola amigo!
Es igual
```

    Recuerde que `cin` deja de leer cuando llega a un carácter de espacio en blanco.

18. `Hola Lola`

19. El programa es legal. La salida es

```
0 1 2 3 4 5 6 7 8 9
```

    Observe que si se modifica `v` no se modifica `copia`. Una copia de verdad independiente se realiza mediante la asignación

```
copia = v;
```

20. El tamaño es el número de elementos en un vector, mientras que la capacidad es el número de elementos para los que hay memoria asignada.

Proyectos de programación

1. Escriba un programa que lea una oración de hasta 100 caracteres y que muestre la oración con los espacios corregidos y las letras con su capitalización corregida. En otras palabras, en la oración de salida todas las cadenas de dos o más espacios en blanco deberán comprimirse a un solo espacio en blanco. La oración deberá empezar con una letra mayúscula pero no deberá contener más letras mayúsculas. No se preocupe por los nombres propios; si la primera letra se cambia a minúscula, eso es aceptable. Trate un retorno de línea como si fuera un espacio en blanco, en el sentido en que un retorno de línea y cualquier número de espacios en blanco deberán comprimirse a un solo espacio en blanco. Suponga que la oración termina con un punto y que no contiene más puntos. Por ejemplo, la entrada

   ```
   la Respuesta a la vida, al Universo, y   a todo
   ES 42.
   ```

   deberá producir la siguiente salida:

   ```
   La respuesta a la vida, al universo, y a todo es 42.
   ```

2. 
   Escriba un programa que lea una línea de texto y muestre el número de palabras en la línea y en el número de ocurrencias de cada letra. Defina una palabra como cualquier cadena de letras que se delimita en cada extremo ya sea por espacio en blanco, por un punto, una coma o por el comienzo o el final de la línea. Puede suponer que la entrada consiste sólo de letras, espacio en blanco, comas y puntos. Cuando se muestre el número de letras que aparecen en una línea, asegúrese de contar las versiones mayúscula y minúscula de la letra como si fuera la misma. Muestre las letras en orden alfabético y liste sólo aquellas letras que aparezcan en la línea de entrada. Por ejemplo, la línea de entrada

   ```
   Yo digo Hola.
   ```

   deberá producir una salida similar a la siguiente:

   ```
   3 palabras
   1 a
   1 d
   1 g
   1 h
   1 i
   1 l
   3 o
   1 y
   ```

3. 
   Proporcione la definición de la función que contiene la siguiente declaración de función. Integre su definición en un programa de prueba adecuado.

   ```
   void obtiene_double(double& numero_entrada);
   //Postcondición: numero_entrada recibe un valor
   //que apruebe el usuario
   ```

   Puede suponer que el usuario escribe los datos de entrada en notación de uso común, tal como `23.789`, en vez de usar notación científica. Modele su definición a partir de la definición de la función `get_int` que se proporciona en el cuadro 11.3, para que su función lea la entrada como caracteres, edite la cadena de caracteres y convierta la cadena resultante en un número de tipo *double*. Tendrá que definir una función como `lee_y_limpia` que sea más sofisticada que la del cuadro 11.2, ya que debe tratar con el punto decimal. Este proyecto es bastante sencillo. Para un proyecto más difícil, permita que el usuario introduzca el número ya sea en notación de uso común, como vimos antes, o en notación científica. Su función deberá decidir si la entrada está o no en notación científica; para ello deberá leer la entrada, *no* preguntar al usuario si va a utilizar notación científica o no.

4. Escriba un programa que lea el nombre de una persona en el siguiente formato: primer nombre, después segundo nombre o inicial y luego apellido. El programa deberá entonces mostrar el nombre en el siguiente formato:

```
Apellido, Primer_Nombre Inicial_Segundo_Nombre.
```

Por ejemplo, la entrada

```
Juan Usuario Comun
```

deberá producir la salida:

```
Comun, Juan U.
```

La entrada

```
Juan U. Comun
```

también deberá producir la salida:

Comun, Juan U.

Su programa deberá funcionar igual y deberá colocar un punto después de la inicial del segundo nombre aún y cuando la entrada no contenga un punto. Su programa deberá tener en cuenta a los usuarios que no proporcionen su segundo nombre ni inicial. En ese caso, la salida desde luego no contendrá segundo nombre ni inicial. Por ejemplo, la entrada

```
Juan Usuario
```

deberá producir la salida

```
Usuario, Juan
```

Si va a utilizar cadenas tipo C, suponga que cada nombre tiene un máximo de 20 caracteres. O también puede usar la clase `string`. *Sugerencia:* tal vez sea conveniente que utilice tres variables de cadena en vez de una variable de cadena larga para la entrada. Tal vez sea más sencillo si *no* utiliza `getline`.

5. Escriba un programa que lea una línea de texto y sustituya todas las palabras de cuatro letras con la palabra "amor". Por ejemplo, la cadena de entrada

```
Tengo odio hacia ti, loco!
```

deberá producir la salida

```
Tengo amor hacia ti, amor!
```

Claro que la salida no siempre tendrá sentido. Por ejemplo, la cadena de entrada

```
John corre a su casa.
```

deberá producir la salida

```
Amor corre a su amor.
```

Si la palabra de cuatro letras empieza con una letra mayúscula, deberá sustituirse por `"Amor"` y no por `"amor"`. Sólo necesita comprobar el uso de mayúsculas en la primera letra de una palabra, la cual puede ser cualquier cadena formada por las letras del alfabeto y delimitada en cada extremo por un espacio en blanco, el final de la línea o cualquier otro carácter que no sea una letra. Su programa deberá repetir esta acción hasta que el usuario diga que quiere salir.

6. Escriba un programa que lea una línea de texto y muestre la línea en pantalla, sustituyendo con `'x'` todos los dígitos en todos los números enteros.

   Por ejemplo,

   Entrada:

   ```
   Mi Idusuario es juan17 y mi nip de 4 digitos es 1234, el cual es secreto.
   ```

   Salida:

   ```
   Mi Idusuario es juan17 y mi nip de x digitos es xxxx, el cual es secreto.
   ```

   Observe que si un dígito es parte de una palabra, entonces ese dígito no se cambia por una 'x'. Por ejemplo, observe que `juan17` NO se cambia a `juanxx`. Incluya un ciclo que permita al usuario repetir este cálculo de nuevo hasta que éste diga que desea terminar el programa.

7. Escriba un programa que pueda usarse para capacitar al usuario para que use un lenguaje menos sexista, mediante la sugerencia de versiones alternativas de oraciones proporcionadas por el usuario. El programa pedirá una oración, leerá la oración en una variable `string` y sustituirá todas las ocurrencias de los pronombres masculinos con pronombres de género neutral. Por ejemplo, sustituirá `"el"` con `"el o ella"`. Por ende, la oración de entrada

   ```
   Vea a un consultor, hable con el y escuche lo que el le quiera decir.
   ```

   deberá producir la siguiente versión de la oración con las modificaciones sugeridas:

   ```
   Vea a un consultor, hable con el o ella y escuche lo que el o ella le quiera decir.
   ```

   Asegúrese de preservar las letras mayúsculas para la primera palabra de la oración. Permita al usuario repetir esto para más oraciones, hasta que diga que ya quiere terminar.

   Éste será un programa largo que requiere de mucha paciencia. Su programa no deberá sustituir la cadena `"el"` cuando ocurra dentro de otra palabra, como `"elemento"`. Una palabra es cualquier cadena formada por las letras del alfabeto y delimitada en cada extremo por un espacio en blanco, por el final de la línea o por cualquier otro carácter que no sea una letra. Las oraciones podrán tener hasta 100 caracteres de longitud.

8. Escriba una función para ordenar que sea similar al cuadro 10.12 en el capítulo 10, con la excepción de que tenga un argumento para un vector de valores *`int`* en vez de un arreglo. Esta función no requerirá un parámetro tal como `numero_usado` como en el cuadro 10.12, ya que un vector puede determinar el número que se va a utilizar con la función miembro `size( )`. Esta función de ordenamiento sólo tendrá este parámetro, que será de un tipo vector. Use el algoritmo de ordenamiento por selección (que se utilizó en el cuadro 10.12).

9. Vuelva a hacer el proyecto de programación 6 del capítulo 10, pero esta vez utilice vectores en vez de arreglos. (Tal vez sea útil hacer primero el proyecto de programación anterior.)

10. Vuelva a hacer el proyecto de programación 5 del capítulo 10, pero esta vez utilice vectores en vez de arreglos. Es conveniente que vuelva a hacer el proyecto de programación 8 o 9 antes de hacer éste. No obstante, tendrá que escribir su propio código de ordenamiento (similar) para este proyecto en vez de utilizar la función de ordenamiento del proyecto de programación 7 u 8 sin modificaciones. Tal vez quiera usar un vector con el tipo *`struct`* como tipo base.

12 Apuntadores y arreglos dinámicos

12 Apuntadores y arreglos dinámicos

La memoria es necesaria para todas las operaciones de la razón.

BLAISE PASCAL, *Pensées*

Introducción

Un *apuntador* es una construcción que nos da más control sobre la memoria de la computadora. En este capítulo veremos cómo se usan apuntadores con arreglos y presentaremos una nueva forma de arreglo llamada *arreglo dinámico*. Los arreglos dinámicos son arreglos cuyo tamaño se determina durante la ejecución del programa, en lugar de fijarse cuando se escribe el programa.

Prerrequisitos

La sección 12.1, que cubre aspectos básicos de los apuntadores, utiliza material de los capítulos 2 al 7. Esta sección no requiere ningún material de los capítulos 8, 9, 10 u 11. Particularmente, no empleará arreglos y utilizará lo básico de las estructuras y clases.

La sección 12.2, que habla acerca de arreglos dinámicos, utiliza material de la sección 12.1, de los capítulos 2 al 7 y del capítulo 10. Esta sección no requiere ningún material de los capítulos 8, 9 u 11; en particular, sólo utilizará lo básico de las estructuras y las clases.

La sección 12.3, que habla acerca de las relaciones entre arreglos dinámicos y clases, así como de algunas propiedades dinámicas de las clases, utiliza material de las secciones 12.1 y 12.2 así como de los capítulos del 2 al 10 y de las secciones 11.1 y 11.2 del capítulo 11.

12.1 Apuntadores

No confundas el dedo que apunta con la Luna.
REFRÁN ZEN

apuntador

Un **apuntador** es la dirección en memoria de una variable. Recuerde que la memoria de una computadora se divide en posiciones de memoria numeradas (llamadas bytes), y que las variables se implementan como una sucesión de posiciones de memoria adyacentes. Recuerde también que en ocasiones el sistema C++ utiliza estas direcciones de memoria como nombres de las variables. Si una variable se implementa como tres posiciones de memoria, la dirección de la primera de esas posiciones a veces se usa como nombre para esa variable. Por ejemplo, cuando la variable se usa como argumento de llamada por referencia, es esta dirección, no el nombre del identificador de la variable, lo que se pasa a la función invocadora.

Una dirección que se utiliza para nombrar una variable de este modo (dando la dirección de memoria donde la variable inicia) se llama *apuntador* porque podemos pensar que la dirección "apunta" a la variable. La dirección "apunta" a la variable porque la identifica diciendo *dónde* está, en lugar de decir qué nombre tiene. Digamos que una variable está en la posición número 1007; podemos referirnos a ella diciendo "es la variable que está allá, en la posición 1007".

En varias ocasiones hemos empleado apuntadores. Como señalamos en el párrafo anterior, cuando una variable es un argumento de llamada por referencia en una llamada de función, la función recibe esta variable argumento en forma de un apuntador a la variable. Éstos son dos usos importantes de los apuntadores, pero el sistema C++ se encarga de ello automáticamente. En este capítulo mostraremos cómo escribir programas que manipulen apuntadores de cualquier forma que queramos, en lugar de confiar en que el sistema lo haga.

Variables de apuntador

Un apuntador se puede guardar en una variable. Sin embargo, aunque un apuntador es una dirección de memoria y una dirección de memoria es un número, no podemos guardar un apuntador en una variable de tipo `int` o `double`. Una variable que va a contener un apuntador se debe declarar como de tipo apuntador. Por ejemplo, lo que sigue declara `p` como una variable de apuntador que puede contener un apuntador que apunta a una variable de tipo `double`:

declaración de variables de apuntador

```
double *p;
```

La variable `p` puede contener apuntadores a variables de tipo `double`, pero normalmente no puede contener un apuntador a una variable de algún otro tipo, como `int` o `char`. Cada tipo de variable requiere un tipo de apuntador distinto.

En general, si queremos declarar una variable que pueda contener apuntadores a otras variables de un tipo específico, declaramos las variables de apuntador igual que declaramos una variable ordinaria de ese tipo, pero colocamos un asterisco antes del nombre de la variable. Por ejemplo, lo siguiente declara las variables `p1` y `p2` de modo que puedan contener apuntadores a variables de tipo `int`; también se declaran dos variables ordinarias `v1` y `v2` de tipo `int`:

```
int *p1, *p2, v1, v2;
```

Es necesario que haya un asterisco antes de *cada una* de las variables de apuntador. Si omitimos el segundo asterisco en la declaración anterior, `p2` no será una variable de apunta-

dor; será una variable ordinaria de tipo `int`. El asterisco es el mismo símbolo que hemos estado usando para la multiplicación, pero en este contexto tiene un significado totalmente distinto.

Declaraciones de variables de apuntador

Una variable que puede contener apuntadores a otras variables de tipo *Nombre_de_Tipo* se declara del mismo modo que una variable de tipo *Nombre_de_Tipo*, excepto que se antepone un asterisco al nombre de la variable.

Sintaxis:

Nombre_de_Tipo `*`*Nombre_de_Variable1*, `*`*Nombre_de_Variable2*, ...;

Ejemplo:

```
double * apuntador1, * apuntador2;
```

Cuando tratamos apuntadores y variables de apuntador, normalmente hablamos de *apuntar* en lugar de hablar de *direcciones*. Cuando una variable de apuntador, como `p1`, contiene la dirección de una variable, como `v1`, decimos que dicha variable *apunta a la variable* `v1` o *es un apuntador a la variable* `v1`.

Direcciones y números

Un apuntador es una dirección, una dirección es un entero, pero un apuntador no es un entero. Esto no es absurdo, ¡es abstracción! C++ insiste en que usemos un apuntador como una dirección y no como un número. Un apuntador no es un valor de tipo `int` ni de ningún otro tipo numérico. Normalmente no podemos guardar un apuntador en una variable de tipo `int`. Si lo intentamos, casi cualquier compilador de C++ generará un mensaje de error o de advertencia. Además, no podemos efectuar las operaciones aritméticas normales con apuntadores. (Podemos realizar una especie de suma y una especie de resta con apuntadores, pero no son la suma y resta normales con enteros.)

el operador &

Las variables de apuntador, como `p1` y `p2` en nuestro ejemplo de declaración, pueden contener apuntadores a variables como `v1` y `v2`. Podemos usar el operador `&` para determinar la dirección de una variable, y luego podemos asignar esa dirección a una variable de apuntador. Por ejemplo, lo siguiente asigna a la variable `p1` un apuntador que apunta a la variable `v1`:

```
p1 = &v1;
```

el operador *

desreferenciación

Ahora tenemos dos formas de referirnos a `v1`: podemos llamarla `v1` o "la variable a la que `p1` apunta". En C++ la forma de decir "la variable a la que `p1` apunta" es `*p1`. Éste es el mismo asterisco que usamos al declarar `p1`, pero ahora tiene otro significado. Cuando el asterisco se usa de esta manera se le conoce como **operador de desreferenciación**, y decimos que la variable de apuntador está **desreferenciada**.

Si armamos todas estas piezas podemos obtener algunos resultados sorprendentes. Consideremos el siguiente código:

```
v1 = 0;
p1 = &v1;
```

```
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

Este código despliega lo siguiente en la pantalla:

```
42
42
```

En tanto p1 contenga un apuntador que apunte a v1, entonces v1 y *p1 se referirán a la misma variable. Así pues, si asignamos 42 a *p1, también estamos asignando 42 a v1.

El símbolo & que usamos para obtener la dirección de una variable es el mismo símbolo que usamos en una declaración de función para especificar un parámetro de llamada por referencia. Esto no es una coincidencia. Recuerde que un argumento de llamada por referencia se implementa dando la dirección del argumento a la función invocadora. Así pues, estos dos usos del símbolo & son básicamente el mismo. Sin embargo, hay ciertas diferencias pequeñas en la forma de uso, así que los consideraremos dos usos distintos (aunque íntimamente relacionados) del símbolo &.

Los operadores * y &

El operador * antepuesto a una variable de apuntador produce la variable a la que apunta. Cuando se usa de esta forma, el operador * se llama **operador de desreferenciación**.

El operador & antepuesto a una variable ordinaria produce la dirección de esa variable; es decir, produce un apuntador que apunta a la variable. El operador & se llama simplemente **operador de dirección de**.

Por ejemplo, consideremos las declaraciones

```
double *p, v;
```

Lo siguiente establece a p de modo que apunte a la variable v:

```
p= &v;
```

*p produce la variable a la que apunta p, así que después de la asignación anterior *p y v se refieren a la misma variable. Por ejemplo, lo siguiente establece el valor de v a 9.99, aunque nunca se usa explícitamente el nombre v:

```
*p = 9.99;
```

apuntadores en instrucciones de asignación

Podemos asignar el valor de una variable de apuntador a otra variable de apuntador. Esto copia una dirección de una variable de apuntador a otra. Por ejemplo, si p1 todavía está apuntando a v1, lo que siguiente establecerá el valor de p2 de modo que también apunte a v1:

```
p2 = p1;
```

Siempre que no hayamos modificado el valor de v1, lo siguiente también desplegará 42 en la pantalla:

```
cout << *p2;
```

Asegúrese de no confundir

```
p1 = p2;
```

con

```
*p1 = *p2;
```

Cuando añadimos el asterisco, no estamos tratando con los de apuntadores `p1` y `p2`, sino con las variables a las que estos apuntan. Esto se ilustra en el cuadro 12.1.

Puesto que podemos usar un de apuntador para referirnos a una variable, el programa puede manipular variables aunque éstas carezcan de identificadores que las nombren. Podemos usar el operador *new* para crear variables sin identificadores que sean sus nombres. Hacemos referencia a estas variables sin nombre mediante apuntadores. Por ejemplo, lo siguiente crea una nueva variable de tipo *int* y asigna a la variable de apuntador `p1` la dirección de esta nueva variable (es decir, `p1` apunta a esta nueva variable sin nombre):

new

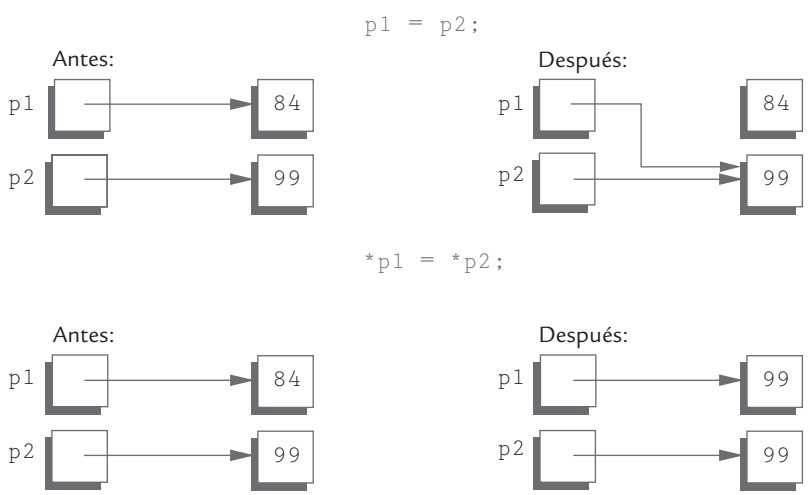
```
p1 = new int;
```

Podemos usar `*p1` para referirnos a esta nueva variable (es decir, la variable a la que `p1` apunta). Podemos hacer con esta variable sin nombre lo mismo que con cualquier otra variable de tipo *int*. Por ejemplo, lo que sigue lee un valor de tipo *int* del teclado y lo coloca en la variable sin nombre, suma `7` al valor y luego despliega el nuevo valor en la pantalla:

```
cin >> *p1;
*p1 = *p1 + 7;
cout << *p1;
```

El operador *new* produce una nueva variable sin nombre y devuelve un apuntador que apunta a esta nueva variable. Especificamos el tipo de la nueva variable escribiendo el nombre

CUADRO 12.1 *Usos del operador de asignación*



variables dinámicas

del tipo después del operador *new*. Las variables que se crean con el operador *new* se llaman **variables dinámicas** porque se crean y se destruyen mientras el programa se está ejecutando. El programa del cuadro 12.2 demuestra algunas operaciones sencillas con apuntadores y variables dinámicas. El cuadro 12.3 ilustra el funcionamiento del programa del cuadro 12.2. En el cuadro 12.3, las variables se representan como cuadritos y el valor de la variable se escribe dentro del cuadrito. No hemos mostrado las direcciones numéricas reales en las variables de apuntadores. Los números reales no son importantes. Lo que es importante es que el número es la dirección de una variable dada. Así pues, en lugar de usar el número real de la dirección, sólo hemos indicado la dirección con una flecha que apunta a la variable que tiene esa dirección. Por ejemplo, en la ilustración (b) del cuadro 12.3, `p1` contiene la dirección de una variable en la que se escribió un signo de interrogación.

Uso de variables de apuntador con =

Si `p1` y `p2` son variables de apuntador, la instrucción

```
p1 = p2;
```

modificará `p1` de modo que apunte a lo mismo que `p2`.

El operador *new*

El operador *new* crea una nueva variable dinámica del tipo que se especifica y devuelve un apuntador que apunta a esta nueva variable. Por ejemplo, lo que sigue crea una variable dinámica nueva del tipo `MiTipo` y deja a la variable del apuntador `p` apuntando a esa nueva variable.

```
MiTipo *p;
p = new MiTipo;
```

Si el tipo es una clase que tiene un constructor, se invoca el constructor predeterminado para la variable dinámica recién creada. Se pueden especificar inicializadores que hagan que se invoquen otros constructores:

```
int *n;
n = new int(17);  //inicializa n con 17
MiTipo *apuntMt;
apuntMt = new MiTipo(32.0, 17);  // invoca MiTipo(double, int);
```

El estándar de C++ estipula que si no hay suficiente memoria desocupada para crear la nueva variable, la acción predeterminada del operador *new* es terminar el programa.[1]

[1]Técnicamente, el operador *new* lanza una excepción que, si no se atrapa, termina el programa. Es posible "atrapar" la excepción o instalar un `controlador new`, pero estos temas rebasan el alcance de este libro.

CUADRO 12.2 *Manipulaciones básicas de apuntadores*

```
//Programa para demostrar apuntadores y variables dinámicas.
#include <iostream>
using namespace std;

int main()
{
    int *p1, *p2;

    p1 = new int;
    *p1 = 42;
    p2 = p1;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    *p2 = 53;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

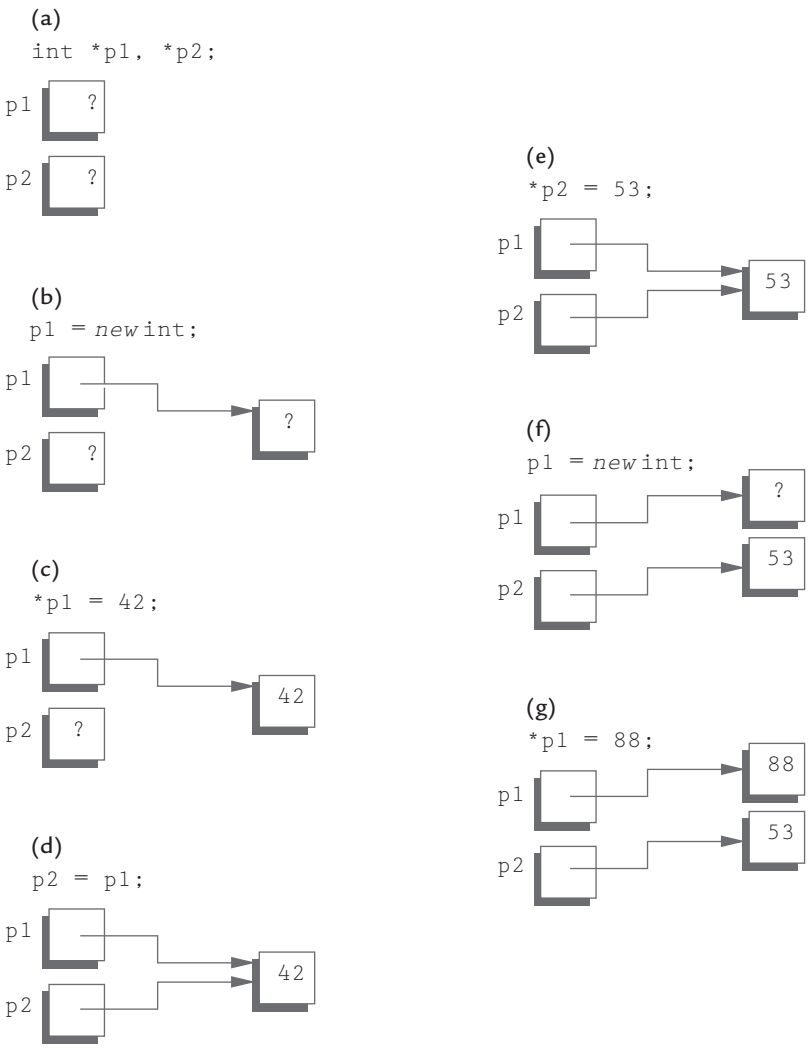
    p1 = new int;
    *p1 = 88;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    cout << "Ojala hayas entendido este ejemplo!\n";
    return 0;
}
```

Diálogo de ejemplo

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Ojala hayas entendido este ejemplo!
```

CUADRO 12.3 *Explicación del cuadro 12.2*



Ejercicios de AUTOEVALUACIÓN

1. Explique el concepto de apuntador en C++.

2. ¿Qué interpretación errónea puede ocurrir con la siguiente declaración?

```
int* apunt_int1, apunt_int2;
```

3. Mencione al menos dos usos del operador *. Indique lo que está haciendo el *, y nombre el uso de * que está presentando.

4. ¿Qué salidas produce el siguiente código?

```
int *p1, *p2;
p1 = new int;
p2 = new int;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
p1 = p2;
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

¿Cómo cambiarían las salidas si se sustituyera

```
*p1 = 30;
```

con lo siguiente?

```
*p2 = 30;
```

5. ¿Qué salidas produce el siguiente código?

```
int *p1 = *p2;
p1 = new int;
p2 = new int;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
*p1 = *p2; //Esto es diferente del ejercicio 4.
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

Administración de memoria básica

Se reserva un área especial de la memoria, llamada **almacén libre**[2] para las variables dinámicas. Toda variable dinámica nueva creada por un programa consume parte de la memo- **almacén libre**

[2]El almacén libre también se conoce como montículo (*heap*).

ria del almacén libre. Si nuestro programa crea demasiadas variables dinámicas, consumirá toda la memoria del almacén libre. Si esto sucede, todas las llamadas subsecuentes a `new` fracasarán.

El tamaño del almacén libre varía de una implementación de C++ a otra. Por lo regular es grande y es poco probable que un programa modesto use toda la memoria del almacén libre. No obstante, incluso en los programas modestos es recomendable reciclar la memoria del almacén libre que ya no se necesite. Si nuestro programa ya no necesita una variable dinámica dada, la memoria ocupada por esa variable puede ser reciclada.

delete

El operador `delete` elimina una variable dinámica y devuelve al almacén libre la memoria que ocupaba, para poder reutilizarla. Supongamos que `p` es una variable de apuntador que está apuntando a una variable dinámica. Lo que sigue destruye la variable dinámica a la que `p` apunta y devuelve al almacén libre la memoria ocupada por esa variable dinámica:

```
delete p;
```

Después de la llamada a `delete`, el valor de `p` es indefinido y `p` será tratada como una variable no inicializada.

El operador *delete*

El operador `delete` elimina una variable dinámica y devuelve al almacén libre la memoria que esa variable ocupaba. Esa memoria puede entonces reutilizarse para crear nuevas variables dinámicas. Por ejemplo, lo que sigue elimina la variable dinámica a la que apunta la variable de apuntador `p`:

```
delete p;
```

Después de una llamada a `delete`, el valor de la variable de apuntador (`p` en nuestro ejemplo) no está definido. (Se usa una versión un poco distinta de `delete`, la cual veremos más adelante, cuando la variable dinámica es un arreglo.)

RIESGO *Apuntadores colgantes*

Apuntadores colgantes

Cuando aplicamos `delete` a una variable de apuntador, la variable dinámica a la que apunta se destruye. En ese momento, el valor de la variable de apuntador queda indefinido, lo que significa que no sabemos a dónde está apuntando ni qué valor hay en el lugar al que está apuntando. Es más, si alguna otra variable de apuntador estaba apuntando a la variable dinámica que se destruyó, esa otra variable de apuntador también quedará indefinida. Estas variables de apuntador no definidas se llaman **apuntadores colgantes**. Si `p` es un apuntador colgante y el programa aplica el operador de desreferenciación `*` a `p` (para producir la expresión `*p`), el resultado es impredecible y por lo regular desastroso. Antes de aplicar el operador de desreferenciación a una variable de apuntador debe asegurarse de que ésta apunta a alguna variable.

Variables estáticas, dinámicas y automáticas

variables dinámicas

Las variables que se crean con el operador *new* se llaman **variables dinámicas** porque se crean y destruyen mientras el programa se está ejecutando. En comparación con estas variables dinámicas, las variables ordinarias parecen estáticas, pero la terminología que usan los programadores en C++ es un poco más complicada, y las variables ordinarias no se denominan *variables estáticas*.

Las variables ordinarias que hemos estado usando en capítulos anteriores no son realmente estáticas. Si una variable es local respecto a una función, el sistema C++ crea la variable cuando se invoca la función, y la destruye cuando se completa la llamada. Puesto que la parte `main` del programa en realidad no es más que una función llamada `main`, esto se cumple incluso para las variables declaradas en la parte `main`. (Puesto que la llamada a `main` no termina hasta que el programa termina, las variables declaradas en `main` no se destruyen hasta que el programa finaliza, pero el mecanismo para manejar variables locales es el mismo para `main` que para cualquier otra función.) Las variables ordinarias que hemos estado usando (es decir, las variables declaradas dentro de `main` o dentro de alguna otra definición de función) se llaman **variables automáticas** porque sus propiedades dinámicas se controlan automáticamente; se crean de manera automática cuando se invoca la función en la que se declararon y se destruyen de igual forma cuando termina la llamada a la función. Por lo regular llamaremos a estas variables **variables ordinarias**, pero otros libros las llaman *variables automáticas*.

variables automáticas

Existe otra catégoria de variables llamada **variables globales**. Éstas son variables que se declaran fuera de cualquier definición de función (incluso fuera de `main`). Tratamos a las variables globales brevemente en el capítulo 3. Da la casualidad que no necesitamos variables globales y por ello no las hemos usado.

variables globales

TIP DE PROGRAMACIÓN

Defina tipos de apuntadores

typedef

Podemos definir un nombre de tipo de apuntador para poder declarar variables de apuntador igual que otras variables, sin tener que colocar un asterisco antes de cada variable de apuntador. Por ejemplo, lo que sigue define un tipo llamado `ApuntInt`, que es el tipo para variables de apuntador que contienen apuntadores a variables *int*:

```
typedef int* ApuntInt;
```

Así pues, estas dos declaraciones de variable de apuntador son equivalentes:

```
ApuntInt p;
```

e

```
int *p;
```

Podemos usar *typedef* para definir un alias para cualquier nombre o definición de tipo. Por ejemplo, lo que se muestra a continuación define el nombre de tipo Kilometros de modo que signifique lo mismo que el nombre de tipo *double*:

```
typedef double Kilometros;
```

Una vez dada esta definición de tipo, podemos definir una variable de tipo *double* así:

```
Kilometros distancia;
```

En ocasiones, este renombramiento de tipos existentes puede ser útil. Sin embargo, nuestro principal uso de *typedef* será en la definición de tipos para variables de apuntador.

El uso de nombres de tipo de apuntador definidos, como ApuntInt, tiene dos ventajas. Primero, evita el error de omitir un asterisco. Recuerde, si queremos que p1 y p2 sean apuntadores, lo siguiente es un error:

```
int *p1, p2;
```

Puesto que omitimos el * de p2, la variable p2 es una variable *int* ordinaria, no una variable de apuntador. Si nos confundimos y colocamos el * en *int*, el problema es el mismo pero es más difícil de detectar. C++ nos permite colocar el * junto al nombre del tipo, como *int*, de modo que lo siguiente es válido:

```
int* p1, p2;
```

Aunque lo anterior es válido, es engañoso. Pareciera que tanto p1 como p2 son variables de apuntador, pero en realidad sólo p1 lo es; p2 es una variable *int* ordinaria. En lo que al compilador de C++ concierne, el * que se anexó a la palabra *int* igualmente podría haberse anexado al identificador p1. Una forma correcta de declarar tanto p1 como p2 como variables de apuntador es

```
int *p1, *p2;
```

Una forma más fácil y menos propensa a errores para declarar tanto p1 como p2 como variables de apuntador es usar el nombre de tipo definido ApuntInt de esta manera:

```
ApuntInt p1, p2;
```

La segunda ventaja de usar un tipo de apuntador definido como ApuntInt se hace evidente cuando definimos una función con un parámetro de llamada por referencia para una variable de apuntador. Sin el nombre de tipo de apuntador definido, necesitaríamos incluir tanto un * como un & en la declaración de la función, y los detalles pueden dar pie a confusión. Si usamos un nombre de tipo para el tipo de apuntador, el uso de un parámetro de llamada por referencia para dicho tipo no tendrá complicaciones. Definiremos un parámetro de llamada por referencia para un tipo de apuntador definido de la misma manera que definimos cualquier otro parámetro de llamada por referencia. He aquí un ejemplo:

```
void funcion_muestra(ApuntInt& variable_apuntador);
```

Definiciones de tipos

Podemos asignar un nombre a una definición de tipo y luego usar el nombre de tipo para declarar variables. Esto se hace con la palabra clave *typedef*. Estas definiciones de tipos normalmente se colocan afuera del cuerpo de la parte `main` del programa (y fuera del cuerpo de otras funciones) en el mismo lugar que las definiciones de `struct` y clases. Usaremos definiciones de tipos para definir nombres de tipos de apuntadores, como se muestra en el siguiente ejemplo:

Sintaxis:

typedef Definicion_de_Tipo_Conocido Nuevo_Nombre_de_Tipo;

Ejemplo:

```
typedef int* ApuntInt;
```

El nombre de tipo `ApuntInt` se puede usar entonces para declarar apuntadores a variables dinámicas de tipo *int*, como en:

```
ApuntInt apuntador1, apuntador2;
```

Ejercicios de AUTOEVALUACIÓN

6. Suponga que se crea una variable dinámica así:

   ```
   char *p;
   p = new char;
   ```

   Suponiendo que el valor de `p` no haya cambiado (o sea, que todavía apunte a la misma variable dinámica), ¿cómo podemos destruir esta nueva variable dinámica y devolver al almacén libre la memoria que ocupa, a fin de poder reutilizarla en la creación de otras variables dinámicas nuevas?

7. Escriba una definición de un tipo llamado `ApuntNumero` que sea el tipo para variables de apuntador que contengan apuntadores a variables dinámicas de tipo *int*. También, escriba una declaración de una variable de apuntador llamada `mi_apunt`, que sea de tipo `ApuntNumero`.

8. Describa la acción del operador *new*. ¿Qué devuelve el operador *new*?

12.2 Arreglos dinámicos

En esta sección veremos que las variables de arreglo son en realidad variables de apuntador. También aprenderemos a escribir programas con arreglos dinámicos. Un **arreglo dinámico** es un arreglo cuyo tamaño no se especifica al momento de escribir el programa, sino durante la ejecución de éste.

arreglo dinámico

Variables de arreglo y variables de apuntador

En el capítulo 10 describimos la forma en que los arreglos se guardan en la memoria. Entonces no habíamos visto los apuntadores, así que tratamos los arreglos en términos de direcciones de memoria. Sin embargo, una dirección de memoria es un apuntador, así que en C++ una variable de arreglo en realidad es una variable de apuntador que apunta a la primera variable indizada del arreglo. Dadas las dos declaraciones siguientes, `p` y `a` son variables de la misma especie:

```
int a[10];
typedef int* ApuntInt;
ApuntInt p;
```

El hecho de que `a` y `p` sean variables de la misma especie se ilustra en el cuadro 12.4. Puesto que `a` es un apuntador que apunta a una variable de tipo *`int`* (a saber, la variable `a[0]`), podemos asignar el valor de `a` a la variable de apuntador `p` así:

```
p = a;
```

Después de esta asignación, `p` apunta a la misma posición de memoria que `a`. Así pues, `p[0]`, `p[1]`, ... `p[9]` se refieren a las variables indizadas `a[0]`, `a[1]`, ... `a[9]`. La notación de corchetes que hemos estado usando con arreglos aplica a las variables de apuntador en tanto éstas apunten a un arreglo en memoria. Después de la asignación anterior, podemos tratar el identificador `p` como si fuera un identificador de arreglo. También podemos tratar al identificador `a` como si fuera una variable de apuntador, con una importante diferencia. *No podemos cambiar el valor de apuntador de una variable de arreglo, como* `a`. Usted podría pensar que lo siguiente es válido, pero no es así:

```
ApuntInt p2;
...//p2 es dado por algún valor de apuntador
a = p2; //NO VÁLIDO. No podemos asignar una dirección distinta a a.
```

Cómo crear y usar arreglos dinámicos

Un problema que presentan los arreglos que hemos usado hasta ahora es que es preciso especificar el tamaño del arreglo en el momento de escribir el programa, y podría ser imposible saber de qué tamaño necesitamos un arreglo antes de ejecutar el programa. Por ejemplo, un arreglo podría contener una lista de números de identificación de estudiantes, pero el tamaño del grupo podría ser distinto cada vez que se ejecuta el programa. Con los tipos de arreglos que hemos usado hasta ahora, tendríamos que estimar el tamaño más grande que podríamos llegar a necesitar para el arreglo, y rezar porque sea suficiente. Esto implica dos problemas. Primero, podríamos hacer un estimado demasiado bajo, y entonces nuestro programa no funcionaría en todas las situaciones. Segundo, dado que el arreglo podría tener muchas posiciones desocupadas, se podría desperdiciar memoria de la computadora. Los arreglos dinámicos evitan estos problemas. Si nuestro programa usa un arreglo dinámico para los números de identificación de los estudiantes, podremos introducir el tamaño del grupo como entrada del programa y crear un arreglo dinámico exactamente de ese tamaño.

creación de un arreglo dinámico

Los arreglos dinámicos se crean con el operador *`new`*. La creación y uso de arreglos dinámicos son sorprendentemente sencillos. Puesto que las variables de arreglo son variables

CUADRO 12.4 *Arreglos y variables de apuntador*

```
//Programa para demostrar que una variable arreglo es una especie de variable
//de apuntador.
#include <iostream>
using namespace std;

typedef int* ApuntInt;

int main()
{
    ApuntInt p;
    int a[10];
    int indice;

    for (indice = 0; indice < 10; indice++)
        a[indice] = indice;
    p = a;

    for (indice = 0; indice < 10; indice++)
        cout << p[indice] << "";
    cout << endl;

    for (indice = 0; indice < 10; indice++)
        p[indice] = p[indice] + 1;

    for (indice = 0; indice < 10; indice++)
        cout << a[indice] << " ";
    cout << endl;

    return 0;
}
```

Observe que los cambios al arreglo `p` *también son cambios al arreglo* `a`.

Salida

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```

de apuntador, podemos usar el operador *new* para crear variables dinámicas que sean arreglos y tratarlas como si fueran arreglos ordinarios. Por ejemplo, lo siguiente crea una variable arreglo dinámica con 10 elementos de tipo *double*.

```
typedef double* ApuntDouble;
ApuntDouble p;
p = new double[10];
```

Si queremos crear un arreglo dinámico de elementos de cualquier otro tipo, simplemente sustituimos *double* por el tipo deseado. En particular, podemos sustituir el tipo *double* por un tipo *struct* o de clase. Si queremos obtener una variable de arreglo dinámico de cualquier otro tamaño, basta con sustituir 10 por el tamaño deseado.

Este ejemplo tiene varios detalles menos obvios que es importante notar. En primer lugar, el tipo que usamos para un apuntador a un arreglo dinámico es el mismo que el tipo de apuntador que usaríamos para un solo elemento del arreglo. Por ejemplo, el tipo de apuntador para un arreglo de elementos de tipo *double* es el mismo que usaríamos para una variable simple de tipo *double*. El apuntador al arreglo en realidad es un apuntador a la primera variable indizada del arreglo. En el ejemplo anterior se crea un arreglo con 10 variables indizadas, y el apuntador p apunta a la primera de esas 10 variables indizadas.

Cómo usar un arreglo dinámico

- *Defina un tipo de apuntador:* Defina un tipo para apuntadores a variables del mismo tipo que los elementos del arreglo. Por ejemplo, si el arreglo dinámico es un arreglo de *double*, podría usar lo siguiente:

  ```
  typedef double* ApuntArregloDouble;
  ```

- *Declare una variable de apuntador:* Declare una variable de apuntador de este tipo definido. Dicha variable apuntará al arreglo dinámico en la memoria y servirá como nombre del arreglo dinámico.

  ```
  ApuntArregloDouble a;
  ```

- *Llame a new:* Cree un arreglo dinámico usando el operador *new*:

  ```
  a = new double[tamanio_arreglo];
  ```

  El tamaño del arreglo dinámico se da entre corchetes como en el ejemplo anterior. Se puede dar el tamaño usando una variable *int* o alguna otra expresión *int*. En el ejemplo anterior, tamanio_arreglo puede ser una variable de tipo *int* cuyo valor se determina mientras el programa se está ejecutando.

- *Utilice como un arreglo ordinario:* La variable de apuntador, digamos a, se usa igual que un arreglo ordinario. Por ejemplo, las variables indizadas se escriben de la forma acostumbrada, a[0], a[1], etcétera. No debemos asignar a dicha variable ningún otro valor de apuntador; debemos usarla como una variable de arreglo.

- *Llame a delete []:* Cuando el programa haya terminado de usar la variable dinámica, use *delete* y unos corchetes vacíos junto con la variable de apuntador para eliminar el arreglo dinámico y devolver al almacén libre la memoria que ocupa, a fin de reutilizarla. Por ejemplo,

  ```
  delete [] a;
  ```

Observe también que, al usar *new*, damos el tamaño del arreglo dinámico en corchetes después del tipo, que en este ejemplo es *double*. Esto le dice a la computadora cuánto espacio debe reservar para el arreglo dinámico. Si omitimos los corchetes y el 10, la computadora sólo asignará suficiente memoria para una variable de tipo *double*, no para un arreglo de 10 variables indizadas de tipo *double*. Como se ilustra en el cuadro 12.5, puede utilizar una variable de tipo *int* en lugar de una constante 10 de manera que el tamaño del arreglo pueda ser leído dentro del programa.

El cuadro 12.5 contiene un programa que ordena una lista de números. Este programa funciona con listas de cualquier tamaño porque usa un arreglo dinámico para contener los números. El tamaño del arreglo se determina cuando se ejecuta el programa. Se pregunta al usuario cuántos números habrá y entonces el operador *new* crea un arreglo dinámico de ese tamaño. El tamaño del arreglo dinámico está dado por la variable tam_arreglo.

delete []

Observe la instrucción *delete* que destruye la variable del arreglo dinámico a en el cuadro 12.5. Puesto que el programa ya estaba a punto de terminar, en realidad no era necesario incluir esta instrucción *delete*; pero si el programa fuera a hacer otras cosas con variables dinámicas es recomendable incluir esa instrucción para que la memoria ocupada por el arreglo dinámico se devuelva al almacén libre. La instrucción *delete* para un arreglo dinámico es similar a la que vimos antes, excepto que en este caso es preciso incluir un par de corchetes, así:

```
delete [] a;
```

Los corchetes le dicen a C++ que se está eliminando una variable de arreglo dinámico, así que el sistema verifica el tamaño del arreglo y elimina ese número de variables indizadas. Si omitimos los corchetes le estaríamos diciendo a la computadora que sólo eliminara una variable de tipo *int*. Por ejemplo,

```
delete a;
```

no es válido, pero ningún compilador que el autor conozca detecta el error. El Estándar ANSI C++ dice que lo que sucede cuando se hace esto "no está definido". Eso significa que el autor del compilador puede decidir que se haga lo que crea más conveniente (para el autor del compilador, no para nosotros). Aunque se haga algo útil, no hay garantía de que la siguiente versión de ese compilador, o cualquier otro compilador que usemos para compilar este código, hará la misma cosa. Moraleja: siempre use la sintaxis

```
delete [] apuntArreglo;
```

al eliminar memoria que se asignó con, por ejemplo,

```
apuntArreglo = new MiTipo[37];
```

Creamos un arreglo dinámico con una llamada a *new* usando un apuntador, como el apuntador a del cuadro 12.5. Después de la llamada a *new*, no deberemos asignar ningún otro valor de apuntador a esta veriable apuntador, pues ello podría confundir al sistema cuando la memoria del arreglo dinámico se devuelva al almacén libre con una llamada a *delete*.

Los arreglos dinámicos se crean usando *new* y una variable de apuntador. Cuando nuestro programa termine de usar un arreglo dinámico, es muy recomendable devolver al almacén libre la memoria que ocupaba, con una llamada a *delete*. Por lo demás, un arreglo dinámico se puede usar como cualquier otro arreglo.

CUADRO 12.5 *Un arreglo dinámico (parte 1 de 2)*

```
//Ordena una lista de números introducidos con el teclado.
#include <iostream>
#include <cstdlib>
#include <cstddef>

typedef int* ApuntArregloInt;

void llenar_arreglo(int a[], int tamanio);
//Precondición: tamanio es el tamaño del arreglo a.
//Postcondición: a[0] hasta a[tamanio-1] se han
//llenado con valores leídos del teclado.

void ordenar(int a[], int tamanio);
//Precondición: tamanio es el tamaño del arreglo a.
//Los elementos a[0] hasta a[tamanio-1] tienen valores.
//Postcondición: Los valores de a[0] a a[tamanio-1] se reacomodaron
//de modo que a[0] <= a[1] <= ... <= a[tamanio-1].

int main()
{
    using namespace std;
    cout << "Este programa ordena numeros de menor a mayor.\n";

    int tamanio_arreglo;
    cout << "Cuantos numeros se ordenaran? ";
    cin >> tamanio_arreglo;

    ApuntArregloInt a;
    a = new int[tamanio_arreglo];

    llenar_arreglo(a, tamanio_arreglo);
    ordenar(a, tamanio_arreglo);
```

Parámetros de arreglos ordinarios.

CUADRO 12.5 *Un arreglo dinámico (parte 2 de 2)*

```
    cout << "De menor a mayor los numeros son:\n";
    for (int indice = 0; indice < tamanio_arreglo; indice++)
        cout << a[indice] << " ";
    cout << endl;

    delete [] a;

    return 0;
}

//Usa la biblioteca iostream:
void llenar_arreglo(int a[], int tamanio)
{
    using namespace std;
    cout << "Escriba " << tamanio << " enteros.\n";
    for (int indice = 0; indice < tamanio; indice++)
        cin >> a[indice];
}

void ordenar(int a[], int tamanio)
```

El arreglo dinámico a *se usa como un arreglo ordinario.*

<Se puede usar cualquier implementación de `ordenar`. Esto podría requerir la definición de funciones adicionales. La implementación ni siquiera necesita saber que `ordenar` se invocará para un arreglo dinámico. Por ejemplo, podemos usar la implementación del cuadro 10.12 (con ajustes apropiados a los nombres de los parámetros).>

Ejercicios de AUTOEVALUACIÓN

9. Escriba una definición de tipo para variables de apuntador que se usarán para apuntar a arreglos dinámicos. Los elementos de los arreglos serán de tipo `char`. Llame al tipo `ArregloChar`.

10. Suponga que su programa contiene código para crear un arreglo dinámico de esta manera:

```
int *dato;
dato = new int[10];
```

de modo que la variable de apuntador `dato` apunte a este arreglo dinámico. Escriba código para llenar este arreglo con 10 números que se introducen desde el teclado.

11. Suponga que su programa contiene código para crear un arreglo dinámico como en el ejercicio anterior, y que no se ha modificado el valor (de apuntador) de la variable de apuntador `dato`. Escriba código que destruya este nuevo arreglo dinámico y devuelva al almacén libre la memoria que ocupa.

12. ¿Qué salida producirá el siguiente código incrustado en un programa completo?

```
int a[10];
int *p = a;
int i;
for (i = 0; i < 10; i++)
   a[i] = i;
for (i = 0; i < 10; i++)
   cout << p[i] << " ";
cout << endl;
```

13. ¿Qué salida producirá el siguiente código incrustado en un programa completo?

```
int tamanio_arreglo = 10;
int *a;
a = new int[tamanio_arreglo];
int *p = a;
int i;
for (i = 0; i < tamanio_arreglo; i++)
   a[i] = i;
p[0] = 10;
for (i = 0; i < tamanio_arreglo; i++)
   cout << a[i] << " ";
cout << endl;
```

Aritmética de apuntadores (*Opcional*)

Existe un tipo de aritmética que podemos efectuar con apuntadores, pero es una aritmética de direcciones, no de números. Por ejemplo, supongamos que nuestro programa contiene el siguiente código:

```
typedef double* ApuntDouble;
ApuntDouble d;
d = new double[10];
```

direcciones, no números

Después de estas instrucciones, `d` contiene la dirección de la variable indizada `d[0]`. La evaluación de la expresión `d + 1` da la dirección de `d[1]`, `d + 2` es la dirección de `d[2]`, etcétera. Observe que aunque el valor de `d` es una dirección y una dirección es un número, `d + 1` no se limita a sumar `1` al número que está en `d`. Si una variable de tipo *double* requiere ocho bytes (ocho posiciones de memoria) y `d` contiene la dirección `2001`, entonces la evaluación de `d + 1` da la dirección de memoria `2009`. Desde luego, podemos sustituir el tipo *double* por cualquier otro tipo, y entonces la suma de apuntadores se efectuará en unidades de variables de ese tipo.

Esta aritmética de apuntadores nos ofrece otra forma de manipular arreglos. Por ejemplo, si `tamanio_arreglo` es el tamaño del arreglo dinámico al que apunta `d`, lo que sigue desplegará en la pantalla el contenido del arreglo dinámico:

```
for (int i = 0; i < tamanio_arreglo; i++)
    cout << *(d + i) << " ";
```

Lo anterior equivale a:

```
for (int i = 0; i < tamanio_arreglo; i++)
    cout << d[i] << " ";
```

No se permite efectuar multiplicación o división con apuntadores. Lo único que puede hacerse es sumar un entero a un apuntador, restar un entero a un apuntador, o restar dos apuntadores del mismo tipo. Cuando restamos dos apuntadores el resultado es el número de variables indizadas que hay entre dos direcciones. Recuerde, para poder restar dos valores de apuntadores los valores deben apuntar al mismo arreglo. No tiene sentido restar un apuntador que apunta a un arreglo a otro apuntador que apunta a un arreglo distinto.

++ y −−

Podemos usar los operadores de incremento y decremento ++ y −−. Por ejemplo, `d++` aumenta el valor de `d` de modo que contenga la dirección de la siguiente variable indizada, y `d−−` modifica `d` de modo que contenga la dirección de la variable indizada anterior.

Ejercicios de AUTOEVALUACIÓN

Estos ejercicios aplican para la sección opcional aritmética de apuntadores.

14. ¿Qué salida producirá el siguiente código incrustado en un programa completo?

```
int tamanio_arreglo = 10;
int *a;
a = new int[tamanio_arreglo];
int i;
for (i = 0; i < tamanio_arreglo; i++)
   *(a + i) = i;
for (i = 0; i < tamanio_arreglo; i++)
   cout << a[i] << " ";
cout << endl;
```

15. ¿Qué salida producirá el siguiente código incrustado en un programa completo?

```
int tamanio_arreglo = 10;
int *a;
a = new int[tamanio_arreglo];
int i;
for (i = 0; i < tamanio_arreglo; i++)
   a[i] = i;
while (*a < 9)
{
    a++;
    cout << *a << " ";
}
cout << endl;
```

Arreglos dinámicos multidimensionales (*Opcional*)

Podemos tener arreglos dinámicos multidimensionales. Sólo recuerde que los arreglos multidimensionales son arreglos de arreglos, o arreglos de arreglos de arreglos, etcétera. Por ejemplo, para crear un arreglo dinámico bidimensional, debemos recordar que éste es un arreglo de arreglos. Para crear un arreglo bidimensional de enteros, debemos crear un arreglo dinámico unidimensional de apuntadores de tipo `int*`, el cual es el tipo para los arreglos unidimensionales de tipo `int`. Después deberemos crear un arreglo dinámico de `int`s para cada variable indizada del arreglo de apuntador.

Una definición de tipo puede ayudar a mantener las cosas en orden. La siguiente variable de tipo es un arreglo dinámico unidimensional ordinario de `int`s:

```
typedef int* ApuntArregloInt;
```

Para obtener un arreglo de enteros de 3-por-4, necesitará un arreglo cuyo tipo base sea `ApuntArregloInt`. Por ejemplo:

```
ApuntArregloInt *m = new ApuntArregloInt[3];
```

Éste es un arreglo de tres apuntadores, cada uno de los cuales puede nombrar a un arreglo dinámico de `int`s, como sigue:

```
for (int i = 0; i < 3; i++)
m[i] = new int[4];
```

El arreglo resultante `m` es un arreglo dinámico de 3 por 4. Un programa simple para ilustrar esto se da en el cuadro 12.6.

`delete[]`

Asegúrese de observar el uso de `delete` en el cuadro 12.6. Dado que el arreglo dinámico m es un arreglo de arreglos, cada uno de los arreglos creados con new `int` en el ciclo `for` debe ser devuelto al controlador del almacén libre con una llamada a `delete[]`; entonces el arreglo `m` debe ser devuelto al almacén libre con otra llamada `delete[]` debe haber una llamada a `delete[]` por cada llamada a new que creó un arreglo. (Dado que el programa termina justo después de las llamadas a `delete[]` podríamos omitir estas llamadas pero queríamos demostrar su uso.)

12.3 Clases y arreglos dinámicos

Con todos los adminículos y recursos de sobra.

William Shakespeare, Rey *Enrique IV, parte III*

Un arreglo dinámico puede tener un tipo base que sea una clase. Una clase puede tener una variable miembro que sea un arreglo dinámico. Podemos combinar las técnicas que aprendimos para las clases y las que aprendimos para los arreglos dinámicos de prácticamente cualquier manera. Hay unas cuantas cosas más de las cuales preocuparse cuando se usan clases y arreglos dinámicos, pero las técnicas básicas son las que ya hemos usado. Comencemos con un ejemplo.

CUADRO 12.6 *Arreglo dinámico bidimensional (parte 1 de 2)*

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

typedef int* ApuntArregloInt;

int main()
{
    int d1, d2;
    cout << "Escriba las dimensiones de las filas y de las columnas del arreglo:\n";
    cin >> d1 >> d2;

    ApuntArregloInt *m = new ApuntArregloInt[d1];
    int i, j;
    for (i = 0; i < d1; i++)
           m[i] = new int[d2];
    //m es ahora arreglos a d1 de d2.

    cout << " Escriba " << d1 << " filas de "
           << d2 << " enteros cada una:\n";
    for (i = 0; i < d1; i++)
           for (j = 0; j < d2; j++)
                  cin >> m[i][j];

    cout << "Valores del arreglo bidimensional:\n";
    for (i = 0; i < d1; i++)
    {
           for (j = 0; j < d2; j++)
                  cout << m[i][j] << " ";
           cout << endl;
    }
```

CUADRO 12.6 *Arreglo dinámico bidimensional (parte 2 de 2)*

```
    for (i = 0; i < d1; i++)
         delete[] m [i];
    delete[] m;

    return 0;
}
```

Observe que debe haber una llamada a `delete[]` *para cada llamada a* `new` *que creó un arreglo. (Estas llamadas a* `delete[]` *en realidad no se necesitan, dado que el programa está finalizando, pero en otro contexto podría ser importante incluirlos.)*

Diálogo de ejemplo

```
Escriba las dimensiones de las filas y de las columnas del arreglo:
3 4
Escriba 3 filas de 4 enteros cada una:
1 2 3 4
5 6 7 8
9 0 1 2
Valores del arreglo bidimensional:
1 2 3 4
5 6 7 8
9 0 1 2
```

EJEMPLO DE PROGRAMACIÓN
Una clase de variables de cadena

En el capítulo 11 explicamos cómo definir variables arreglo para contener cadenas estándar. En la sección anterior aprendimos a definir arreglos dinámicos que permiten determinar el tamaño del arreglo durante la ejecución del programa. En este ejemplo definiremos una clase llamada `VarCadena` cuyos objetos son variables de cadena. Un objeto de la clase `VarCadena` se implementará con un arreglo dinámico cuyo tamaño se determina en el momento de ejecutarse el programa. Así, los objetos de tipo `VarCadena` tienen todas las ventajas de los arreglos dinámicos, pero también tienen algunas características adicionales. Definiremos funciones miembro de `VarCadena` tales que si tratamos de asignar una cadena demasiado larga a un objeto de tipo `VarCadena` se genere un mensaje de error. La versión que definimos aquí sólo proporciona un conjunto pequeño de operaciones para manipular objetos de cadena. En el Proyecto de Programación 1 se pedirá al lector mejorar la definición de la clase añadiendo más funciones miembro y operadores sobrecargados.

Puesto que podríamos usar la clase estándar `string`, como se explica en el capítulo 11, no necesitamos realmente la clase `VarCadena`, pero será un buen ejercicio diseñarla y codificarla.

constructores

En el cuadro 12.7 se da la interfaz del tipo `VarCadena`. Un constructor de la clase `VarCadena` recibe un solo argumento de tipo `int`. Este argumento determina la longitud máxima que puede tener un valor de cadena almacenado en el objeto. Un constructor predeterminado crea un objeto con una longitud máxima permitida de `100`. Otro constructor recibe un argumento arreglo que contiene una de las cadenas estándar de C++ como las que vimos en el capítulo 11. Tenga presente que esto implica que el argumento de este constructor puede ser una cadena entrecomillada. Este constructor inicializa el objeto de modo que pueda contener cualquier cadena cuya longitud sea igual o menor que la longitud de su argumento, e inicializa el valor de cadena del objeto con el valor de su argumento. Por el momento, haremos caso omiso del constructor rotulado *constructor de copia*. También haremos caso omiso de la función miembro llamada `~VarCadena`. Aunque podría parecer un constructor, esta función no lo es. Trataremos estos dos nuevos tipos de funciones miembro en subsecciones posteriores. El significado de las demás funciones miembro de la clase `VarCadena` es obvio.

tamaño de un valor de cadena

En el cuadro 12.8 se da un sencillo programa de demostración. Se declaran dos objetos, `tu_nombre` y `nuestro_nombre`, dentro de la definición de la función `conversacion`. El objeto `tu_nombre` puede contener cualquier cadena que tenga `tam_max_nombre` caracteres de largo, o menos. El objeto `nuestro_nombre` se inicializa con el valor de cadena `"Borg"`, y podemos cambiar su valor a cualquier otra cadena de longitud `4` o menor.

implementación

Como señalamos al principio de esta subsección, la clase `VarCadena` se implementa con un arreglo dinámico. La implementación se muestra en el cuadro 12.9. Cuando se declara un objeto de tipo `VarCadena`, se invoca un constructor para inicializar el objeto. El constructor usa el operador `new` para crear un arreglo dinámico de caracteres nuevo para la variable miembro `valor`. El valor de cadena se guarda en el arreglo `valor` como un valor de cadena ordinario, usando `'\0'` para marcar el final de la cadena. Tenga presente que el tamaño del arreglo no se determina sino hasta que se declara el arreglo, y entonces se invoca el constructor y el argumento del constructor determina el tamaño del arreglo dinámico. Como se indica en el cuadro 12.8, este argumento puede ser una variable de tipo `int`. Examine la declaración del objeto `tu_nombre` en la definición de la función `conversacion`. El argumento del constructor es el parámetro de llamada por valor `tam_max_nombre`. Recuerde que un parámetro de llamada por valor es una variable local, así que `tam_max_nombre` es una variable. Podemos usar cualquier variable `int` como argumento del constructor.

La implementación de las funciones miembro `longitud`, `capturar_linea` y el operador de salida sobrecargado `<<` son sencillas. En las siguientes subsecciones trataremos la función `~VarCadena` y el *constructor de copia*.

Destructores

La variables dinámicas presentan un problema. No desaparecen si nuestro sistema no efectúa una llamada apropiada a `delete`. Incluso si la variable dinámica se creó usando una variable de apuntador local y ésta desaparece al final de una llamada de función, la variable dinámica persistirá a menos que se invoque `delete`. Si no eliminamos las variables dinámicas con `delete`, seguirán ocupando espacio en la memoria, y el programa podría terminar antes de tiempo si se agota toda la memoria del almacén libre. Además, si la variable dinámica está incrustada en la implementación de una clase, el programador que usa la clase no sabrá de la existencia de la variable dinámica y no es razonable esperar que invocará a `delete`. De hecho, dado que los miembros datos normalmente son miembros privados, el programador *no puede* acceder a las variables de apuntador requeridas y por tanto *no puede* invocar a `delete` con estas variables de apuntador. Para

CUADRO 12.7 *Archivo de interfaz para la clase* VarCadena *(parte 1 de 2)*

```
//Archivo de encabezado varcad.h. Es la INTERFAZ de la clase VarCadena cuyos valores
//son cadenas. Un objeto se declara como sigue.
//Observe que se usa (tam_max), no [tam_max]
//        VarCadena el_objeto(tam_max);
//donde tam_max es la longitud máxima que puede tener una cadena.
//tam_max puede ser una variable.
#ifndef VARCAD_H
#define VARCAD_H
#include <iostream>
using namespace std;

namespace varcadsavitch
{

    class VarCadena
    {
    public:
        VarCadena(int tamanio);
        //Inicializa el objeto para aceptar valores de cadena de hasta tamanio
        //caracteres. Asigna al objeto la cadena vacía como valor.

        VarCadena();
        //Inicializa el objeto para aceptar valores de cadena de hasta 100
        //caracteres. Asigna al objeto la cadena vacía como valor.

        VarCadena(const char a[]);
        //Precondición: El arreglo a contiene caracteres que terminan con '\0'.
        //Inicializa el objeto de modo que su valor es la cadena contenida
        //en a y luego se le puedan asignar valores de cadena de hasta
        //strlen(a) caracteres.
        VarCadena(const VarCadena& objeto_cadena);
        //Constructor de copia.

        ~VarCadena();
        //Devuelve al montículo toda la memoria dinámica usada por el objeto.

        int longitud() const;
        //Devuelve la longitud del valor de cadena actual.
```

CUADRO 12.7 *Archivo de interfaz para la clase* VarCadena *(parte 2 de 2)*

```
        void capturar_linea(istream& entra);
        //Precondición: Si entra es un flujo de archivo de entrada ya
        //se conectó a un archivo.
        //Acción: Se copia en el objeto invocador el texto del flujo
        //de entrada entra,
        //hasta '\n'. Si no hay suficiente espacio
        //sólo se copia lo que quepa.

        friend ostream& operator <<(ostream& sale, const VarCadena& la_cadena);
        //Sobrecarga el operador << para poder exhibir con él
        //valores de tipo VarCadena.
        //Precondición: Si sale es un flujo de archivo de salida, ya
        //se conectó a un archivo.

    private:
        char *valor; //apuntador al arreglo dinámico que contiene
        //el valor de cadena.
        int longitud_max; //longitud máxima declarada de cualquier
        //valor de cadena.
    };
}//varcadsavitch

#endif  //VARCAD_H
```

superar este problema C++ cuenta con un tipo especial de función miembro llamada *destructor*.

destructor

Un **destructor** es una función miembro que se invoca automáticamente cuando un objeto de la clase deja de estar dentro del alcance. Esto significa que si nuestro programa contiene una variable local que es un objeto provisto de un destructor, cuando termine la llamada a la función el destructor se invocará automáticamente. Si el destructor se define correctamente, invocará a `delete` para eliminar todas las variables dinámicas creadas por el objeto. Esto podría efectuarse con una sola llamada a `delete` o podría requerir varias llamadas a `delete`. También podríamos querer que nuestro destructor realice algunas otras tareas de aseo, pero devolver memoria al almacén libre es la principal labor del destructor.

nombre del destructor

La función miembro `~VarCadena` es el destructor de la clase `VarCadena` que se muestra en el cuadro 12.7. Al igual que un constructor, un destructor siempre tiene el mismo nombre que la clase de la cual es miembro, pero el destructor lleva el símbolo de tilde ~ al principio de su nombre (para distinguirlo de los constructores). Al igual que un constructor, un destructor no tiene un tipo para el valor devuelto, ni siquiera el tipo `void`. Un destructor no tiene parámetros; por tanto, una clase sólo puede tener un destructor; no se puede sobrecargar el destructor de una clase. Por lo demás, los destructores se definen igual que cualquier otra función miembro.

CUADRO 12.8 *Programa que usa la clase* VarCadena

```
//Programa para demostrar el uso de la clase VarCadena.
#include <iostream>
#include "varcad.h"

void conversacion(int tam_max_nombre);
//Conversa con el usuario.

int main()
{
    using namespace std;
    conversacion(30);
    cout << "Fin de la demostracion.\n";
    return 0;
}
// Ésta es sólo una función de demostración:
void conversacion(int tam_max_nombre)
{
    using namespace std;
    using namespace varcadsavitch;

    VarCadena tu_nombre(tam_max_nombre), nuestro_nombre("Borg");

    cout << "Como te llamas?\n";
    tu_nombre.capturar_linea(cin);
    cout << "Somos " << nuestro_nombre << endl;
    cout << "Nos volveremos a ver, " << tu_nombre << endl;
}
```

Se devuelve memoria al montículo cuando termina la llamada a la función.

Determina el tamaño del arreglo dinámico.

Diálogo de ejemplo

```
Como te llamas?
Kathryn Janeway
Somos Borg
Nos volveremos a ver, Kathryn Janeway
Fin de la demostracion
```

CUADRO 12.9 *Implementación de* VarCadena *(parte 1 de 2)*

```
//Éste es el archivo de implementación: varcad.cxx
//(Su sistema podría requerir otro sufijo en lugar de .cxx.)
//Ésta es la implementación de la clase VarCadena.
//La interfaz de la clase VarCadena está en el archivo de encabezado varcad.h.
#include <iostream>
#include <cstdlib>
#include <cstddef>
#include <cstring>
#include "varcad.h"//Podría necesitarse <cstring> en vez de éste
using namespace std;

namespace varcadsavitch
{
    //Usa stddef y stdlib:
    VarCadena::VarCadena(int tamanio): longitud_max(tamanio)
    {
         valor = new char[longitud_max + 1]; //+1 es para '\0'.
         valor[0] = '\0';
    }

    //Usa stddef y stdlib:
    VarCadena::VarCadena(): longitud_max(100)
    {
         valor = new char[longitud_max + 1]; //+1 es para '\0'.
         valor[0] = '\0';
    }

    //Usa string o cstring, stddef y stdlib:
    VarCadena::VarCadena(const char a[]) : longitud_max(strlen(a))
    {
         valor = new char[longitud_max + 1]; //+1 es para '\0'.
         strcpy(valor, a);
    }
```

CUADRO 12.9 *Implementación de* VarCadena *(parte 2 de 2)*

Constructor copia (analizada más adelante en este capítulo).

```
//Usa cstring, cstddef y cstdlib:
VarCadena::VarCadena(const VarCadena& objeto_cadena)
                        :longitud_max(objeto_cadena.longitud())
{
valor = new char[longitud_max + 1]; //+1 es para '\0'.
strcpy(valor, objeto_cadena.valor);
}

  VarCadena::~VarCadena()
{
     delete [] valor;
}

//Usa cstring:
int VarCadena::longitud() const
{
     return strlen(valor);
}

//Usa iostream:
void VarCadena::capturar_linea(istream& entra)
{
     entra.getline(valor, longitud_max + 1);
}

//Usa iostream:
ostream& operator <<(ostream& sale, const VarCadena& la_cadena)
{
     sale << la_cadena.valor;
     return sale;
}

}//varcadsavitch
```

Destructor

Destructor

Un **destructor** es una función miembro de una clase que se invoca automáticamente cuando un objeto de la clase deja de estar dentro del alcance. Entre otras cosas, esto implica que si un objeto del tipo de clase es una variable local de una función, el destructor se invocará automáticamente como última acción antes de terminar la llamada de función. Los destructores sirven para eliminar cualesquier variables dinámicas que el objeto haya creado, y así se devuelva al almacén libre la memoria que ocupaban. Los destructores podrían realizar también otras labores de aseo. El nombre de un destructor debe consistir en el símbolo de tilde ~ seguido del nombre de la clase.

~VarCadena

Examine la definición del destructor `~VarCadena` que se da en el cuadro 12.9. `~VarCadena` invoca a *`delete`* para eliminar el arreglo dinámico al que apunta la variable miembro `valor`. Estudie otra vez la función `conversacion` del programa de ejemplo que se muestra en el cuadro 12.8. Cada una de las variables locales `tu_nombre` y `nuestro_nombre` crean un arreglo dinámico. Si esta clase no tuviera un destructor, una vez terminada la llamada a `conversacion` estos arreglos dinámicos seguirían ocupando memoria, aunque de nada le sirven al programa. Esto no sería un problema en este caso porque el programa de ejemplo termina poco después de terminar la llamada a `conversacion`, pero si escribiéramos un programa que invocara una y otra vez funciones como `conversacion`, y si la clase `VarCadena` no tuviera un destructor apropiado, las llamadas de función podrían consumir toda la memoria del almacén libre y el programa terminaría de forma anormal.

RIESGO *Apuntadores como parámetros de llamada por valor*

Cuando un parámetro de llamada por valor es de un tipo de apuntador, su comportamiento puede ser sutil y problemático. Considere la llamada de función que se muestra en el cuadro 12.10. El parámetro `temp` de la función `artera` es de llamada por valor, y por tanto es una variable local. Cuando se invoca la función, el valor de `temp` se establece al valor del argumento `p` y se ejecuta el cuerpo de la función. Puesto que `temp` es una variable local, ningún cambio a `temp` deberá trascender fuera de la función `artera`. En particular, no se deberá modificar el valor de la variable de apuntador `p`. No obstante, el diálogo de muestra parece indicar que el valor de la variable de apuntador `p` ha cambiado. Antes de la llamada a la función `artera`, el valor de `*p` era `77`, y después de la llamada el valor de `*p` es `99`. ¿Qué sucedió?

La situación se presenta en forma diagramática en el cuadro 12.11. Aunque el diálogo de ejemplo podría hacernos pensar que `p` se modificó, la llamada a la función `artera` no modificó el valor de `p`. Hay dos cosas asociadas al apuntador `p`: su valor de apuntador y el valor almacenado en el lugar al que `p` apunta. Después de la llamada a `artera`, la variable `p` contiene el mismo valor de apuntador (es decir, la misma dirección de memoria). Lo que la llamada a `artera` modificó fue el valor de la variable a la que `p` apunta, pero no el valor de `p`.

Si el tipo del parámetro es un tipo de clase o estructura que tiene variables miembro de un tipo de apuntador, podrían ocurrir cambios sorpresa similares con argumentos de llamada por valor del tipo de clase. Sin embargo, en este caso podemos evitar (y controlar) tales cambios sorpresa definiendo un *constructor de copia*, como se describe en la siguiente subsección.

CUADRO 12.10 *Un parámetro apuntador de llamada por valor*

```
//Programa para demostrar la forma en que se comportan los
//parámetros de llamada por valor con argumentos apuntador.
#include <iostream>
using namespace std;

typedef int* ApuntadorInt;

void artera(ApuntadorInt temp);

int main()
{
    ApuntadorInt p;

    p = new int;
    *p = 77;
    cout << "Antes de invocar la funcion, *p == "
         << *p << endl;

    artera(p);

    cout << "Despues de invocar la funcion, *p == "
         << *p << endl;

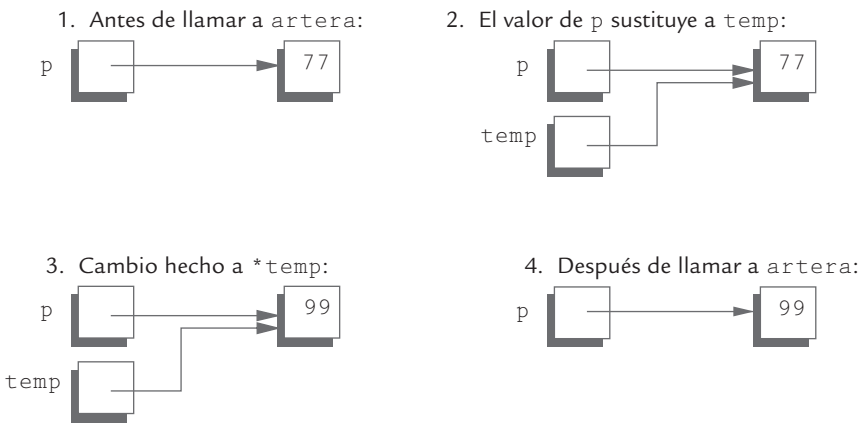
    return 0;
}

void artera(ApuntadorInt temp)
{
    *temp = 99;
    cout << "Dentro de la llamada de funcion, *temp == "
         << *temp << endl;
}
```

Diálogo de ejemplo

```
Antes de invocar la funcion, *p == 77
Dentro de la llamada de funcion, *temp == 99
Despues de invocar la funcion, *p == 99
```

CUADRO 12.11 *La llamada a la función* `artera(p);`



Constructores de copia

constructor de copia

Un **constructor de copia** es un constructor que tiene un parámetro del mismo tipo que la clase. Ese parámetro único debe ser de llamada por referencia, y normalmente va precedido del modificador *const*, así que es un parámetro constante. En todos los demás sentidos un constructor de copia se define y utiliza de la misma manera que cualquier otro constructor.

se invoca cuando se declara un objeto

Por ejemplo, un programa que usa la clase `VarCadena` que definimos en el cuadro 12.7 podría contener lo siguiente:

```
VarCadena linea(20), lema("Los constructores ayudan.");
cout << "Escriba una cadena de longitud 20 o menor:\n";
linea.capturar_linea(cin);
VarCadena temp(linea);//Inicializada por el constructor de copia.
```

El constructor que se usa para inicializar cada uno de los tres objetos de tipo `VarCadena` depende del tipo del argumento que se da entre paréntesis después del nombre del objeto. El objeto `linea` se inicializa con el constructor que tiene un parámetro de tipo *int*; el objeto `lema` se inicializa con el constructor que tiene un parámetro de tipo *const char* `a[]`. De forma similar, el objeto `temp` se inicializa con el constructor que tiene un argumento de tipo *const* `VarCadena&`. Cuando un constructor de copia se usa de esta manera, funciona como cualquier otro constructor.

Un constructor de copia se debe definir de modo que el objeto que se inicializa se convierta en una copia completa e independiente de su argumento. Así pues, en la declaración

```
VarCadena temp(linea);
```

la variable miembro `temp.valor` no se establece simplemente al valor de `linea.valor`; eso produciría dos apuntadores que apuntan al mismo arreglo dinámico. La definición del constructor de copia se muestra en el cuadro 12.9. Observe que en la definición del constructor de copia se crea un nuevo arreglo dinámico y el contenido de un arreglo dinámico se copia en el otro. Así pues, en la declaración anterior, `temp` se inicializa de modo que su valor de cadena sea igual al de `linea`, pero `temp` tiene un arreglo dinámico aparte. De este modo, cualquier cambio que se efectúe a `temp` no afectará a `linea`.

Como hemos visto, los constructores de copia se pueden usar igual que cualquier otro constructor. Además, se invocan automáticamente en ciertas situaciones distintas. En términos generales, siempre que C++ necesita crear una copia de un objeto, invoca automáticamente el constructor de copia. En particular, el constructor de copia se invoca automáticamente en tres circunstancias: 1) cuando se está definiendo un objeto de clase y se inicializa con otro objeto del mismo tipo, 2) cuando una función devuelve un valor del tipo de clase y 3) siempre que un argumento del tipo de clase se "inserta" en lugar de un parámetro de llamada por valor. En este caso, el constructor de copia define lo que significa "insertar".

parámetros de llamada por valor

por qué es necesario un constructor de copia

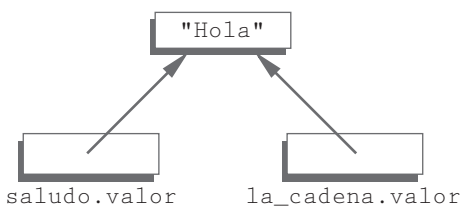
Para entender por qué necesitamos un constructor de copia, veamos qué sucedería si no definiéramos uno para la clase `VarCadena`. *Supongamos que no incluimos el constructor de copia en la definición de la clase* `VarCadena` y que usamos un parámetro de llamada por valor en una definición de función, por ejemplo:

```
void mostrar_cadena(VarCadena la_cadena)
{
    cout << "La cadena es: "
         << la_cadena << endl;
}
```

Consideremos el siguiente código, que incluye una llamada de función:

```
VarCadena saludo("Hola");
mostrar_cadena(saludo);
cout << "Después de la llamada: " << saludo << endl;
```

Suponiendo que no hay constructor de copia, las cosas suceden así: cuando se ejecuta la llamada de función, el valor de `saludo` se copia en la variable local `la_cadena`, de modo que `la_cadena.valor` se hace igual a `saludo.valor`. Sin embargo, éstas son variables de apuntador, así que durante la llamada de función `la_cadena.valor` y `saludo.valor` apuntan al mismo arreglo dinámico, así:



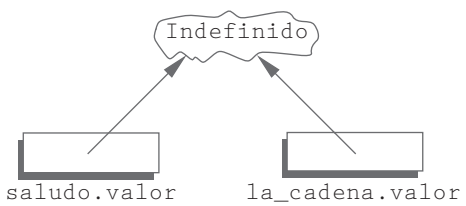
Cuando termina la llamada de función, se invoca el destructor de VarCadena para devolver al almacén libre la memoria que ocupó `la_cadena`. La definición del destructor contiene la siguiente instrucción:

```
delete [] valor;
```

Puesto que el destructor se invoca con el objeto `la_cadena`, esta instrucción equivale a:

```
delete [] la_cadena.valor;
```

lo que cambia la situación a:



Puesto que `saludo.valor` y `la_cadena.valor` apuntan al mismo arreglo dinámico, la eliminación de `la_cadena.valor` equivale a eliminar `saludo.valor`. Así, cuando el programa llega a la instrucción

```
cout << "Después de la llamada: " << saludo << endl;
```

La instrucción `cout` no está definida. Podría ser que la instrucción `cout` produzca la salida que queremos, pero tarde o temprano el hecho de que `saludo.valor` no está definido causará problemas. Un problema importante se presenta si el objeto `saludo` es una variable local en alguna función. En este caso se invocará el destructor con `saludo` cuando termina la llamada a la función. Esa llamada al destructor será equivalente a

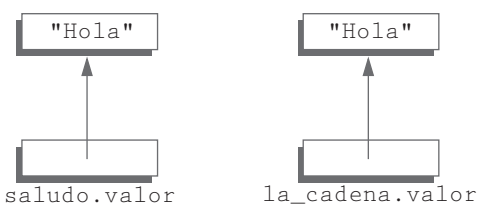
```
delete [] saludo.valor;
```

Sin embargo, como acabamos de ver, el arreglo dinámico al que `saludo.valor` apunta ya se eliminó una vez, y ahora el sistema está tratando de eliminarlo una segunda vez. Invocar *`delete`* dos veces para eliminar el mismo arreglo dinámico (u otra variable creada con *`new`*) puede producir un error de sistema grave que haga que el sistema se caiga.

Eso es lo que sucedería si no hubiera constructor de copia. Por fortuna, incluimos uno en nuestra definición de la clase `VarCadena`, así que se invocará automáticamente el constructor de copia cuando se ejecute la siguiente llamada de función:

```
VarCadena saludo("Hola");
mostrar_cadena(saludo);
```

El constructor de copia define lo que significa "insertar" el argumento `saludo` en lugar del parámetro de llamada por valor `la_cadena`, así que la situación ahora es la siguiente:



De este modo, ningún cambio que se haga a `la_cadena.valor` afectará al argumento `saludo`, y no habrá problemas con el destructor. Si el destructor se invoca para `la_cadena` y luego para `saludo`, cada llamada al destructor eliminará un arreglo dinámico distinto.

valor devuelto

Cuando una función devuelve un valor de un tipo de clase, se invoca automáticamente el constructor de copia para copiar el valor especificado por la instrucción return. Si no hay constructor de copia, se presentarán problemas similares a los que describimos para los parámetros de llamada por valor.

cuando se necesita un constructor de copia

Si en una definición de clase intervienen apuntadores y memoria asignada dinámicamente con el operador *new*, es necesario incluir un constructor de copia. Las clases en las que no intervienen apuntadores ni memoria asignada dinámicamente no necesitan un constructor de copia.

instrucciones de asignación

Al contrario de lo que podríamos esperar, el constructor de copia *no* se invoca cuando hacemos un objeto igual a otro usando el operador de asignación.[3] Sin embargo, si no nos gusta lo que hace el operador de asignación predeterminado, podemos redefinirlo tal como explicamos en la subsección "Sobrecarga del operador de asignación".

Constructor de copia

Un **constructor de copia** es un constructor que tiene un parámetro de llamada por referencia que es del mismo tipo que la clase. Ese único parámetro debe ser de llamada por referencia, y normalmente también es un parámetro constante, es decir, va precedido por el modificador *const*. El constructor de copia de una clase se invoca automáticamente siempre que una función devuelve un valor del tipo de clase. El constructor de copia también se llama automáticamente siempre que un argumento se "inserta" en lugar de un parámetro de llamada por valor del tipo de clase. Además, los constructores de copia se pueden usar de las mismas maneras que los otros constructores.

Cualquier clase que usa apuntadores y el operador *new* debe tener un constructor de copia.

Los tres grandes

El **constructor de copia**, el **operador =** y el **destructor** se conocen como **los tres grandes** porque según los expertos, si necesitamos uno de ellos, necesitamos a los tres. Si cualquiera de ellos falta, el compilador lo creará, pero podría no comportarse como queremos. Por lo tanto, es conveniente definirlos. El constructor de copia y el operador = sobrecargado que el compilador genera si nosotros no los incluimos, funcionan bien si todas las variables miembro son de tipos predefinidos como *int* y *double*, pero podrían tener un comportamiento no deseado con clases que tienen variables miembro que son clases. Para cualquier clase que usa apuntadores y el operador *new*, lo más seguro es definir nuestro propio constructor de copia, = sobrecargado, y destructor.

[3]En C++ existe una distinción clara entre inicialización (los tres casos en los que se invoca el constructor de copia) y asignación. La inicialización usa el constructor de copia para crear un objeto nuevo; el operador de asignación toma un objeto existente y lo modifica de modo que sea una copia idéntica (en todo excepto su ubicación) del miembro derecho de la asignación.

16. Si una clase se llama `MiClase` y tiene un constructor, ¿qué nombre tiene el constructor? Si `MiClase` tiene un destructor, ¿cómo se llama el destructor?

17. Suponga que modifica la definición del destructor del cuadro 12.9 a lo siguiente. ¿Cómo cambiaría el diálogo de ejemplo del cuadro 12.8?

```
VarCadena::~VarCadena()
{
     cout << endl
          << "¡Adios, mundo cruel! La corta vida de\n"
          << "este arreglo dinamico se acerca a su fin.\n";
     delete [] valor;
}
```

18. La que sigue es la primera línea de la definición del constructor de copia de la clase `VarCadena`. El identificador `VarCadena` ocurre tres veces y significa algo ligeramente distinto en cada ocasión. ¿Qué significa en cada uno de los tres casos?

```
VarCadena::VarCadena(const VarCadena& objeto_cadena)
```

19. Conteste estas preguntas acerca de los destructores:

    a) ¿Qué es un destructor y qué nombre debe llevar?

    b) ¿Cuándo se invoca un destructor?

    c) ¿Qué hace realmente un destructor?

    d) ¿Qué debería hacer un destructor?

Sobrecarga del operador de asignación

Suponga que `cadena1` y `cadena2` se declaran como sigue:

```
VarCadena cadena1(10), cadena2(20);
```

La clase `VarCadena` se definió en los cuadros 12.7 y 12.9. Si `cadena2` ha recibido un valor de alguna manera, la instrucción de asignación estará definida, pero su significado podría no ser el que queremos que sea:

```
cadena1 = cadena2;
```

Como siempre, esta versión predefinida del operador de asignación copia el valor de cada una de las variables miembro de `cadena2` en las variables miembro correspondientes de `cadena1`, de modo que el valor de `cadena1.longitud_max` pase a ser el mismo que el de `cadena2.longitud_max` y el valor de `cadena1.valor` pase a ser el mismo que el de `cadena2.valor`. Sin embargo, esto puede causar problemas con `cadena1` y probablemente también con `cadena2`.

La variable miembro `cadena1.valor` contiene un apuntador, y la instrucción de asignación hace que ese apuntador sea el mismo que está contenido en `cadena2.valor`. Por lo tanto, `cadena1.valor` y `cadena2.valor` apuntan al mismo lugar de la memoria. Entonces, si modificamos el valor de cadena que está en `cadena1`, también modificaremos el que está en `cadena2`. Si modificamos el valor de cadena que está en `cadena2`, eso modificará el valor de cadena el que está en `cadena1`.

En síntesis, el operador de asignación predefinido no hace lo que nos gustaría que una instrucción de asignación hiciera con objetos de tipo `VarCadena`. El uso de la versión predefinida del operador de asignación con la clase `VarCadena` sólo puede traer problemas. La forma de resolver esto es sobrecargar el operador de asignación = de modo que haga lo que queremos que haga con objetos de la clase `VarCadena`.

= debe ser miembro

El operador de asignación no se puede sobrecargar de la misma manera que hemos sobrecargado otros operadores, como << y +. Cuando sobrecargamos el operador de asignación éste debe ser miembro de la clase; no puede ser un amigo de la clase. Para añadir una versión sobrecargada del operador de asignación a la clase `VarCadena`, la definición de `VarCadena` debe modificarse así:

```
class VarCadena
{
public:
    void operator =(const VarCadena& miembro_derecho);
    //Sobrecarga el operador de asignación = para copiar
    //una cadena de un objeto a otro.
    <El resto de la definición de la clase puede ser igual
     que en el cuadro 12.7.>
```

Ahora, el operador de asignación se usa como siempre. Por ejemplo, consideremos lo siguiente:

```
cadena1 = cadena2;
```

objeto invocador para =

En la llamada anterior, `cadena1` es el objeto invocador y `cadena2` es el argumento del operador miembro =.

La definición del operador de asignación puede ser ésta:

```
//La que sigue es aceptable, pero daremos una mejor
//definición:
void VarCadena::operator =(const VarCadena& miembro_derecho)
{
    int nueva_longitud = strlen(miembro_derecho.valor);
    if ((nueva_longitud) > longitud_max)
        nueva_longitud = longitud_max;
    for (int i = 0; i < nueva_longitud; i++)
        valor[i] = miembro_derecho.valor[i];
    valor[nueva_longitud] = '\0';
}
```

Observe que se verifica la longitud de la cadena que está en el objeto que está a la derecha del operador de asignación. Si esta cadena es demasiado larga para caber en el objeto que está a la izquierda del operador de asignación (el objeto invocador), sólo se copiarán en el objeto que recibe la cadena tantos caracteres como quepan. Sin embargo, supongamos que no queremos perder caracteres en el proceso de copiado. Para que quepan todos los caracteres, podemos crear un arreglo dinámico nuevo más grande para el objeto que está a la izquierda del operador de asignación. Podríamos tratar de redefinir el operador de asignación como sigue:

```
//Esta versión tiene un error.
void VarCadena::operator =(const VarCadena& miembro_derecho)
{
```

```
        delete [] valor;
        int nueva_longitud = strlen(miembro_derecho.valor);
        longitud_max = nueva_longitud;
        valor = new char[longitud_max + 1];
       for (int i = 0; i < nueva_longitud; i++)
            valor[i] = miembro_derecho.valor[i];
       valor[nueva_longitud] = '\0';
}
```

Esta versión tiene un problema cuando se usa en una asignación en la que el mismo objeto está a ambos lados del operador de asignación, como la siguiente:

```
mi_cadena = mi_cadena;
```

Cuando se ejecuta esta asignación, la primera instrucción que se ejecuta es

```
delete [] valor;
```

Sin embargo, el objeto invocador es `mi_cadena`, así que esto significa

```
delete [] mi_cadena.valor;
```

Por lo tanto, se elimina el valor de cadena que está en `mi_cadena` y el apuntador `mi_cadena.valor` ya no está definido. El operador de asignación ha corrompido el objeto `mi_cadena` y es probable que esta ejecución del programa se haya arruinado.

Una forma de corregir este error es verificar primero si hay suficiente espacio en el miembro arreglo dinámico del objeto que está a la izquierda del operador de asignación y sólo eliminar el arreglo si se necesita espacio adicional. Nuestra definición final del operador de asignación sobrecargado realiza tal verificación:

```
//Ésta es la versión final.
void VarCadena::operator =(const VarCadena& miembro_derecho)
{
     int nueva_longitud = strlen(miembro_derecho.valor);
     if (nueva_longitud > longitud_max)
     {
         delete [] valor;
         longitud_max = nueva_longitud;
         valor = new char[longitud_max + 1];
     }
     for (int i = 0; i < nueva_longitud; i++)
         valor[i] = miembro_derecho.valor[i];
     valor[nueva_longitud] = '\0';
}
```

Para muchas clases, la definición obvia de la sobrecarga del operador de asignación no funciona correctamente cuando el mismo objeto está a ambos lados del operador. Siempre debe verificar este caso y tener cuidado de escribir la definición del operador de asignación sobrecargado de modo que también funcione en este caso.

Ejercicios de AUTOEVALUACIÓN

20. a) Explique con detalle por qué no se necesita un operador de asignación sobrecargado cuando todos los datos son de tipos predefinidos.

    b) Igual que la parte a, pero para un constructor de copia.

    c) Igual que la parte a, pero para un destructor.

Resumen del capítulo

- Un **apuntador** es una dirección de memoria, y permite nombrar de forma indirecta una variable nombrando la dirección que tiene en la memoria de la computadora.
- Las **variables dinámicas** son variables que se crean (y destruyen) mientras un programa se está ejecutando.
- El espacio para las variables dinámicas está en una zona especial de la memoria de la computadora llamada **almacén libre**. Cuando un programa termina de usar una variable dinámica, podemos devolver al almacén libre la memoria que ocupaba, a fin de reutilizarla; esto se hace con la instrucción `delete`.
- Un **arreglo dinámico** es un arreglo cuyo tamaño se determina durante la ejecución del programa. Los arreglos dinámicos se implementan como variables dinámicas de un tipo arreglo.
- Un **destructor** es un tipo especial de función miembro de una clase. El destructor se invoca automáticamente cuando un objeto de la clase deja de estar dentro del alcance. El principal motivo para tener un destructor es devolver memoria al almacén libre a fin de poder reutilizarla.
- Un **constructor de copia** es el que tiene un solo argumento del mismo tipo que la clase. Si definimos un constructor de copia, se invocará automáticamente cuando una función devuelva un valor del tipo de clase y cuando un argumento se "inserte" en lugar de un parámetro de llamada por valor del tipo de clase. Toda clase que use apuntadores y el operador `new` deberá tener un constructor de copia.
- El operador de asignación = puede ser sobrecargado por una clase, de manera que ésta se comporte como quisiéramos que lo hiciera una clase. Sin embargo, éste debe ser sobrecargado como miembro de la clase. No se puede sobrecargar como amiga. Cualquier clase que utilice apuntadores y el operador `new`, debe sobrecargar al operador de asignación para utilizarlo con esa clase.

Respuestas a los ejercicios de autoevaluación

1. Un apuntador es la dirección en memoria de una variable.

2. Si no se pone atención, o no se tiene mucha experiencia, esto parecería la declaración de dos objetos de tipo apuntador a `int`, es decir, `int*`. El problema es que el `*` se relaciona con el *identificador*, no con el tipo (es decir, no con `int`), así que esta declaración declara `apunt_int1` como un apuntador a `int`, y a `apunt_int2` como una variable de tipo `int`.

3.
```
int *p;   //Esto declara una variable apuntador que puede
          // contener un apuntador que apunta a una variable de tipo int.
*p = 17;  //Aquí, * es el operador de desreferencia. Esto asigna 17
//a la posición de memoria a la que p apunta.
```

4.
```
10   20
20   20
30   20
```

Si sustituimos `*p1 = 30;` por `*p2 = 30;`, la salida sería la misma.

5.
```
10   20
20   20
30   30
```

6.
```
delete p;
```

7.
```
typedef int* ApuntNumero;

ApuntNumero mi_apunt;
```

8. El operador *new* recibe un tipo como argumento, y asigna suficiente espacio en el almacén libre para una variable del tipo indicado por el argumento. Si hay suficiente espacio en el almacén libre, *new* devuelve un apuntador a esa memoria (es decir, un apuntador a la nueva variable dinámica); si no hay suficiente espacio en el almacén libre nuestro programa terminará.

9.
```
typedef char* ArregloChar;
```

10.
```
cout << "Escriba 10 enteros:\n";
for (int i = 0; i < 10; i++)
     cin >> dato[i];
```

11.
```
delete [] dato;
```

12.
```
0 1 2 3 4 5 6 7 8 9
```

13.
```
10 1 2 3 4 5 6 7 8 9
```

14.
```
0 1 2 3 4 5 6 7 8 9
```

15.
```
1 2 3 4 5 6 7 8 9
```

16. El *constructor* se llama `MiClase`, que es también el nombre de la clase. El *destructor* se llama `~MiClase`.

17. El diálogo cambiaría a lo siguiente:

```
¿Como te llamas?
Kathryn Janeway
Somos Borg
Nos volveremos a ver, Kathryn Janeway

Adios, mundo cruel! La corta vida de
este arreglo dinamico se acerca a su fin.

Adios, mundo cruel! La corta vida de
este arreglo dinamico se acerca a su fin.
Fin de la demostración
```

18. El `VarCadena` antes del `::` es el nombre de la clase. El `VarCadena` que está inmediatamente después es el nombre de la función miembro. (Recuerde, un constructor es una función miembro que tiene el mismo nombre que la clase.) El `VarCadena` que está dentro de los paréntesis es el tipo del parámetro `objeto_cadena`.

19. a) Un destructor es una función miembro de una clase. El nombre de un destructor siempre comienza con una tilde, ~, seguida del nombre de la clase.

    b) El destructor se invoca cuando un objeto de la clase sale del alcance.

    c) Un destructor hace lo que el programador desea.

    d) Se supone que un destructor debe eliminar las variables dinámicas a las que constructores de la clase han asignado memoria. También podrían efectuar otras tareas de aseo.

20. En el caso del operador de asignación = y el constructor de copia, si los datos sólo son de tipos predefinidos, el mecanismo de copiado es exactamente el que se desea, así que la acción predeterminada es satisfactoria. En el caso del destructor, no se asigna memoria dinámicamente (no se usan apuntadores), así que la acción predeterminada (no hacer nada) también es la deseada.

Proyectos de programación

1. Mejore la definición de la clase `VarCadena` dada en los cuadros 12.6 y 12.8 añadiendo lo siguiente:

   - Funciones miembro `copiar_parte`, que devuelve una subcadena dada, la función miembro `un_car`, que devuelve un solo carácter, y la función `fijar_car`, que cambia un solo carácter.
   - Una versión sobrecargada del operador == (tenga presente que sólo tienen que ser iguales los valores de cadena; los valores de `longitud_max` no tienen que ser iguales).

- Una versión sobrecargada de + que efectúe concatenación de cadenas de tipo `VarCadena`.
- Una versión sobrecargada del operador de extracción >> que lea una palabra (no como `capturar_linea`, que lee toda una línea).

Si vio la sección que trata la sobrecarga del operador de asignación, añádalo también. Escriba un programa de prueba apropiado y pruebe exhaustivamente su definición de clase.

2. 
Realice el proyecto de programación 11 del capítulo 10 empleando un arreglo dinámico. En esta versión de la clase, el constructor deberá tener un solo argumento de tipo *int* que especifica el número máximo de entradas de la lista.

3. Realice el proyecto de programación 10 del capítulo 10 empleando un arreglo dinámico. El programa pedirá al usuario el número de verificaciones en cada categoría y usará esta información para determinar los tamaños de los arreglos dinámicos.

4. 
En el capítulo 11 hablamos sobre los vectores, que son como arreglos que pueden crecer en tamaño. Suponga que no se han definido los vectores en C++. Defina una clase llamada `VectorDouble` que sea como una clase para un vector con el tipo base *double*. Su clase `VectorDouble` deberá tener una variable miembro privada para un arreglo dinámico de objetos tipo *double*. También deberá tener dos variables miembro de tipo *int*; uno llamado `conteo_max` para el tamaño del arreglo dinámico de objetos *double*, y uno llamado `conteo` para el número de posiciones en el arreglo que estén guardando valores. (`conteo_max` es lo mismo que la capacidad de un vector; `conteo` es lo mismo que el tamaño de un vector.)

   Si trata de agregar un elemento (un valor de tipo *double*) al objeto vector de la clase `VectorDouble` y no hay más espacio, entonces se deberá crear un nuevo arreglo dinámico con el doble de la capacidad del arreglo dinámico anterior y los valores del arreglo dinámico anterior se deberán copiar al nuevo arreglo dinámico.

   Su clase deberá tener todo lo siguiente:

- Tres constructores: un constructor predeterminado que cree un arreglo dinámico para 50 elementos, un constructor con un argumento *int* para el número de elementos en el arreglo dinámico inicial, y un constructor de copia.
- Un destructor.
- Una sobrecarga adecuada del operador de asignación =.
- Una sobrecarga adecuada del operador de igualdad ==. Para tener igualdad, los valores de `conteo` y los `conteo` elementos del arreglo deben ser iguales, pero los valores de `conteo_max` no necesitan ser iguales.
- Las funciones miembro `push_back`, `capacity`, `size`, `reserve` y `resize` que se comportan de la misma forma que las funciones miembro con los mismos nombres para los vectores.
- Dos funciones miembro para dar a su clase la misma utilidad que los corchetes: `value_at(i)`, que devuelve el valor del `i`-ésimo elemento en el arreglo dinámico; y `change_value_at(d, i)`, que modifica el valor *double* en el `i`-ésimo elemento del arreglo dinámico y le asigna `d`. Implemente restricciones adecuadas en los argumentos para `value_at` y `change_value_at`. (Su clase no deberá funcionar con los corchetes. Puede modificarse de manera que funcione con corchetes, pero no hemos cubierto el material que le indicará cómo hacer eso.)

5. Defina una clase llamada `Texto` cuyos objetos almacenen listas de palabras. La clase `Texto` será justo igual que la clase `VarCadena`, sólo que utilizará un arreglo dinámico con el tipo base `VarCadena` en vez del tipo base *`char`* y marcará el final del arreglo con un objeto `VarCadena` que consista de un solo espacio en blanco, en vez de utilizar `'\0'` como el marcador final. De manera intuitiva, un objeto de la clase `Texto` representa cierto texto que consiste de palabras separadas por espacios en blanco. Implemente la restricción que asegure que los elementos del arreglo de tipo `VarCadena` no contengan espacios en blanco (excepto los elementos de marcador final de tipo `VarCadena`).

   Su clase `Texto` deberá tener funciones miembro que correspondan a todas las funciones miembro de `VarCadena`. El constructor con un argumento de tipo *`const char`* `a[]` inicializará el objeto `Texto` de la misma manera que se describió antes para `capturar_linea`. Si el argumento de cadena tipo C contiene el símbolo de nueva línea `'\n'`, se considerará como un error y el programa terminará con un mensaje de error.

   La función miembro `capturar_linea` leerá cadenas separadas por espacios en blanco y almacenará cada cadena en un elemento del arreglo dinámico con el tipo base `VarCadena`. Varios espacios en blanco se tratarán como un solo espacio en blanco. Cuando despliegue un objeto de la clase `Texto`, inserte un espacio en blanco entre cada valor de tipo `VarCadena`. Puede suponer que no se utilizan símbolos de tabulación o puede tratar estos símbolos de la misma forma que un espacio en blanco; si es una asignación de su clase, pregunte a su instructor cómo debe tratar el símbolo de tabulación.

   Agregue las mejoras descritas en el Proyecto de programación 1. La versión sobrecargada del operador de extracción >> sólo llenará un elemento del arreglo dinámico.

6. Mediante el uso de arreglos dinámicos, implemente una clase polinomial con suma, resta y multiplicación de polinomios.

   Discusión: Una variable en un polinomio lo único que hace es actuar como recipiente para los coeficientes. Por ende, lo único interesante acerca de los polinomios es el arreglo de coeficientes y el correspondiente exponente. Piense en el polinomio

```
x*x*x + x + 1
```

   Una manera simple de implementar la clase polinomial es utilizar un arreglo de objetos *`double`* para almacenar los coeficientes. El índice del arreglo es el exponente del término correspondiente. ¿En dónde está el término en `x*x` en el ejemplo anterior? Si falta un término, significa que tiene un coeficiente cero.

   Existen técnicas para representar polinomios de alto grado con muchos términos faltantes. Éstos utilizan las técnicas polinomiales denominadas como escasas. A menos que usted ya conozca estas técnicas o que las aprenda con mucha rapidez, no es conveniente que las utilice.

   Proporcione un constructor predeterminado, un constructor de copia y un constructor parametrizado que permita la construcción de un polinomio arbitrario. Suministre también un operador sobrecargado = y un destructor.

   Proporcione estas operaciones:

   - polinomio + polinomio
   - constante + polinomio
   - polinomio + constante
   - polinomio – polinomio
   - constante – polinomio

- polinomio – constante
- polinomio * polinomio
- constante * polinomio
- polinomio * constante

Suministre funciones para asignar y extraer coeficientes, indizados por exponente.

Suministre una función para evaluar el polinomio en un valor de tipo `double`.

Usted deberá decidir si debe implementar estas funciones como miembros, como friend o como funciones independientes.

7. Haga el proyecto de programación 13 del capítulo 10, pero utilice un arreglo dinámico (o arreglos). En esta versión, su programa preguntará al usuario cuántas filas tiene el avión y manejará toda esa cantidad de filas (y no siempre supondrá que el avión tiene 7 filas, como en el proyecto 13 original del capítulo 10).

13 Recursión

13 Recursión

Después de una conferencia sobre cosmología y la estructura del Sistema Solar, una viejecita abordó a William James.
"Su teoría de que el Sol es el centro del Sistema Solar y la Tierra es una esfera que gira alrededor de él suena convincente, señor James, pero es errónea. Yo tengo una teoría mejor", dijo la anciana.
"¿Y en qué consiste, señora?" preguntó James cortésmente.
"En que vivimos sobre una corteza de tierra que descansa en el lomo de una tortuga gigante."
No queriendo dar al traste con tan absurda teoría recurriendo a la gran masa de pruebas científicas que tenía a su disposición, James decidió tratar de disuadir a su oponente señalando algunas de las deficiencias de su postulado.
"Si su teoría es correcta, señora", preguntó, "¿en qué está parada esta tortuga?"
"Es usted un hombre muy astuto, señor James, y su pregunta es muy buena" replicó la viejecita, "pero puedo contestarla. La respuesta es: la primera tortuga está parada en el lomo de una segunda tortuga, mucho más grande, que está parada directamente abajo de la primera."
"Pero, ¿en qué está parada esta segunda tortuga?", persistió James pacientemente.
La anciana lanzó una carcajada triunfante. "No le busque, señor James— hay tortugas hasta abajo."

J. R. ROSS, *Constraints on Variables in Syntax*

Introducción

Ya hemos visto unos cuantos casos de definiciones circulares que tienen un desenlace satisfactorio. Los ejemplos más prominentes son las definiciones de ciertas instrucciones de C++. Por ejemplo, la definición de la instrucción `while` dice que puede contener otras instrucciones (más pequeñas). Puesto que una posibilidad para esas instrucciones más pequeñas es otra instrucción `while`, la definición tiene cierta circularidad. Si escribimos detalladamente la definición de la instrucción `while`, contendrá una referencia a instrucciones `while`. En matemáticas, este tipo de definición circular se denomina definición recursiva. En C++ una función se puede definir en términos de sí misma de la misma manera. Para decirlo con mayor precisión, una definición de función puede contener una llamada a sí misma. En tales casos se dice que la función es recursiva. En este capítulo trataremos la recursión en C++ y, más generalmente, la recursión como técnica de programación y resolución de problemas.

Prerrequisitos

Las secciones 13.1 y 13.2 utilizan material de los capítulos 2 al 4 y del capítulo 7. La sección 13.3 utiliza material de los capítulos 2 al 7, 9 y 10.

13.1 Funciones recursivas para realizar tareas

Recordé también aquella noche que está a la mitad de las Mil y una noches, en la que Sherezada (por un mágico descuido del copista) comienza a relatar palabra por palabra la historia de las Mil y una noches, con el consiguiente riesgo de llegar otra vez a la noche en la que debe repetirla, y así eternamente.

Jorge Luis Borges, *El jardín de senderos que se bifurcan*

Al escribir una función para realizar una tarea, una técnica básica de diseño consiste en dividir la tarea en subtareas. En ocasiones, al menos una de las subtareas es un ejemplo más pequeño de la misma tarea. Por ejemplo, si la tarea consiste en buscar cierto valor en un arreglo, podríamos dividirla en la subtarea de examinar la primera mitad del arreglo y la subtarea de examinar la segunda mitad del arreglo. Las subtareas de examinar las mitades del arreglo son versiones "más pequeñas" de la tarea original. Siempre que una subtarea es una versión más pequeña de la tarea original a realizar, se puede resolver la tarea original usando una función recursiva. Se requiere algo de práctica para descomponer fácilmente los problemas de esta manera, pero una vez que aprende la técnica puede ser una de las formas más fáciles de diseñar un algoritmo, y en última instancia una función C++. Comenzaremos con un sencillo caso de estudio para ilustrar esta técnica.

Recursión

En C++ una definición de función puede contener una llamada a la función que se está definiendo. En tales casos se dice que la función es **recursiva**.

CASO DE ESTUDIO Números verticales

En este caso de estudio diseñaremos una función `void` recursiva que escribe números en la pantalla colocando los dígitos verticalmente, de modo que 1984, por ejemplo, se vea así:

```
1
9
8
4
```

Definición del problema

La declaración de función y el comentario de encabezado para nuestra función son:

```
void escribir_vertical(int n);
//Precondición: n >= 0.
//Postcondición: El número n se escribe en la pantalla
//verticalmente, con cada dígito en una línea aparte.
```

Diseño del algoritmo

Un caso es muy sencillo. Si `n`, el número que se va a escribir, sólo tiene un dígito, simplemente se escribe el número. A pesar de ser tan sencillo, este caso es importante, y no debemos olvidarlo.

Caso sencillo: Si `n < 10`, escribe el número `n` en la pantalla.

Consideremos ahora el caso más representativo en el que el número a escribir consiste en dos o más dígitos. Supongamos que queremos escribir el número `1234` verticalmente, de modo que el resultado sea:

```
1
2
3
4
```

Una forma de descomponer esta tarea en dos subtareas es la siguiente:

1. Despliega todos los dígitos menos el último, así:

```
1
2
3
```

2. Despliega el último dígito, que en este ejemplo es `4`.

La subtarea 1 es una versión más pequeña de la tarea original, así que podemos implementar esta subtarea con una llamada recursiva. La subtarea 2 es el caso sencillo antes mencionado. Así pues, el pseudocódigo que se muestra a continuación es un bosquejo de nuestro algoritmo para la función `escribir_vertical` con parámetro `n`:

```
if (n < 10)
{
    cout << n << endl;
}
else  //n tiene dos o más dígitos:
{
      escribir_vertical(el número n sin su último dígito);
      cout << el ultimo digito de n << endl;
}
```

subtarea recursiva

Para convertir este pseudocódigo en el código de una función C++, lo único que necesitamos es traducir los siguientes dos elementos de pseudocódigo a expresiones C++:

el número `n` sin su último dígito

y

el último dígito de `n`

Estas expresiones se pueden traducir fácilmente a expresiones C++ usando los operadores de división entera / y % como se muestra a continuación:

```
n/10 //el número n sin su último dígito
n%10 //el último dígito de n
```

Por ejemplo, 1234/10 da 123, y 1234%10 da 4.

Varios factores influyeron en nuestra selección de las dos subtareas que usamos en este algoritmo. Uno fue que era fácil calcular el argumento de la llamada recursiva a escribir_vertical (en negritas) que usamos en el pseudocódigo. El número n sin su último dígito se calcula fácilmente con n/10. Como alternativa, podríamos haber intentado dividir las subtareas así:

1. Despliega el primer dígito de n.
2. Despliega el número n sin su primer dígito.

Ésta es una descomposición de la tarea en subtareas perfectamente válida, y se puede implementar de manera recursiva. Sin embargo, es difícil calcular el resultado de eliminar el primer dígito de un número; en cambio, es fácil calcular el resultado de eliminar el último dígito de un número.

Otra razón para elegir esta descomposición es que una de las dos subtareas no requiere una llamada recursiva. Una buena definición de una función recursiva siempre incluye al menos un caso que no implica una función recursiva (además de uno o más casos que sí implican al menos una llamada recursiva). Este aspecto del algoritmo recursivo se analiza en las subsecciones que siguen a este caso de estudio.

Codificación

Ahora podemos armar todas las piezas para producir la función recursiva escribir_vertical que se muestra en el cuadro 13.1. En la siguiente subsección explicaremos con más detalles el funcionamiento de la recursión en este ejemplo.

Rastreo de una llamada recursiva

Veamos qué sucede exactamente cuando se efectúa la siguiente llamada:

```
escribir_vertical(123);
```

Al ejecutarse esta llamada de función, la computadora hace lo que haría con cualquier llamada de función. El argumento 123 sustituye al parámetro n y se ejecuta el cuerpo de la función. Después de la sustitución de 123 por n, el código a ejecutar es:

```
if (123 < 10)
{
    cout << 123 << endl;
}
else  //n tiene dos o más dígitos:
{
     escribir_vertical(123/10);
     cout << (123%10) << endl;
}
```

El cálculo parará aquí hasta que la llamada recursiva regrese.

CUADRO 13.1 *Una función de salida recursiva (parte 1 de 2)*

```
//Programa para demostrar la función recursiva escribir_vertical.
#include <iostream>
using namespace std;

void escribir_vertical(int n);
//Precondición: n >= 0.
//Postcondición: El número n se escribe en la pantalla
//verticalmente, con cada dígito en una línea aparte.

int main()
{
    cout << "escribir_vertical(3):" << endl;
    escribir_vertical(3);

    cout << "escribir_vertical(12):" << endl;
    escribir_vertical(12);

    cout << "escribir_vertical(123):" << endl;
    escribir_vertical(123);

    return 0;
}

//usa iostream:
void escribir_vertical(int n)
{
    if (n < 10)
    {
       cout << n << endl;
    }
    else //n tiene dos o más dígitos:
    {
       escribir_vertical(n/10);
       cout << (n%10) << endl;
    }
}
```

CUADRO 13.1 *Una función de salida recursiva (parte 2 de 2)*

Diálogo de ejemplo

```
escribir_vertical(3):
3
escribir_vertical(12):
1
2
escribir_vertical(123):
1
2
3
```

Puesto que `123` no es menor que `10`, la expresión lógica de la instrucción *`if-else`* es *`false`*, y se ejecuta la parte *`else`*. Sin embargo, la parte *`else`* inicia con la siguiente llamada de función:

```
escribir_vertical(n/10);
```

que (dado que `n` es igual a `123`) es la llamada

```
escribir_vertical(123/10);
```

que equivale a:

```
escribir_vertical(12);
```

Cuando la ejecución llega a esta llamada recursiva, el cálculo de la función en curso se coloca en animación suspendida y se ejecuta esta llamada recursiva. Cuando termine esta llamada recursiva, la ejecución del cálculo suspendido regresará a este punto y el cálculo continuará a partir de aquí.

La llamada recursiva

```
escribir_vertical(12);
```

se maneja igual que cualquier otra llamada de función. El argumento `12` sustituye al parámetro `n` y se ejecuta el cuerpo de la función. Después de que `12` sustituye a `n`, hay dos cálculos pendientes, uno suspendido y uno activo, como se muestra a continuación:

```
if (123 < 10)
{
    cout
}
else //
{
    esc
    cout
```

```
if (12 < 10)
{
    cout << 12 << endl;
}
else //n tiene dos o más dígitos:
{
      escribir_vertical(12/10);
      cout << (12%10) << endl;
}
```

El cálculo parará aquí hasta que la llamada recursiva regrese.

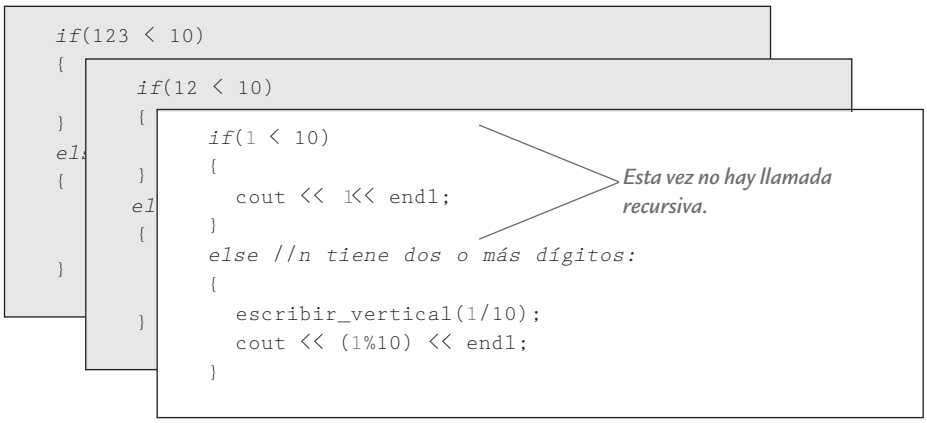
Puesto que 12 no es menor que 10, la expresión booleana de la instrucción *if-else* es *false* y se ejecuta la parte *else*. Sin embargo, como ya vimos, dicha parte inicia con una llamada recursiva. El argumento para esta llamada recursiva es n/10, que en este caso equivale a 12/10. Por tanto, este segundo cálculo de la función escribir_vertical quedará suspendido y se ejecutará la siguiente llamada recursiva:

```
escribir_vertical(12/10);
```

que equivale a:

```
escribir_vertical(1);
```

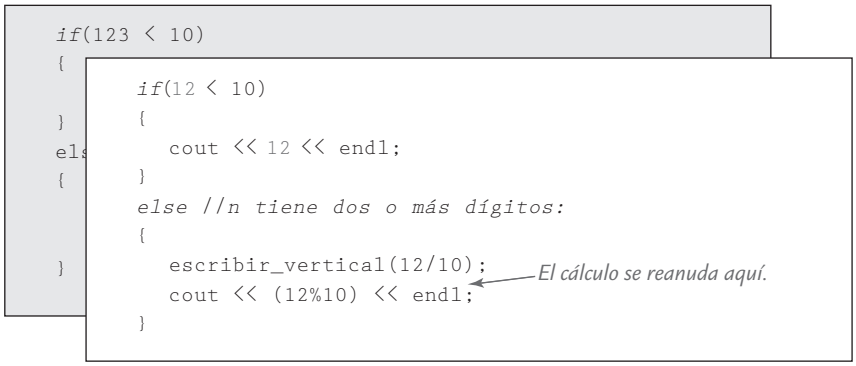
En este punto hay dos cálculos suspendidos que esperan para reanudarse y la computadora comienza a ejecutar la nueva llamada recursiva, que se maneja igual que las anteriores. El argumento 1 sustituye al parámetro n y se ejecuta el cuerpo de la función. En este punto, el cálculo tiene el siguiente aspecto:



Cuando el cuerpo de la función se ejecuta esta vez, sucede algo distinto. Puesto que 1 es menor que 10, la expresión booleana de la instrucción *if-else* es *true*, y se ejecuta la instrucción que está antes del *else*. Ése no es más que una instrucción cout que escribe el argumento 1 en la pantalla, así que la llamada escribir_vertical(1) despliega 1 en la pantalla y termina sin una llamada recursiva.

desplegar el dígito 1

Cuando termina la llamada escribir_vertical(1), el cálculo suspendido que estaba esperando a que terminara continúa donde se quedó, y la situación es la siguiente:



desplegar el dígito 2

Cuando este cálculo suspendido se reanuda, ejecuta una instrucción `cout` que despliega el valor `12%10`, que es `2`. Con eso termina el cálculo, pero todavía hay otro cálculo suspendido que espera para reanudarse. Cuando se reanuda ese último cálculo, la situación es:

```
if (123 < 10)
{
   cout << 123 << endl;
}
   else //n tiene dos o más dígitos:
{
   escribir_vertical(123/10);
   cout << (123%10) << endl;   <-- El cálculo se reanuda aquí.
}
```

desplegar el dígito 3

Al reanudarse este último cálculo suspendido, se despliega el valor `123%10`, que es `3`, y con ello termina la ejecución de la llamada de función original. Y, efectivamente, se han desplegado en la pantalla los dígitos `1`, `2` y `3`, uno por línea, en ese orden.

La recursión bajo la lupa

La definición de la función `escribir_vertical` usa recursión. Sin embargo, no hicimos nada nuevo o diferente al evaluar la llamada de función `escribir_vertical(123)`; la tratamos igual que cualquiera de las llamadas de funciones que vimos en capítulos anteriores. Simplemente sustituimos el argumento `123` por el parámetro `n` y luego ejecutamos el código del cuerpo de la definición de función. Cuando llegamos a la llamada recursiva

```
escribir_vertical(123/10);
```

simplemente repetimos el proceso una vez más.

cómo funciona la recursión

La computadora sigue la pista a las llamadas recursivas de la siguiente manera. Cuando se invoca una función, la computadora inserta los argumentos en lugar de los parámetros y comienza a ejecutar el código. Si encuentra una llamada recursiva, detiene temporalmente el cálculo. Hace esto porque necesita conocer el resultado de la llamada recursiva para poder continuar. La computadora guarda toda la información que necesita para continuar el cálculo más adelante, y procede a evaluar la llamada recursiva. Cuando ésta se completa, la computadora regresa para terminar el cálculo exterior.

cómo termina la recursión

El lenguaje C++ no restringe el uso de llamadas recursivas en las definiciones de funciones. Sin embargo, para que una definición recursiva sea útil, debe diseñarse de modo que cualquier llamada a la función termine finalmente con algún fragmento de código que no dependa de la recursión. La función podría llamarse a sí misma, y esa llamada recursiva podría llamar a la función otra vez. El proceso podría repetirse cualquier cantidad de veces. Sin embargo, el proceso sólo terminará si en algún momento una de las llamadas recursivas puede devolver un valor sin depender de la recursión. El bosquejo general de una definición de función recursiva útil es el siguiente:

- Uno o más casos en los que la función realiza su cometido usando llamadas recursivas para efectuar una o más versiones más pequeñas de la tarea.
- Uno o más casos en los que la función efectúa su tarea sin usar llamadas recursivas. Estos casos sin llamadas recursivas se llaman **casos base** o **casos de paro**.

caso base
caso de paro

Es común que una instrucción `if-else` determine cuál de los casos se ejecutará. Una situación típica es aquella en la que la llamada de función original ejecuta un caso que incluye una llamada recursiva. Esa llamada recursiva podría a su vez ejecutar un caso que requiere otra llamada recursiva. Durante cierto número de veces, cada llamada recursiva produce otra llamada recursiva, pero llega un momento en que se aplica uno de los casos de paro. *Toda llamada a la función debe conducir tarde o temprano a un caso de paro; de lo contrario, la llamada de función nunca terminará porque habrá una cadena infinita de llamadas recursivas*. (En la práctica, una llamada que incluye una cadena infinita de llamadas recursivas termina anormalmente en lugar de continuar de manera indefinida.)

La forma más común de asegurar que se llegue finalmente a un caso de paro es escribir la función de modo que alguna cantidad numérica (positiva) disminuya en cada llamada recursiva, e incluir un caso de paro para algún valor "pequeño". Así es como diseñamos la función `escribir_vertical` en el cuadro 13.1. Cuando se invoca la función `escribir_vertical`, esa llamada produce una llamada recursiva con un argumento más pequeño. Esto continúa y cada llamada recursiva produce otra llamada recursiva, hasta que el argumento es menor que `10`. En ese momento, la llamada de función termina sin producir más llamadas recursivas, y el proceso regresa gradualmente a la llamada original para finalmente terminar.

Forma general de una definición de función recursiva

El bosquejo general de una definición de función recursiva útil es el siguiente:

- Uno o más casos que incluyen una o más llamadas recursivas a la función que se está definiendo. Estas llamadas recursivas deberán resolver versiones "más pequeñas" de la tarea que realiza la función que se está definiendo.
- Uno o más casos que no incluyen llamadas recursivas. Estos casos sin llamadas recursivas se llaman **casos base** o **casos de paro**.

RIESGO *Recursión infinita*

recursión infinita

En el ejemplo de la función `escribir_vertical` que vimos en la subsección anterior, la serie de llamadas recursivas finalmente llegó a una llamada que no implicó recursión (es decir, se llegó a un caso de paro). Si, en vez de ello, cada llamada recursiva produce otra llamada recursiva, una llamada a la función se ejecutará, en teoría, eternamente. Esto se llama **recursión infinita**. En la práctica, una función así se ejecutará hasta que la computadora se quede sin recursos, y el programa terminará anormalmente. Dicho de otro modo, una definición recursiva no debe ser "recursiva hasta abajo". De lo contrario, como la explicación del Universo que dio la viejecita al principio del capítulo, una llamada a la función nunca terminará, excepto quizá en frustración.

No es difícil encontrar ejemplos de recursión infinita. La que sigue es una versión sintácticamente correcta de una función C++, que podría ser resultado de un intento por definir otra versión de la función `escribir_vertical`:

```
void nueva_escribir_vertical(int n)
{
    nueva_escribir_vertical(n/10);
    cout << (n%10) << endl;
}
```

Si incrustamos esta definición en un programa que llama a esta función, el compilador traducirá la definición a código de máquina y podremos ejecutar ese código de máquina. Es más, la definición es hasta cierto punto razonable: dice que para desplegar el argumento de `nueva_escribir_vertical`, primero hay que desplegar todos los dígitos menos el último y luego hay que desplegar el último. Sin embargo, cuando esta función se invoque producirá una sucesión infinita de llamadas recursivas. Si invocamos `nueva_escribir_vertical(12)`, esa ejecución se detendrá para ejecutar la llamada recursiva `nueva_escribir_vertical(12/10)`, que equivale a `nueva_escribir_vertical(1)`. La ejecución de esa llamada recursiva se detendrá a su vez para ejecutar la llamada recursiva

```
nueva_escribir_vertical(1/10);
```

la cual equivale a

```
nueva_escribir_vertical(0);
```

Esa ejecución, a su vez, se detendrá para ejecutar la llamada recursiva `nueva_escribir_vertical(0/10);`, que también equivale a

```
nueva_escribir_vertical(0);
```

esto producirá otra llamada recursiva para ejecutar otra vez la misma llamada recursiva `nueva_escribir_vertical(0);`, y así eternamente. Puesto que la definición de `nueva_escribir_vertical` no tiene caso de paro, el proceso continuará indefinidamente (o hasta que la computadora se quede sin recursos).

Ejercicios de AUTOEVALUACIÓN

1. ¿Qué salidas produce el siguiente programa?

```
#include <iostream>
using namespace std;
void porra(int n);

int main()
{
    porra(3);
    return 0;
}

void porra(int n)
{
```

```
    if (n == 1)
    {
        cout << "Hurra\n";
    }
    else
    {
        cout << "Hip ";
        porra(n - 1);
    }
}
```

2. Escriba una función *void* recursiva que tenga un parámetro que sea un entero positivo y despliegue en la pantalla ese número de asteriscos '*', todos en una línea.

3. Escriba una función *void* recursiva que tenga un parámetro que sea un entero positivo. Al invocarse, la función desplegará al revés y en la pantalla a su argumento. Por ejemplo, si el argumento es 1234, la función desplegará lo siguiente:

   4321

4. Escriba una función *void* recursiva que reciba un solo argumento *int* n y que despliegue los enteros 1, 2, ..., n.

5. Escriba una función *void* recursiva que reciba un solo argumento *int* n y que despliegue los enteros n, n−1, ..., 3, 2, 1. Sugerencia: Puede llegar del código del ejercicio 4 al del ejercicio 5 (o viceversa) intercambiando sólo dos líneas.

Pilas para recursión

pila

Para seguir la pista a la recursión, y para otras cosas, casi todos los sistemas de cómputo emplean una estructura llamada *pila*. Una **pila** es un tipo de estructura de memoria muy especializado análogo a una pila de hojas de papel. En esta analogía hay un abasto inagotable de hojas de papel en blanco. Si queremos guardar alguna información en la pila, la escribimos en una de estas hojas de papel y la colocamos encima de la pila de papeles. Si queremos colocar más información en la pila, tomamos una hoja limpia, escribimos la información en ella, y colocamos la hoja encima de la pila. De esta sencilla forma, podemos colocar más y más información en la pila.

El procedimiento para sacar información de la pila también es muy sencillo. Podemos leer la hoja de papel que está hasta arriba, y si ya no la necesitamos podemos tirarla. Hay una complicación: la única hoja accesible es la de arriba. Si queremos leer, digamos, la tercera hoja contando desde arriba, tendremos que desechar las dos primeras hojas. Puesto que la última hoja que se coloca en la pila es la primera que se saca de la pila, decimos que la pila es una estructura de memoria de **último en entrar/primero en salir**.

último en entrar/primero en salir

recursión

Con una pila, la computadora puede seguir la pista fácilmente a la recursión. Cada vez que se llama a una función, se toma una nueva hoja de papel, en la cual se copia la definición de la función, sustituyendo sus parámetros por los argumentos. Luego la computadora comienza a ejecutar el cuerpo de la definición de la función. Si la computadora encuentra una llamada recursiva, detiene el cálculo que está efectuando en esa hoja para calcular el valor devuelto por la llamada recursiva. Sin embargo, antes de calcular la llamada recursiva, la computadora guarda suficiente información para que, cuando por fin determine el valor devuelto por la llamada recursiva, pueda continuar con el cálculo detenido. Esta información guardada se escribe en una hoja de papel y se coloca en la pila. Se usa una nueva hoja de papel para la llamada recursiva. La computadora escribe una segunda copia de

la definición de función en esta nueva hoja, inserta los argumentos en lugar de los parámetros de la función, y comienza a ejecutar la llamada recursiva. Si la computadora llega a una llamada recursiva dentro de la copia que se invocó recursivamente, repite el proceso de guardar información en la pila y usa una nueva hoja de papel para la nueva llamada recursiva. Este proceso se ilustra en la subsección llamada "Rastreo de una llamada recursiva". Aunque no las llamamos pilas en ese momento, las ilustraciones de cálculos colocados uno sobre otro ilustran las acciones de la pila.

Este proceso continúa hasta que alguna llamada recursiva a la función termina su cálculo sin producir más llamadas recursivas. Cuando eso sucede, la computadora dirige su atención a la hoja de papel que está hasta arriba de la pila. Esta hoja contiene el cálculo parcialmente efectuado que estaba esperando el cálculo recursivo que se acaba de completar. Ahora es posible continuar con ese cálculo suspendido. Cuando termina ese cálculo, la computadora desecha esa hoja de papel y el cálculo suspendido que estaba abajo de ella en la pila se convierte en el cálculo que está hasta arriba de la pila. La computadora dirige su atención al cálculo suspendido que ahora está arriba de la pila, y así sucesivamente. El proceso continúa hasta que se completa el cálculo de la hoja de hasta abajo. Dependiendo del número de llamadas recursivas hechas y de la forma en que se escribió la definición de la función, la pila podría crecer y encogerse de diversas maneras. Tenga presente que sólo podemos acceder a las hojas de la pila según un régimen de último en entrar/primero en salir, pero eso es exactamente lo que se necesita para seguir la pista a las llamadas recursivas. Cada versión suspendida está esperando que se finalice la versión que está inmediatamente arriba de ella en la pila.

Las computadoras no usan hojas de papel. Esto era sólo una analogía. La computadora usa porciones de la memoria en lugar de hojas de papel. El contenido de una de esas porciones de memoria ("hojas de papel") se denomina **marco de activación**. Estos marcos de activación se manejan según el régimen de último en entrar/primero en salir que acabamos de ver. (Los marcos de activación no contienen una copia completa de la definición de la función; sólo hacen referencia a la copia única de dicha definición. No obstante, un marco de activación contiene suficiente información para que la computadora pueda actuar como si el marco de activación contuviera una copia completa de la definición de la función.

marco de activación

Pila

Una **pila** es una estructura de memoria de *último en entrar/primero en salir*. El primer elemento al que se hace referencia o que se saca de la pila es siempre el último que se metió en la pila. Las computadoras usan pilas para seguir la pista a la recursión (y para otras cosas).

RIESGO *Desbordamiento de la pila*

El tamaño de una pila siempre tiene un límite. Si hay una cadena larga en la que una función se llama recursivamente a sí misma, y el resultado es otra llamada recursiva, y esa llamada produce una llamada recursiva más, y así sucesivamente, cada llamada recursiva de la cadena hará que se coloque otro marco de activación en la pila. Si la cadena es demasiado larga, la pila tratará de crecer más allá de su límite. Ésta es una condición de error

llamada **desbordamiento de pila**. Si aparece un mensaje de error que dice *stack overflow* (desbordamiento de pila), lo más probable es que una llamada de función haya producido una cadena excesivamente larga de llamadas recursivas. Una causa común de desbordamiento de pila es la recursión infinita. Si una función está generando una recursión infinita, tarde o temprano tratará de hacer que la pila exceda su límite de tamaño, por grande que éste sea.

Desbordamiento de la pila

Recursión e iteración

La recursión no es absolutamente necesaria. De hecho, algunos lenguajes de programación no la permiten. Cualquier tarea que puede efectuarse con recursión también puede realizarse de alguna otra manera. Por ejemplo, el cuadro 13.2 contiene una versión no recursiva de la función que se dio en el cuadro 13.1. La versión no recursiva de una función por lo regular usa uno o más ciclos de algún tipo en lugar de la recursión. Por esa razón, la versión no recursiva se conoce como **versión iterativa**. Si sustituimos la definición de la función `escribir_vertical` del cuadro 13.1 por la versión del cuadro 13.2, la salida será la misma. Como sucede en este caso, una versión recursiva de una función puede ser mucho más sencilla que una versión iterativa.

versión iterativa

Una función escrita recursivamente por lo regular se ejecuta con mayor lentitud y usa más memoria que una versión iterativa equivalente. Aunque parece que la versión iterativa de `escribir_vertical` del cuadro 13.2 usa más memoria y realiza más cálculos que la versión recursiva del cuadro 13.1, en realidad las dos versiones usan aproximadamente la misma memoria y realizan un número comparable de cálculos. De hecho, la versión recursiva podría usar más memoria y ejecutarse un poco más lentamente, porque la computadora necesita realizar un buen número de manipulaciones de la pila para seguir la pista a la recursión. Sin embargo, dado que el sistema hace todo esto automáticamente, la recursión a veces puede facilitar la tarea del programador, y en algunos casos producir código más fácil de entender. Como veremos en los ejemplos de este capítulo y en los ejercicios de autoevaluación y proyectos de programación, hay ocasiones en las que una definición recursiva es más sencilla y más clara, y otras en las que una definición iterativa es más sencilla y más clara.

eficiencia

Ejercicios de AUTOEVALUACIÓN

6. Si un programa produce un mensaje de error que dice *stack overflow* (desbordamiento de pila), ¿cuál es la causa probable del error?

7. Escriba una versión iterativa de la función `porra` definida en el ejercicio de autoevaluación 1.

8. Escriba una versión iterativa de la función definida en el ejercicio de autoevaluación 2.

9. Escriba una versión iterativa de la función definida en el ejercicio de autoevaluación 3.

10. Rastree la solución recursiva que dio para el ejercicio de autoevaluación 4.

11. Rastree la solución recursiva que dio para el ejercicio de autoevaluación 5.

CUADRO 13.2 *Versión iterativa de la función del cuadro 13.1*

```
//Uses iostream:
void escribir_vertical(int n)
{
    int decenas_en_n = 1;
    int fragmento_izq = n;
    while (fragmento_izq > 9)
    {
        fragmento_izq = fragmento_izq/10;
        decenas_en_n = decenas_en_n*10;
    }
    //decenas_en_n es una potencia de 10 que tiene el mismo
    //número de dígitos que n. Por ejemplo, si n es 2345,
    //decenas_en_n es 1000
    for (int potencia_de_10 = decenas_en_n;
           potencia_de_10 > 0; potencia_de_10 = potencia_de_10/10)
    {
        cout << (n/potencia_de_10) << endl;
        n = n%potencia_de_10;
    }
}
```

13.2 Funciones recursivas para obtener valores

La iteración es humana, la recursión es divina.

Anónimo

Forma general de una función recursiva que devuelve un valor

Todas las funciones recursivas que hemos visto hasta ahora son funciones `void`, pero la recursión no está limitada a este tipo de funciones. Una función recursiva puede devolver un valor de cualquier tipo. La técnica para diseñar funciones recursivas que devuelven un valor es básicamente la misma que aprendimos para las funciones `void`. He aquí un bosquejo para una definición de función recursiva útil que devuelve un valor:

- Uno o más casos en los que el valor devuelto se calcule en términos de llamadas a la misma función (es decir, usando llamadas recursivas). Como en el caso de las funciones `void`, los argumentos de las llamadas recursivas deberán ser intuitivamente "más pequeños".
- Uno o más casos en los que el valor devuelto se calcule sin usar llamadas recursivas. Estos casos sin llamadas recursivas se llaman **casos base** o **casos de paro** (igual que con las funciones `void`).

casos base
casos de paro

Esta técnica se ilustra en el siguiente ejemplo de programación.

EJEMPLO DE PROGRAMACIÓN

Otra función de potencias

En el capítulo 3 presentamos la función predefinida `pow` que calcula potencias. Por ejemplo, `pow(2.0, 3.0)` devuelve $2.0^{3.0}$, así que lo siguiente asigna `8.0` a la variable `x`:

```
double x = pow(2.0, 3.0);
```

La función `pow` recibe dos argumentos de tipo *`double`* y devuelve un valor de tipo *`double`*. El cuadro 13.3 contiene una definición recursiva de una función que es similar pero que opera con el tipo *`int`*, en lugar de *`double`*. Esta nueva función se llama `potencia`. Por ejemplo, lo que sigue asigna a `y` el valor `8`, ya que 2^3 es 8:

```
int y = potencia(2, 3);
```

Nuestra principal razón para definir la función `potencia` es tener un ejemplo sencillo de función recursiva, pero hay situaciones en las que `potencia` es preferible a `pow`. La función `pow` devuelve valores de tipo *`double`*, que son cantidades aproximadas. La función `potencia` devuelve valores de tipo *`int`*, que son cantidades exactas. En algunas situaciones podría necesitarse la exactitud adicional que ofrece la función `potencia`.

La definición de la función `potencia` se basa en la siguiente fórmula:

x^n es igual a x^{n-1} * x

Si traducimos esta fórmula a C++ dirá que el valor devuelto por `potencia(x, n)` debe ser igual al valor de la expresión

```
potencia(x, n − 1)*x
```

La definición de la función `potencia` dada en el cuadro 13.3 sí devuelve este valor para `potencia(x, n)`, siempre que `n > 0`.

El caso en el que `n` es igual a `0` es el caso de paro. Si `n` es `0`, entonces `potencia(x, n)` simplemente devuelve `1` (porque x^0 es 1).

Veamos qué sucede cuando se invoca la función `potencia` con algunos valores de muestra. Consideremos primero la sencilla expresión:

```
potencia(2, 0)
```

Cuando se invoca la función, el valor de `x` se establece a `2`, el valor de `n` se establece a `0`, y se ejecuta el código del cuerpo de la definición de la función. Puesto que el valor de `n` es válido, se ejecuta la instrucción *`if-else`*. Puesto que este valor de `n` no es mayor que `0`, se usa la instrucción *`return`* que sigue al *`else`*, y la llamada de función devuelve `1`. Así, lo que sigue hará el valor de `y` igual a 1.

```
int y = potencia(2, 0);
```

Veamos ahora un ejemplo que implica una llamada recursiva. Consideremos la expresión

```
potencia(2, 1)
```

CUADRO 13.3 *La función recursiva* potencia

```
//Programa que demuestra la función recursiva potencia.
#include <iostream>
#include <cstdlib>
using namespace std;

int potencia(int x, int n);
//Precondición: n >= 0.
//Devuelve x elevado a la potencia n.

int main()
{
    for (int n = 0; n < 4; n++)
         cout << "3 a la potencia " << n
              << " es " << potencia(3, n) << endl;

           return 0;
}
//Usa iostream y cstdlib:
int potencia(int x, int n)
{
    if (n < 0)
    {
         cout << "Argumento de potencia no valido.\n";
         exit(1);
    }
    if (n > 0)
         return ( potencia(x, n - 1)*x );
    else // n == 0
         return (1);
}
```

Diálogo de ejemplo

```
3 a la potencia 0 es 1
3 a la potencia 1 es 3
3 a la potencia 2 es 9
3 a la potencia 3 es 27
```

Cuando se invoca la función, el valor de `x` se establece a `2`, el valor de `n` se establece a `1`, y se ejecuta el código del cuerpo de la definición de la función. Puesto que este valor de `n` es mayor que `0`, se usa la siguiente instrucción *`return`* para determinar el valor devuelto:

```
return ( potencia(x, n - 1)*x );
```

que en este caso equivale a

```
return ( potencia(2, 0)*2 );
```

En este momento se suspende el cálculo de `potencia(2, 1)`, se coloca en la pila una copia de este cálculo suspendido, y la computadora inicia una nueva llamada de función para calcular el valor de `potencia(2, 0)`. Como ya vimos, el valor de `potencia(2, 0)` es `1`. Después de determinar el valor de `potencia(2, 0)`, la computadora sustituye la expresión `potencia(2, 0)` por su valor de `1` y reanuda el cálculo suspendido. Dicho cálculo determina el valor final de `potencia(2, 1)` a partir de la instrucción *`return`* anterior así:

```
potencia(2, 0)*2 es 1*2 que es 2
```

y así el valor final devuelto por `potencia(2, 1)` es `2`. Entonces, lo que sigue establecerá el valor de `z` igual a `2`:

```
int z = potencia(2, 1);
```

Si el segundo argumento es un número grande se producirá una cadena larga de llamadas recursivas. Por ejemplo, considere la instrucción:

```
cout << potencia(2, 3);
```

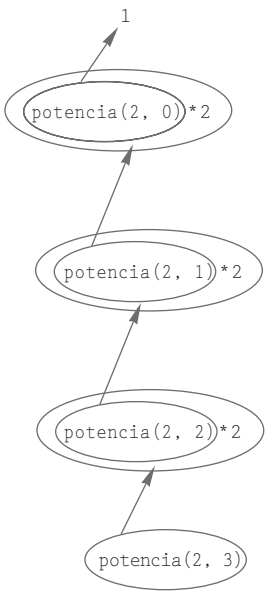
El valor de `potencia(2, 3)` se calcula así:

```
potencia(2, 3) es potencia(2, 2)*2
  potencia(2, 2) es potencia(2, 1)*2
    potencia(2, 1) es potencia(2, 0)*2
      potencia(2, 0) es 1 (caso de paro)
```

Cuando la computadora llega al caso de paro `potencia(2, 0)`, hay tres cálculos suspendidos. Después de calcular el valor devuelto para el caso de paro, la computadora reanuda el cálculo que se suspendió más recientemente para determinar el valor de `potencia(2, 1)`. Después, la computadora completa todos los demás cálculos suspendidos, insertando cada valor calculado en el siguiente cálculo suspendido hasta llegar al cálculo de la llamada original `potencia(2, 3)` y completarlo. Los detalles del cálculo se ilustran en el cuadro 13.4.

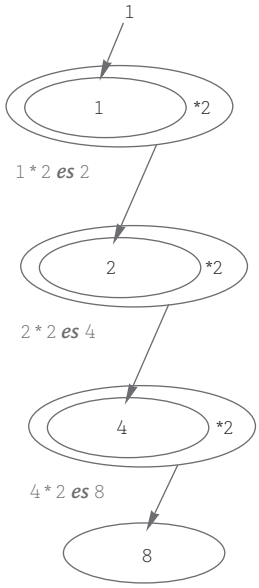
CUADRO 13.4 *Evaluación de una llamada a la función recursiva* `potencia(2, 3)`

Sucesión de llamadas recursivas



Comience aquí

Cómo se calcula el valor final



`potencia(2, 3)` es 8

Ejercicios de AUTOEVALUACIÓN

12. ¿Qué salida produce el siguiente programa?

```
#include <iostream>
using namespace std;
int misterio(int n);
//Precondición n >= 1.

int main()
{
    cout << misterio(3);
    return 0;
}
```

```
int misterio(int n)
{
    if (n <= 1)
       return 1;
    else
       return ( misterio(n - 1) + n );
}
```

13. ¿Qué salida produce el siguiente programa?, ¿qué función matemática muy conocida es `rosa`?

```
#include <iostream>
using namespace std;
int rosa(int n);
//Precondición n >= 0.
int main()
{
    cout << rosa(4);
    return 0;
}
int rosa(int n)
{
    if(n <= 0)
       return 1;
    else
       return ( rosa(n - 1) * n );
}
```

14. Redefina la función `potencia` de modo que también funcione con exponentes negativos. Para ello también será preciso cambiar a `double` el tipo del valor devuelto. La declaración de función y comentario de encabezado para la versión redefinida de `potencia` son:

```
double potencia(int x, int n);
//Precondición: Si n < 0, entonces x no es 0.
//Devuelve x elevado a la potencia n.
```

Sugerencia: x^{-n} es igual a $1/(x^n)$.

13.3 Razonamiento recursivo

En el mundo hay dos clases de personas, las que dividen
el mundo en dos clases de personas, y las que no lo hacen.

Anónimo

Técnicas de diseño recursivo

Cuando define y utiliza funciones recursivas no necesita saber en todo momento acerca de la pila y de los cálculos suspendidos. La utilidad de la recursión proviene del hecho de que podemos olvidarnos de esos detalles y dejar que la computadora se encargue de la conta-

bilidad. Consideremos el ejemplo de la función `potencia` del cuadro 13.3. La forma de pensar en la definición de esta función es:

`potencia(x, n)` devuelve `potencia(x, n − 1)*x`

Puesto que x^n es igual a $x^{n-1}*x$, éste es el valor devuelto correcto, siempre y cuando el cálculo llegue a un caso de paro y calcule correctamente el caso de paro. Por tanto, después de verificar que la parte recursiva de la definición sea correcta, lo único que necesitamos verificar es que la cadena de llamadas recursivas siempre llegue a un caso de paro y que el caso de paro devuelva el valor correcto.

Al diseñar una función recursiva no es preciso rastrear toda la sucesión de llamadas recursivas para los ejemplares de esa función en nuestro programa. Si la función devuelve un valor, lo único que necesitamos hacer es verificar que se satisfagan las siguientes tres propiedades:

criterio para funciones que devuelven un valor

1. No hay recursión infinita. (Una llamada recursiva puede dar pie a otra llamada recursiva y ésa puede dar pie a otra y así sucesivamente, pero toda cadena de llamadas recursivas deberá llegar tarde o temprano a un caso de paro.)
2. Todos los casos de paro devuelven el valor correcto para ese caso.
3. Para los casos que implican recursión: *si* todas las llamadas recursivas devuelven el valor correcto, *entonces* el valor final devuelto por la función es el valor correcto.

Por ejemplo, consideremos la función `potencia` del cuadro 13.3:

1. *No hay recursión infinita:* El segundo argumento de `potencia(x, n)` se decrementa en uno en cada llamada recursiva, así que cualquier cadena de llamadas recursivas deberá llegar tarde o temprano al caso `potencia(x, 0)`, que es el caso de paro. Por tanto, no hay recursión infinita.
2. *Todos los casos de paro devuelven el valor correcto para ese caso:* El único caso de paro es `potencia(x, 0)`. Una llamada de la forma `potencia(x, 0)` siempre devuelve `1`, y el valor correcto de x^0 es 1. Por tanto, el caso de paro devuelve el valor correcto.
3. *Para los casos que implican recursión: si todas las llamadas recursivas devuelven el valor correcto*, *entonces* el valor final devuelto por la función es el valor correcto: El único caso que implica recursión es cuando `n > 1`. Cuando `n > 1`, `potencia(x, n)` devuelve

   `potencia(x, n − 1)*x.`

   Para constatar que éste es el valor devuelto correcto, observe que: *si* `potencia(x, n − 1)` devuelve el valor correcto, *entonces* `potencia(x, n − 1)` devuelve x^{n-1} y por tanto `potencia(x, n)` devuelve

   $x^{n-1} * x$, que es x^n

   y ése es el valor correcto de `potencia(x, n)`.

Esto es todo lo que hay que verificar para asegurarse de que la definición de `potencia` sea correcta. (La técnica anterior se llama *inducción matemática*, concepto que tal vez haya conocido en sus clases de matemáticas. Sin embargo, no es necesario estar familiarizado con el término *inducción matemática* para usar esta técnica.)

Ya dimos tres criterios para verificar la corrección de una función recursiva que devuelve un valor. Básicamente se pueden aplicar las mismas reglas a una función *`void`* recursiva.

Si demostramos que la definición de nuestra función *void* recursiva satisface los tres criterios siguientes, sabremos que la función opera correctamente:

criterio para funciones *void*

1. No hay recursión infinita.
2. Todos los casos de paro realizan la acción correcta para ese caso.
3. Para los casos que implican recursión: *si* todas las llamadas recursivas realizan sus acciones correctamente, *entonces* todo el caso opera correctamente.

CASO DE ESTUDIO Búsqueda binaria—Un ejemplo de razonamiento recursivo

En este estudio de caso crearemos una función recursiva que examina un arreglo para averiguar si contiene o no un valor dado. Por ejemplo, el arreglo podría contener una lista de los números de tarjetas de crédito que ya no son válidas. El empleado de una tienda necesita examinar la lista para determinar si la tarjeta de un cliente es válida o no. En el capítulo 10 (cuadro 10.10) vimos un método sencillo para buscar en un arreglo que consistía simplemente en examinar cada elemento del arreglo. En esta sección desarrollaremos un método de búsqueda en un arreglo ordenado que es mucho más rápido.

Los índices del arreglo `a` son los enteros de `0` hasta `indice_final`. Para facilitar la tarea de buscar en un arreglo, supondremos que el arreglo está ordenado. Por tanto, sabemos lo siguiente:

```
a[0] <= a[1] <= a[2] <= ... <= a[indice_final]
```

Al buscar en un arreglo, lo más seguro es que queramos saber si el valor está en la lista o no y, si está, en qué lugar de la lista está. Por ejemplo, si buscamos un número de tarjeta de crédito, el índice del arreglo podría servir como número de registro. Otro arreglo indizado con estos mismos índices podría contener un número telefónico u otra información que se usaría para avisar de la tarjeta sospechosa. Así pues, si el valor buscado está en el arreglo, querremos que nuestra función nos diga en qué lugar del arreglo está.

Definición del problema

Diseñaremos nuestra función de modo que use dos parámetros de llamada por referencia para devolver el resultado de la búsqueda. Un parámetro, `hallado`, será de tipo *`bool`*. Si se encuentra el valor, se asignará *`true`* a `hallado`, y a otro parámetro, llamado `posicion`, se asignará el índice del valor encontrado. Si usamos `clave` para denotar el valor que estamos buscando, la tarea a realizar se puede formular de forma precisa:

Precondición: `a[0]` hasta `a[indice_final]`
están ordenados de menor a mayor.
Postcondición: si `clave` no es uno de los valores de `a[0]` hasta `a[indice_final]`, entonces `hallado == false`; de lo contrario `a[posicion] == clave` y `hallado == true`.

Diseño del algoritmo

Procedamos a crear un algoritmo que resuelva esta tarea. Será útil visualizar el problema en términos muy concretos. Supongamos que la lista de números es tan larga que se necesita un libro para enumerarlos todos. De hecho, ésta es la forma en que se distribuyen los números de tarjeta de crédito no válidos a los establecimientos que no tienen acceso a computadoras. Si usted es un empleado y le presentan una tarjeta de crédito, deberá verificar si está en la lista, en cuyo caso no es válida. ¿Cómo procedería? Abriría el libro a la mitad y vería si el número está ahí. Si no está y es menor que el número de en medio, lo buscaría hacia el principio del libro. Si el número es mayor que el número de en medio, lo buscaría hacia el final del libro. Esta idea produce nuestro primer borrador de algoritmo:

algoritmo primera versión

```
hallado = false; //por lo pronto.
mitad = punto medio aproximado entre 0 e indice_final;
if (clave == a[mitad])
{
    hallado = true;
    posicion = mitad;
}
else if (clave < a[mitad])
    buscar desde a[0] hasta a[mitad - 1];
else if (clave > a[mitad])
    buscar desde a[mitad + 1] hasta a[indice_final];
```

Puesto que los exámenes de las listas más cortas son versiones más pequeñas de la tarea para la cual estamos diseñando el algoritmo, este problema se presta naturalmente al uso de la recursión. Podemos examinar las listas más pequeñas con llamadas recursivas al mismo algoritmo.

A nuestro algoritmo le falta un poco de precisión como para traducirlo fácilmente a código C++. El problema tiene que ver con las llamadas recursivas. Se muestran dos llamadas recursivas:

```
buscar desde a[0] hasta a[mitad - 1];
```

y

```
buscar desde a[mitad + 1] hasta a[indice_final];
```

más parámetros

Para implementar estas llamadas recursivas necesitamos otros dos parámetros. Una llamada recursiva especifica que se examine un subintervalo del arreglo. En un caso se trata de los elementos con índices desde `0` hasta `mitad - 1`. En el otro caso se trata de los elementos con índices desde `mitad + 1` hasta `indice_final`. Los dos parámetros extra especificarán el primer y el último índices para la búsqueda, así que los llamaremos `primero` y `ultimo`. Si usamos estos parámetros para los índices más bajo y más alto en lugar de `0` e `indice_final`, podremos expresar el pseudocódigo de forma más precisa como sigue:

algoritmo primera refinación

```
Para buscar desde a[primero] hasta a[ultimo] hacer lo siguiente:
hallado = false; //por lo pronto.
mitad = punto medio aproximado entre primero y ultimo;
if (clave == a[mitad])
{
    hallado = true;
    posicion = mitad;
}
```

```
else if (clave < a[mitad])
    buscar desde a[primero] hasta a[mitad - 1];
else if (clave > a[mitad])
    buscar desde a[mitad + 1] hasta a[ultimo];
```

Para examinar todo el arreglo, el algoritmo se ejecutaría con `primero` igual a `0` y con `ultimo` igual a `indice_final`. Las llamadas recursivas usarán otros valores para `primero` y `ultimo`. Por ejemplo, la primera llamada recursiva asignará `0` a `primero` y el valor calculado `mitad - 1` a `ultimo`.

caso de paro

Al igual que con cualquier algoritmo recursivo es preciso asegurarse de que el algoritmo termine y no produzca recursión infinita. Si el número buscado se encuentra en la lista, no habrá llamada recursiva y el proceso terminará, pero necesitamos alguna forma de detectar si el número no está en la lista. Si `primero` llega a tener un valor mayor que el de `ultimo`, sabremos que no hay más índices que verificar y que el número `clave` no está en el arreglo. Si añadimos esta prueba a nuestro pseudocódigo, obtendremos una solución completa, la cual se muestra en el cuadro 13.5.

algoritmo-versión final

CUADRO 13.5 *Pseudocódigo para búsqueda binaria*

```
int a[Some_Size_Value];

//Precondición:
//a[primero] <= a[primero + 1] <= a[primero + 2] <= ... <= a[ultimo]

Para localizar el valor clave:

if (primero > ultimo) //Un caso de paro
    hallado = false;
else
{
    mitad = punto medio aproximado entre primero y ultimo;
    if (clave == a[mitad]) //Un caso de paro
    {
            hallado = true;
            posicion = mitad;
    }
    else if (clave < a[mitad]) //Un caso con recursión
            buscar desde a[primero] hasta a[mitad - 1];
    else if (clave > a[mitad]) //Un caso con recursión
            buscar desde a[mitad + 1] hasta a[ultimo];
}
```

Codificación

Ahora es fácil traducir el pseudocódigo a código C++. El resultado se muestra en el cuadro 13.6. La función `buscar` es una implementación del algoritmo recursivo del cuadro 13.5. En el cuadro 13.7 se muestra un diagrama de la operación de la función con un arreglo de muestra.

resolver un problema más general

Observe que la función `buscar` resuelve un problema más general que la tarea original. Nuestra meta era diseñar una función para buscar en un arreglo entero. Esta función nos permite examinar cualquier intervalo del arreglo especificando los límites del índice `primero` y `ultimo`. Esto es común al diseñar funciones recursivas. Con frecuencia es necesario resolver un problema más general para poder expresar el algoritmo recursivo. En este caso, sólo queríamos la respuesta en el caso en el que `primero` es igual a `0` y `ultimo` es igual a `indice_final`. Sin embargo, las llamadas recursivas les asignarán otros valores diferentes de `0` e `indice_final`.

Verificación de la recursión

En la subsección intitulada "Técnicas de diseño recursivo" dimos tres criterios que debemos verificar para asegurarnos de que la definición de una función *`void`* recursiva sea correcta. Verifiquemos esas tres cosas para la función `buscar` del cuadro 13.6.

1. *No hay recursión infinita:* En cada llamada recursiva el valor de `primero` se incrementa o el valor de `ultimo` se decrementa. Si la cadena de llamadas recursivas no termina de alguna otra manera, llegará un momento en que la función se invoque con `primero` mayor que `ultimo`, y ése es un caso de paro.

2. *Todos los casos de paro realizan la acción correcta para ese caso:* Hay dos casos de paro, cuando `primero > ultimo` y cuando `clave == a[mitad]`. Consideremos cada caso.

   Si `primero > ultimo`, no habrá elementos del arreglo entre `a[primero]` y `a[ultimo]` y `clave` no estará en este segmento del arreglo. (¡No hay nada en este segmento del arreglo!) Así pues, si `primero > ultimo`, la función `buscar` establece correctamente *`false`* a `hallado`.

   Si `clave == a[mitad]`, el algoritmo establece correctamente *`true`* a `hallado` y `mitad` a `posicion`. Por tanto, ambos casos de paro son correctos.

3. *Para los casos que implican recursión: si todas las llamadas recursivas realizan sus acciones correctamente, entonces todo el caso opera correctamente:* Hay dos casos en los que se efectúan llamadas recursivas, cuando `clave < a[mitad]` y cuando `clave > a[mitad]`. Necesitamos verificar ambos casos.

   Supongamos primero que `clave < a[mitad]`. En este caso, puesto que el arreglo está ordenado, sabemos que si `clave` está en el arreglo será uno de los elementos de `a[primero]` hasta `a[mitad - 1]`. Por lo tanto, la función sólo necesita examinar esos elementos, y eso es exactamente lo que hace la llamada recursiva

   ```
   buscar(a, primero, mitad - 1, clave, hallado, posicion);
   ```

   Así pues, si la llamada recursiva es correcta, toda la acción es correcta.

   Supongamos ahora que `clave > a[mitad]`. En este caso, puesto que el arreglo está ordenado, sabemos que si `clave` está en el arreglo será uno de los elementos de

CUADRO 13.6 *Función recursiva para búsqueda binaria (parte 1 de 2)*

```
//Programa para demostrar la función recursiva de búsqueda binaria.
#include <iostream>
using namespace std;
const int TAM_ARREGLO = 10;


void buscar(const int a[], int primero, int ultimo,
                   int clave, bool& hallado, int& posicion);
//Precondición: a[primero] hasta a[ultimo] están ordenados de menor a mayor.
//Postcondición: Si clave no es uno de los valores de a[primero] a a[ultimo],
//hallado == false; en caso contrario, a[posicion] == clave y hallado == true.


int main()
{
    int a[TAM_ARREGLO];
    const int indice_final = TAM_ARREGLO - 1;

        <Esta porción del programa contiene código para llenar
        y ordenar el arreglo a. Los detalles exactos no vienen al caso en este ejemplo.>

    int clave, posicion;
    bool hallado;
    cout << "Teclee el numero que se busca: ";
    cin >> clave;
    buscar(a, 0, indice_final, clave, hallado, posicion);

    if (hallado)
        cout << clave << " esta en la posicion con indice "
             << posicion << endl;
    else
        cout << clave << " no esta en el arreglo." << endl;

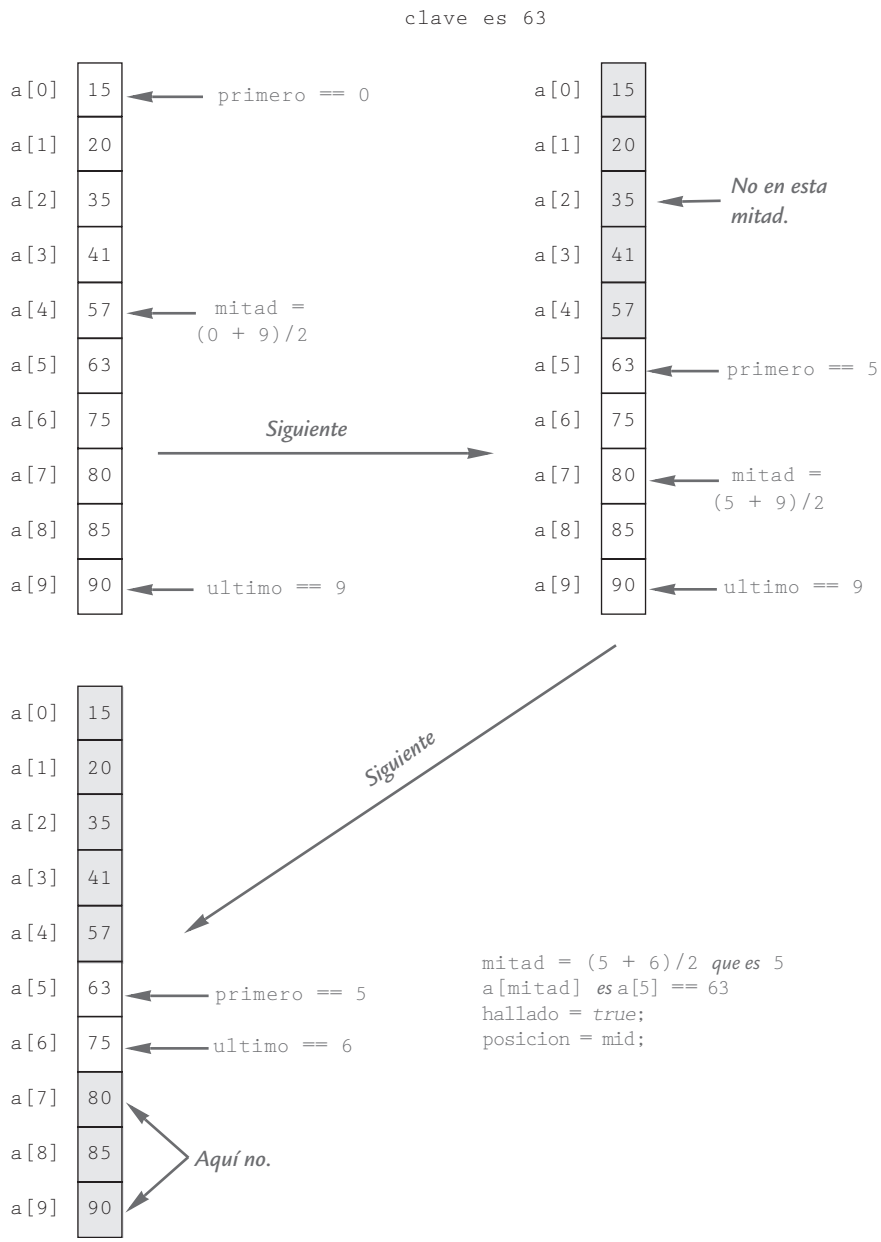
    return 0;
}
```

CUADRO 13.6 *Función recursiva para búsqueda binaria (parte 2 de 2)*

```
void buscar(const int a[], int primero, int ultimo,
                           int clave, bool& hallado, int& posicion)
{
    int mitad;
    if (primero > ultimo)
    {
        hallado = false;
    }
    else
    {
        mitad = (primero + ultimo)/2;

        if (clave == a[mitad])
        {
            hallado = true;
            posicion = mitad;
        }
        else if (clave < a[mitad])
        {
            buscar(a, primero, mitad - 1, clave, hallado, posicion);
        }
        else if (clave > a[mitad])
        {
            buscar(a, mitad + 1, ultimo, clave, hallado, posicion);
        }
    }
}
```

CUADRO 13.7 *Ejecución de la función* buscar



a[mitad + 1] hasta a[ultimo]. Por lo tanto, la función sólo necesita examinar esos elementos, y eso es exactamente lo que hace la llamada recursiva

```
buscar(a, mitad + 1, ultimo, clave, hallado, posicion);
```

Así pues, si la llamada recursiva es correcta, toda la acción es correcta. Por tanto, en ambos casos la función realiza la acción correcta (suponiendo que las llamadas recursivas realizan la acción correcta).

La función buscar pasa nuestras tres pruebas, así que es una buena definición de función recursiva.

Eficiencia

El algoritmo de búsqueda binaria es en extremo rápido en comparación con un algoritmo que simplemente prueba todos los elementos del arreglo en orden. En la búsqueda binaria descartamos de la consideración cerca de la mitad del arreglo como primer paso. Luego descartamos una cuarta parte del arreglo, luego un octavo, y así sucesivamente. Esto hace que el algoritmo sea extraordinariamente rápido. Si el arreglo tiene 100 elementos, la búsqueda binaria nunca necesitará comparar con la clave más de siete elementos del arreglo. Una búsqueda en serie sencilla podría comparar hasta 100 elementos del arreglo con la clave, y en promedio comparará unos 50 elementos del arreglo con la clave. Además, cuanto mayor sea el arreglo, más impresionante será el ahorro de tiempo. Con un arreglo de 1000 elementos la búsqueda binaria sólo necesita comparar unos 10 elementos con el valor clave, en comparación con un promedio de 500 para el algoritmo de búsqueda en serie simple.

versión iterativa

En el cuadro 13.8 se da una versión iterativa de la función buscar. En algunos sistemas la versión iterativa es más eficiente que la recursiva. El algoritmo de la versión iterativa se obtuvo imitando la versión recursiva. En la versión iterativa, las variables locales primero y ultimo reflejan los papeles de los parámetros en la versión recursiva, que también se llaman primero y ultimo. Como ilustra este ejemplo, en muchos casos es conveniente desarrollar un algoritmo recursivo aunque después vayamos a convertirlo en un algoritmo iterativo.

EJEMPLO DE PROGRAMACIÓN
Una función miembro recursiva

Una función miembro de una clase puede ser recursiva. Las funciones miembro pueden usar la recursión exactamente igual que las funciones ordinarias. El cuadro 13.9 contiene un ejemplo de función miembro recursiva. La clase CuentaBancaria que se usa en esa cuadro es la misma clase que se definió en el cuadro 6.6, excepto que hemos sobrecargado el nombre de función miembro actualizar. La primera versión de actualizar no tiene argumentos y suma un año de intereses simples al saldo de la cuenta bancaria. La otra versión de actualizar (la nueva) recibe un argumento *int* que es algún número de años. Esta función miembro actualiza la cuenta sumando al saldo los intereses correspondientes a ese número de años. La nueva versión de actualizar es recursiva. Esta nueva función tiene un parámetro llamado anios y usa el siguiente algoritmo:

Si el número de anios es 1, entonces //*Caso de paro:*
    llamar a la otra función llamada actualizar (la que no tiene argumentos).

Si el número de anios es mayor que 1, entonces //*Caso recursivo:*
    hacer una llamada recursiva para sumar anios − 1 de intereses, y luego llamar a la otra función actualizar (la que no tiene argumentos) para sumar un año más de intereses.

CUADRO 13.8 *Versión iterativa de la búsqueda binaria*

Declaración de función

```
void buscar(const int a[], int extremo_bajo, int extremo_alto,
                           int clave, bool& hallado, int& posicion);
//Precondición: a[extremo_bajo] hasta a[extremo_alto] están
//ordenados de menor a mayor.
//Postcondición: Si clave no es uno de los valores de
//a[extremo_bajo] a a[extremo_alto], entonces hallado == false;
//en caso contrario, a[posicion] == clave y hallado == true.
```

Definición de función

```
void buscar(const int a[], int extremo_bajo, int extremo_alto,
                           int clave, bool& hallado, int& posicion)
{
    int primero = extremo_bajo;
    int ultimo = extremo_alto;
    int mitad;

    hallado = false; //hasta ahora
    while ( (primero <= ultimo) && !(hallado) )
    {
        mitad = (primero + ultimo)/2;
        if (clave == a[mitad])
        {
            hallado = true;
            posicion = mitad;
        }
        else if (clave < a[mitad])
        {
             ultimo = mitad - 1;
        }
        else if (clave > a[mitad])
        {
             primero = mitad + 1;
        }
    }
}
```

CUADRO 13.9 *Una función miembro recursiva (parte 1 de 2)*

```
//Programa para demostrar la función miembro recursiva actualizar(anios).
#include <iostream>
using namespace std;

//Clase para una cuenta bancaria:
class CuentaBancaria
{
public:
    CuentaBancaria(int pesos, int centavos, double tasa);
    //Inicializa el saldo de la cuenta con $pesos.centavos e
    //Inicializa la tasa de interés con tasa por ciento.

    CuentaBancaria(int pesos, double tasa);
    //Inicializa el saldo de la cuenta con $pesos.00 e
    //inicializa la tasa de interés con tasa por ciento.

    CuentaBancaria();
    //Inicializa el saldo con $0.00 e
    //inicializa la tasa de interés con 0.0%.

    void actualizar();
    //Postcondición: Se ha sumado un año de interés
    //simple al saldo de la cuenta.

    void actualizar(int anios);
    //Postcondición: Se han sumado al saldo de la cuenta intereses
    //por el número de años dado. El interés es compuesto anual.

    double obtener_saldo();
    //Devuelve el saldo actual de la cuenta.

    double obtener_tasa();
    //Devuelve la tasa de interés vigente como un porcentaje.

    void salida(ostream& sale);
    //Precondición: Si sale es un flujo de archivo de salida, ya
    //se ha conectado a un archivo.
    //Postcondición: El saldo de la cuenta y la tasa de interés se han escrito en
    //el flujo sale.
private:
    double saldo;
    double tasa_interes;
    double fraccion(double por ciento); //Convierte un porcentaje en fracción.
};
```

La clase `CuentaBancaria` *de este programa es una versión mejorada de la que se da en el cuadro 6.6.*

Dos funciones distintas con el mismo nombre.

CUADRO 13.9 *Una función miembro recursiva (parte 2 de 2)*

```
int main()
{
    CuentaBancaria tu_cuenta(100, 5);
    tu_cuenta.actualizar(10);
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Si depositas $100.00 al 5% de interes,\n"
         << "en diez años tu cuenta ascendera a $"
         << tu_cuenta.obtener_saldo() << endl;
    return 0;
}

void CuentaBancaria::actualizar()
{
    saldo = saldo + fraccion(tasa_interes)*saldo;
}

void CuentaBancaria::actualizar(int anios)
{
    if (anios == 1)
    {
        actualizar();
    }
    else if (anios > 1)
    {
        actualizar(anios - 1);
        actualizar();
    }
}
```

Sobrecarga (es decir, llamadas a otra función del mismo nombre).

Llamada de función recursiva.

<Las definiciones de las demás funciones miembro se dan en los cuadros 6.5 y 6.6, pero no es necesario leer esas definiciones para entender este ejemplo.>

Diálogo de ejemplo

```
Si depositas $100.00 al 5% de interes,
en diez años tu cuenta ascendera a $162.89
```

Es fácil ver que este algoritmo produce el resultado deseado verificando los tres puntos que dimos en la subsección "Técnicas de diseño recursivo".

1. ***No hay recursión infinita:*** Cada llamada recursiva reduce el número de años en uno hasta que `anios` tiene el valor `1`, lo que es el caso de paro. Por lo tanto, no hay recursión infinita.
2. ***Todos los casos de paro realizan la acción correcta para ese caso:*** El único caso de paro es cuando `anios == 1`. Este caso produce la acción correcta porque simplemente invoca la otra función miembro sobrecargada llamada `actualizar`, y ya verificamos la corrección de esa función en el capítulo 6.
3. ***Para los casos que implican recursión: si todas las llamadas recursivas realizan sus acciones correctamente, entonces todo el caso opera correctamente:*** Es obvio que el caso recursivo, `anios > 1`, funciona correctamente porque si la llamada recursiva suma correctamente `anios - 1` de intereses, lo único que se necesita es sumar un año más de intereses, y la llamada a la versión sobrecargada `actualizar` con cero argumentos sumará correctamente un año de intereses. Por lo tanto, *si* la llamada recursiva realiza la acción correcta, *entonces* toda la acción para el caso de `anios > 1` será correcta.

sobrecarga

En este ejemplo hemos sobrecargado `actualizar`, así que hay dos funciones distintas con ese nombre: una que no recibe argumentos y una que recibe un solo argumento. No confunda las llamadas a las dos funciones `actualizar`. Se trata de dos funciones distintas y, en lo que al compilador concierne, es sólo por casualidad que tienen el mismo nombre. El hecho de que la definición de la función `actualizar` con un argumento incluya una llamada a la versión de `actualizar` que no recibe argumentos no implica recursión. Sólo la llamada a la versión de `actualizar` que tiene *exactamente* la misma declaración de función es una llamada recursiva. Para entender mejor esto, consideremos que podríamos haber llamado a la versión de `actualizar` que no recibe argumentos `sumar_un_anio_ de_intereses()`, en lugar de llamarla `actualizar()`, y entonces la definición de la versión recursiva de actualizar sería:

```
void CuentaBancaria::actualizar(int anios)
{
     if (anios == 1)
     {
         sumar_un_anio_de_intereses();
     }
     else if (anios > 1)
     {
          actualizar(anios - 1);
          sumar_un_anio_de_intereses();
     }
}
```

Recursión y sobrecarga

No confunda la recursión con la sobrecarga. Cuando sobrecargamos un nombre de función estamos dando el mismo nombre a dos funciones distintas. Si la definición de una de estas funciones incluye una llamada a la otra, no es recursión. En una función recursiva la definición de la función incluye una llamada a la misma función *exactamente*, con la *misma definición exactamente*, no simplemente una llamada a alguna otra función que por coincidencia usa el mismo nombre. No es un error demasiado grave confundir la sobrecarga con la recursión, ya que ambas son válidas. Simplemente es cuestión de usar de manera correcta la terminología para podernos comunicar claramente con otros programadores, y para entender los procesos subyacentes.

Ejercicios de AUTOEVALUACIÓN

15. Escriba una definición de función recursiva para la siguiente función:

```
int cuadrados(int n);

//Precondición: n >= 1
//Devuelve la suma de los cuadrados de los números 1 hasta n.
```

Por ejemplo, `cuadrados(3)` devuelve `14`, porque $1^2 + 2^2 + 3^2 = 14$.

16. Escriba una versión iterativa de la función miembro con un argumento `CuentaBancaria::actualizar(int anios)` que se describe en el cuadro 13.9.

Resumen del capítulo

- Si un problema se puede reducir a casos más pequeños del mismo problema, es probable que la implementación de una solución recursiva sea fácil.
- Un algoritmo recursivo para una definición de función normalmente contiene dos tipos de casos: uno o más casos que incluyen al menos una llamada recursiva, y uno o más casos de paro en los que el problema se resuelve sin llamadas recursivas.
- Al escribir una definición de función recursiva, siempre verifique que la función no producirá recursión infinita.
- Al definir una función recursiva, use los tres criterios dados en la subsección "Técnicas de diseño recursivo" para verificar que la función sea correcta.
- Al diseñar una función recursiva para resolver una tarea, suele ser necesario resolver un problema más general que la tarea dada. Esto podría requerirse para poder efectuar las llamadas recursivas correctas, ya que los problemas más pequeños podrían no ser exactamente el mismo problema que la tarea dada. Por ejemplo, en el problema de búsqueda binaria la tarea era buscar en todo un arreglo, pero la solución recursiva es un algoritmo para buscar en cualquier porción de un arreglo (todo o una parte).

Respuestas a los ejercicios de autoevaluación

1. `Hip Hip Hurra`

2.
```
void asteriscos(int n)
{
    cout << '*';
    if (n > 1)
        asteriscos(n - 1);
}
```

   Lo que sigue también es correcto, pero es más complicado:

```
void asteriscos(int n)
{
     if (n <= 1)
     {
         cout << '*';
     }
     else
     {
         asteriscos(n - 1);
         cout << '*';
     }
}
```

3.
```
void alreves(int n)
{
     if (n < 10)
     {
         cout << n;
     }
     else
     {
         cout (n%10); //escribir el último dígito
         alreves(n/10); //escribir los demás dígitos al revés
     }
}
```

4-5. La respuesta a 4 es `creciente(int n);`, la respuesta a 5 es `decreciente(int n);`

```
#include <iostream>
using namespace std;
void decreciente(int n)
{
     if (n >= 1)
     {
         cout << n << " ";
         decreciente(n - 1);
     }
}
```

```
void creciente(int n)
{
     if (n >= 1)
     {
         creciente(n - 1);
         cout << n << " ";
     }
}
//código para probar tanto #4 como #5
int main()
{
    cout << "llamando a creciente(" << 10 << ")\n";
    creciente(10);
    cout << endl;
    cout << "llamando a decreciente(" << 10 << ")\n";
    decreciente(10);
    cout << endl;
    return 0;
}
/* Resultados de la prueba
llamado a creciente(10)
1 2 3 4 5 6 7 8 9 10
llamado a decreciente(10)
10 9 8 7 6 5 4 3 2 1
*/
```

6. Un mensaje de error que indica *desbordamiento de pila* nos está diciendo que la computadora trató de colocar en la pila más marcos de activación de los que se permiten en el sistema. Una causa probable de este mensaje de error es una recursión infinita.

7.
```
void porra(int n)
{
     while (n > 1)
     {
         cout << "Hip ";
         n--;
     }
     cout << "Hurra\n";
}
```

8.
```
void asteriscos(int n)
{
     for (int cuanta = 1; cuenta <= n; cuenta++)
         cout << '*';
}
```

9.
```
void alreves(int n)

{
     while (n >= 10)
     {
         cout << (n%10); //escribir el último dígito
         n = n/10; //desechar el último dígito
     }
     cout << n;
}
```

10. Rastreo del ejercicio 4: Si n es 3, el código a ejecutar es:

```
if (3 >= 1)
{
creciente(3 – 1);
cout << 3 << " ";
}
```

En la siguiente recursión n = 2; el código a ejecutar es:

```
if (2 >= 1)
{
creciente(2 – 1);
cout << 2 << " ";
}
```

En la siguiente recursión n = 1; el código a ejecutar es:

```
if (1 >= 1)
{
creciente(1 – 1);
cout << 1 << " ";
}
```

En la recursión final n = 0 y el código a ejecutar es:

```
if (0 >= 1) //condición falsa, no se ejecuta el cuerpo
{
   //se pasa por alto
}
```

Las recursiones inician el regreso; la salida (que se obtuvo mientras las recursiones se acumulaban) es 1 2 3.

11. Rastreo del ejercicio 5: Si n = 3, el código a ejecutar es:

```
if (3 >= 1)
{
    cout << 3 << " ";
    decreciente(3 – 1);
}
```

Siguiente recursión, n = 2, el código a ejecutar es:

```
if (2 >= 1)
{
    cout << 2 << " ";
    decreciente(2 – 1);
}
```

Siguiente recursión, n = 1, el código a ejecutar es:

```
if (1 >= 1)
{
    cout << 1 << " ";
    decreciente(1 – 1);
}
```

Recursión final, `n = 0` y no se ejecuta la cláusula "*verdadero*":

```
if (0 >= 1)    // condición falsa
{
   // esta cláusula se pasa por alto
}
```

La salida es `3 2 1`.

12. `6`

13. La salida es `24`. La función es la función factorial, que normalmente se escribe *n*! y que se define como sigue:

    n! es igual a *n**(*n*−1)*(*n*−2)* ... *1

14.
```
//Usa iostream y cstdlib:
double potencia(int x, int n)
{
   if (n < 0 && x == 0)
   {
       cout << "Argumento no válido para potencia.\n";
       exit(1);
   }
   if (n < 0)
       return ( 1/potencia(x, −n) );
   else if (n > 0)
       return ( potencia(x, n − 1)*x );
   else  // n == 0
       return (1.0);
}
```

15.
```
int cuadrados(int n)
{
    if (n <= 1)
        return 1;
    else
       return (cuadrados(n − 1) + n*n );
}
```

16.
```
void CuentaBancaria::actualizar(int anios)
{
    for (int cuenta = 1; cuenta <= anios; cuenta++)
         actualizar();
}
```

Proyectos de programación

1. Escriba una definición recursiva de una función que tiene un parámetro `n` de tipo `int` y que devuelve el `n`-ésimo número de Fibonacci. La definición de los números de Fibonacci se da en el proyecto de programación 10 del capítulo 7. Incorpore la función en un programa y pruébela.

2. Escriba una versión recursiva de la función `indice_del_menor` que se usó en el programa de ordenamiento del cuadro 10.12 del capítulo 10. Incorpore la función en un programa y pruébela.

3. Escriba una función recursiva para la función `buscar` del cuadro 10.10 del capítulo 10

4. La fórmula para calcular el número de formas de escoger *r* cosas distintas de un conjunto de *n* cosas es:

   $C(n, r) = n!/(r!*(n - r)!)$

   La función factorial *n*! se define como

   $n! = n*(n - 1)*(n - 2)* \ldots *1$.

   Descubra una versión recursiva de la fórmula anterior y escriba una función recursiva que calcule el valor de la fórmula. Incorpore la función en un programa y pruébela.

5. 
   Escriba una función recursiva que reciba un argumento que es un arreglo de caracteres y dos argumentos que sean límites para el índice del arreglo. La función deberá invertir el orden de los elementos del arreglo cuyo índice esté entre los dos límites. Por ejemplo, si el arreglo es:

   ```
   a[1] == 'A'  a[2] == 'B'  a[3] == 'C'  a[4] == 'D'  a[5] == 'E'
   ```

   y los límites son `2` y `5`, entonces después de ejecutarse la función los elementos del arreglo deberán ser:

   ```
   a[1] == 'A'  a[2] == 'E'  a[3] == 'D'  a[4] == 'C'  a[5] == 'B'
   ```

   Incorpore la función en un programa y pruébela. Después de haber depurado cabalmente esta función, defina otra función que reciba un solo argumento que es un arreglo que contiene un valor de cadena, y que invierta los caracteres del valor de cadena del argumento. Esta función incluirá una llamada a la definición recursiva que se obtuvo en la primera parte de este proyecto. Incorpore esta segunda función en un programa y pruébela.

6. Escriba una versión iterativa de la función recursiva del proyecto de programación 4. Incorpore la función en un programa y pruébela.

7. Escriba una función recursiva que ordene un arreglo de enteros de menor a mayor utilizando la siguiente idea: coloque el elemento más pequeño en la primera posición, y luego ordene el resto del arreglo con una llamada recursiva. Ésta es una versión recursiva del algoritmo de ordenamiento por selección que vimos en el capítulo 10. (*Nota:* No basta con tomar el programa del capítulo 10 e insertar una versión recursiva de `indice_del_menor`. La función que realiza el ordenamiento debe ser recursiva por sí misma, no sólo usar una función recursiva.)

8. 
   Torres de Hanoi. Existe una leyenda acerca de unos monjes budistas que están resolviendo este acertijo con 64 discos de piedra. Según la leyenda, cuando los monjes terminen de pasar los discos de un poste a otro usando un tercer poste, el tiempo llegará a su fin.

   Una pila de `n` discos de tamaño decreciente está ensartada en uno de tres postes. La tarea consiste en pasar los discos uno por uno del primer poste al segundo. Para hacerlo, se puede pasar cualquier disco de cualquier poste a cualquier otro poste, sujeto a la regla de que nunca se puede colocar un disco más grande encima de uno más pequeño. Se proporciona un tercer poste (de reserva) para hacer posible la resolución. El proyecto consiste en escribir una función recursiva que describa instrucciones para resolver este problema. No podemos usar gráficos, así que deberá desplegarse en la pantalla una sucesión de instrucciones que resuelva el problema.

   Sugerencia: Si pudiéramos pasar $n - 1$ de los discos del primer poste al tercero usando el segundo como reserva, podríamos pasar el último disco del primer poste al segundo. Luego, utilizando la misma técnica (sea cual fuere) podríamos pasar los $n - 1$ discos del tercer poste al segundo, usando el primer poste como reserva. ¡Presto! Problema resuelto. Sólo hay que decidir cuál es el caso no recursivo, cuál es el recursivo, y cuándo desplegar las instrucciones para trasladar los discos.

14 Plantillas

14 Plantillas

Todos los hombres son mortales.
Aristóteles es un hombre.
Por tanto, Aristóteles es mortal.
Todas las X son Y.
Z es una X.
Por tanto, Z es Y.
Todos los gatos son traviesos.
Garfield es un gato.
Por tanto, Garfield es travieso.

UNA LECCIÓN CORTA SOBRE SILOGISMOS

Introducción

En este capítulo veremos las plantillas de C++. Las plantillas nos permiten definir funciones y clases que tienen parámetros para nombres de tipos; lo que nos permite diseñar funciones que puedan emplearse con argumentos de diferentes tipos y definir clases mucho más generales que las que hemos visto hasta ahora.

Prerrequisitos

En la sección 14.1 emplearemos material de los capítulos 2, 3, 4 y 7, y las secciones 10.1, 10.2 y 10.3. No emplearemos ningún material de clases. La sección 14.2 emplea material de los capítulos 2 al 10.

14.1 Plantillas para abstracción de algoritmos

Muchas de las definiciones de funciones C++ que hemos escrito han tenido un algoritmo subyacente que es mucho más general que el que dimos en la definición de la función. Por ejemplo, consideremos la función `intercambiar_valores` que vimos por primera vez en el capítulo 4. Como referencia, repetiremos la definición de la función:

```
void intercambiar_valores(int& variable1, int& variable2)
{
     int temp;

     temp = variable1;
     variable1 = variable2;
     variable2 = temp;
}
```

Observe que la función `intercambiar_valores` aplica sólo a variables de tipo *`int`*. Sin embargo, el algoritmo que damos en el cuerpo de la función se podría usar para intercambiar los valores de dos variables de tipo *`char`*. Si queremos usar también la función `intercambiar_valores` con variables de tipo *`char`*, podemos sobrecargar el nombre de función `intercambiar_valores` añadiendo la siguiente definición:

```
void intercambiar_valores(char& variable1, char& variable2)
{
     char temp;

     temp = variable1;
     variable1 = variable2;
     variable2 = temp;
}
```

Sin embargo, estas dos definiciones de la función `intercambiar_valores` algo tienen de ineficiente e insatisfactorio: son casi idénticas. La única diferencia es que una definición usa el tipo *`int`* en tres lugares y la otra usa el tipo *`char`* en los mismos tres lugares. Si continuáramos así y quisiéramos aplicar la función `intercambiar_valores` a variables de tipo `double`, tendríamos que escribir una tercera definición de función casi idéntica. Si queremos aplicar `intercambiar_valores` a más tipos, el número de definiciones de función casi idénticas sería todavía mayor. Esto requeriría una buena cantidad de tecleo y atiborraría nuestro código con muchas definiciones casi idénticas. Sería bueno poder decir que la siguiente definición de función aplica a variables de cualquier tipo:

```
    void intercambiar_valores(Tipo_de_las_Variables&
variable1,                        Tipo_de_las_Variables& variable2)
    {
          Tipo_de_las_Variables temp;

          temp = variable1;
          variable1 = variable2;
          variable2 = temp;
    }
```

Como veremos, podemos hacer algo por el estilo. Podemos definir una función que se aplique a todo tipo de variables, aunque la sintaxis es un poco diferente de la que acabamos de mostrar. Describiremos dicha sintaxis en la siguiente subsección.

Plantillas para funciones

El cuadro 14.1 muestra una plantilla C++ para la función `intercambiar_valores`. Esta plantilla de función nos permite intercambiar los valores de dos variables cualesquiera, de cualquier tipo, en tanto las dos tengan el mismo tipo. La definición y la declaración de función comienzan con la línea

```
template<class T>
```

prefijo de plantilla
plantilla
parámetro de tipo

Esto se conoce como **prefijo de plantilla**, y le dice a la computadora que la definición o declaración de función que sigue es una **plantilla** y que `T` es un **parámetro de tipo**. En este contexto, la palabra *`class`* en realidad significa *tipo*.[1] Como veremos, el parámetro de tipo `T` puede sustituirse por cualquier tipo, sea éste una clase o no. Dentro del cuerpo de la definición de función, el parámetro de tipo `T` se usa igual que cualquier otro tipo.

una plantilla sobrecarga el nombre de la función

La definición de plantilla de función es, de hecho, una colección grande de definiciones de funciones. En el caso de la plantilla de función para `intercambiar_valores` que se muestra en el cuadro 14.1, hay efectivamente una definición de función para cada posible nombre de tipo. Obtenemos cada una de estas definiciones sustituyendo el parámetro de tipo `T` por un nombre de tipo. Por ejemplo, obtenemos la siguiente definición de función sustituyendo el parámetro de tipo por el nombre de tipo *`double`*:

```
void intercambiar_valores(double& variable1, double& variable2)
{
     double temp;

     temp = variable1;
     variable1 = variable2;
     variable2 = temp;
}
```

Obtenemos otra definición de `intercambiar_valores` sustituyendo el parámetro de tipo `T` de la plantilla de función por el nombre de tipo *`int`*. Y obtenemos una definición más sustituyendo el parámetro de tipo `T` por *`char`*. La plantilla de función que se muestra en el cuadro 14.1 sobrecarga el nombre de función `intercambiar_valores` de modo que haya una definición ligeramente distinta de la función para cada tipo posible.

El compilador no producirá literalmente definiciones de la función `intercambiar_valores` para todos los tipos posibles, pero se comportará exactamente como si lo hubiera hecho. Se producirá una definición aparte para cada tipo distinto con el cual usemos la plantilla, pero no para los tipos que no usemos. Sólo se genera una definición para un tipo dado, sin importar cuántas veces se use la plantilla para ese tipo. Observe que en el cuadro 14.1 se invoca dos veces la función `intercambiar_valores`: en una ocasión los argumentos son de tipo *`int`* y en la otra son de tipo *`char`*.

[1]De hecho, el Estándar ANSI permite utilizar la palabra clave `typename` en lugar de *`class`* en el prefijo de plantilla. Si bien, estamos de acuerdo con que el uso de `typename` tiene más sentido que la palabra clave *`class`*, sin embargo, el uso de la palabra *`class`* es una tradición firmemente establecida y por lo tanto, utilizamos class para ser consistentes con la mayoría de los programadores y autores.

CUADRO 14.1 *Una plantilla de función*

```
//Programa para demostrar una plantilla de función.
#include <iostream>
using namespace std;

//Intercambia los valores de variable1 y variable2.
template<class T>
void intercambiar_valores(T& variable1, T& variable2)
{
      T temp;

      temp = variable1;
      variable1 = variable2;
      variable2 = temp;
}
int main()
{
    int entero1 = 1, entero2 = 2;
    cout << "Los valores enteros originales son "
         << entero1 << " " << entero2 << endl;
    intercambiar_valores(entero1, entero2);
    cout << "Los valores enteros intercambiados son "
         << entero1 << " " << entero2 << endl;

    char simbolo1 = 'A', simbolo2 = 'B';
    cout << "Los valores de caracter originales son "
         << simbolo1 << " " << simbolo2 << endl;
    intercambiar_valores(simbolo1, simbolo2);
    cout << "Los valores de caracter intercambiados son "
         << simbolo1 << " " << simbolo2 << endl;

    return 0;
}
```

Salida

```
Los valores enteros originales son 1 2
Los valores enteros intercambiados son 2 1
Los valores de caracter originales son A B
Los valores de caracter intercambiados son B A
```

Considere la siguiente llamada de función del cuadro 14.1:

```
intercambiar_valores(entero1, entero2);
```

Cuando el compilador de C++ llega a esta llamada de función, toma nota de los tipos de los argumentos (en este caso *int*) y usa la plantilla para producir una definición de función en la que el parámetro de tipo T se ha sustituido por el nombre de tipo *int*. De forma similar, cuando el compilador ve la llamada de función

```
intercambiar_valores(simbolo1, simbolo2);
```

toma nota de los tipos de los argumentos (en este caso *char*) y luego usa la plantilla para producir una definición de función en la que el parámetro de tipo T se ha sustituido por el nombre de tipo *char*.

Cabe señalar que no es preciso hacer nada especial para invocar una función que se define con una plantilla de función; se invoca igual que cualquier otra función. El compilador se encarga de producir la definición de función a partir de la plantilla de función.

invocación de una plantilla de función

Observe que en el cuadro 14.1 colocamos la definición de una plantilla de función antes de la parte main del programa y no utilizamos una declaración de plantilla de función. Una plantilla de función puede tener una declaración de función, igual que una función ordinaria. Podríamos (o no) colocar la declaración de función y la definición de una plantilla de función en los mismos lugares que colocamos la declaración y definiciones de funciones ordinarias. Sin embargo, muchos compiladores no soportan las declaraciones de plantillas de función y no soportan la compilación por separado de las plantillas de función. Cuando sí la soportan, los detalles pueden ser desordenados y variar de un compilador a otro. Así que lo más seguro es no utilizar declaraciones de plantillas de función y colocar la definición de nuestra plantilla de función en el archivo que las invocamos; además de procurar que aparezca antes de utilizar la plantilla de función.

Dijimos que una definición de plantilla de función debe aparecer en el mismo archivo que utiliza la plantilla de función (es decir, en el mismo archivo que invoca a dicha plantilla de función). Sin embargo, la definición de plantilla de función puede aparecer por medio de una directiva #include. Podemos dar la definición de una plantilla de función en un archivo, y después agregar la directiva #include en un archivo que utilice la plantilla de función. Ésta es la estrategia más clara y segura. Sin embargo, es probable que no funcione en todos los compiladores; si no funciona, consulte a un experto.

A pesar de que en nuestro código no utilizaremos declaraciones de plantillas de función, las describiremos y daremos algunos ejemplos como beneficio de aquellos lectores cuyos compiladores soporten el uso de las declaraciones de función.

En la plantilla de función del cuadro 14.1 usamos la letra T como parámetro de tipo. Esto es tradicional en C++, pero no obligatorio. El parámetro de tipo puede ser cualquier identificador (que no sea palabra clave). T es un buen nombre para el parámetro de tipo, pero es posible usar otros nombres. Por ejemplo, la plantilla de función intercambiar_valores que se presenta en el cuadro 14.1 es equivalente a lo siguiente:

```
template<class VariableType>
void intercambiar_valores(VariableType& variable1,
                                        VariableType& variable2)
{
     VariableType temp;
```

```
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

más que un parámetro de tipo

Es posible tener plantillas de funciones con más de un parámetro de tipo. Por ejemplo, una plantilla de función con dos parámetros de tipo llamados `T1` y `T2` comenzaría así:

```
template<class T1, class T2>
```

Sin embargo, la mayoría de las plantillas de función sólo requieren un parámetro de tipo. No podemos tener parámetros de plantilla que no se usen; es decir, la plantilla de función debe usar todos los parámetros de plantilla.

RIESGO *Complicaciones del compilador*

Muchos compiladores no permiten la compilación por separado de las plantillas, por lo que podría ser necesario incluir la definición de una plantilla junto con el código que la usa. Como siempre, al menos la declaración de función debe preceder a cualquier uso de la plantilla de función.

La estrategia más segura es no utilizar declaraciones de plantillas de función, y asegurarse de que la definición de plantilla de función aparece en el mismo archivo en que se utiliza y que aparece antes de la invocación a la plantilla de función. Sin embargo, la definición de plantilla de función puede aparecer por medio de la directiva `#include`. Podemos dar una definición de plantilla de función en un archivo y después incluir la directiva `#include` en el archivo que utiliza la plantilla de función.

Algunos compiladores C++ tienen otros requisitos especiales para usar plantillas. Si tiene problemas para compilar sus plantillas, revise sus manuales o consulte a un experto. Podría ser necesario especificar opciones especiales o reacomodar las definiciones de plantillas y otros componentes de sus archivos.

Ejercicios de AUTOEVALUACIÓN

1. Escriba una plantilla de función llamada `maximo`. La función recibe dos valores del mismo tipo como argumentos y devuelve el mayor de los dos argumentos (o cualquiera de los valores si son iguales). Dé tanto la declaración de función como la definición de función para la plantilla. Usará el operador `<` en su definición. Por tanto, esta plantilla de función sólo aplicará a tipos para los cuales el operador `<` esté definido. Escriba un comentario para la declaración de función que explique esta restricción.
2. Hemos usado tres clases de función de valor absoluto: `abs`, `labs` y `fabs`. Estas funciones sólo difieren en el tipo de su argumento. Podría ser mejor tener una plantilla para la función de valor absoluto. Dé una plantilla para una función de valor absoluto llamada `absoluto`. La plantilla aplicará sólo a los tipos para los que esté definido `<`, para los que esté definido el operador de negación unario, y para los que se pueda usar la constante `0` en una comparación con un valor de ese tipo. Así pues, la función `absoluto` se podrá

Plantilla de función

Tanto la definición como la declaración de función de una plantilla de función van precedidas de lo siguiente:

```
template<class Parametro_de_Tipo>
```

Por lo demás, la declaración (si se usa) y la definición son como la declaración de función y la definición de cualquier función ordinaria, excepto que se puede usar el *Parametro_de_Tipo* en lugar de un tipo.

Por ejemplo, lo que sigue es la declaración de una plantilla de función:

```
template<class T>
void mostrar_cosas(int cosas1, T cosas2, T cosas3);
```

La definición de esta plantilla de función podría ser:

```
template<class T>
void mostrar_cosas(int cosas1, T cosas2, T cosas3)
{
    cout << cosas1 << endl
         << cosas2 << endl
         << cosas3 << endl;
}
```

La plantilla de función dada en este ejemplo equivale a tener una declaración y una definición de función para todos los posibles nombres de tipo. El nombre de tipo es sustituido por el parámetro de tipo (que es `T` en el ejemplo anterior). Por ejemplo, consideremos la siguiente llamada de función:

```
mostrar_cosas(2, 3.3, 4.4);
```

Cuando se ejecuta esta llamada, el compilador usa la definición de función que se obtiene al sustituir `T` por el nombre de tipo `double`. Se producirá una definición aparte para cada tipo distinto con el cual usemos la plantilla, pero no para los tipos que no usemos. Sólo se generará una definición para un tipo específico, sin importar cuántas veces usemos la plantilla.

Abstracción de algoritmos

Como vimos al examinar la función `intercambiar_valores`, existe un algoritmo muy general para intercambiar el valor de dos variables, y este algoritmo más general aplica a variables de cualquier tipo. Al usar una plantilla de función pudimos expresar este algoritmo más general en C++. Éste es un ejemplo muy sencillo de *abstracción de algoritmos*. Cuando decimos que estamos usando **abstracción de algoritmos** queremos decir que estamos expresando nuestros algoritmos de una forma muy general para poder olvidarnos de los detalles incidentales y concentrarnos en la parte sustancial del algoritmo. Las plantillas de funciones son una característica de C++ que apoya la abstracción de algoritmos.

invocar con cualquiera de los tipos numéricos, como *int*, *long* y *double*. Dé tanto la declaración como la definición de función para la plantilla.

3. Defina o caracterice el recurso de plantillas de C++.

4. En el prefijo de plantilla

```
template<class T>
```

¿qué clase de variable es el parámetro T?

a) T debe ser una clase.

b) T no debe ser una clase.

c) T sólo puede representar tipos predefinidos en el lenguaje C++.

d) T puede ser cualquier tipo, sea predefinido o definido por el programador.

EJEMPLO DE PROGRAMACIÓN

Una función de ordenamiento genérica

En el capítulo 10 dimos un sencillo algoritmo para ordenar un arreglo de valores de tipo *int*. El algoritmo se codificó en C++ como la función ordenar que presentamos en el cuadro 10.12. A continuación repetimos las definiciones de esta función:

```
void ordenar(int a[], int elem_usados)
{
  int indice_del_siguiente_menor;
  for (int indice = 0; indice < elem_usados - 1; indice++)
  {//Colocar el valor correcto en a[indice]:
     indice_del_siguiente_menor =
            indice_del_menor(a, indice, elem_usados);
     intercambiar_valores(a[indice], a[indice_del_siguiente_menor]);
     //a[0] <= a[1] <= ... <= a[indice] son los más pequeños
     //de los elementos originales del arreglo. Los demás
     //elementos están en las posiciones restantes.
 }
}
```

funciones de ayuda

Si estudiamos la definición anterior de la función ordenar veremos que el tipo base del arreglo nunca se usa de forma importante. Si sustituimos el tipo base del arreglo en el encabezado de la función por el tipo double, obtendremos una función de ordenamiento que aplica a arreglos de valores de tipo *double*. Desde luego, también deberemos ajustar las funciones auxiliares de modo que apliquen a arreglos de elementos de tipo *double*. Consideremos entonces las funciones auxiliares que se invocan dentro del cuerpo de la función ordenar. Las dos funciones auxiliares son intercambiar_valores e indice_del_menor.

Ya vimos que intercambiar_valores puede aplicar a variables de cualquier tipo, siempre que la definamos como plantilla de función (como en el cuadro 14.1). Veamos si indice_del_menor depende de alguna forma significativa del tipo base del arreglo que se está ordenando. Repetiremos aquí la definición de indice_del_menor para poder estudiar sus detalles.

```
int indice_del_menor(const int a[], int indice_inicio,
                     int elem_usados)
{
    int min = a[indice_inicio];
    int indice_de_min = indice_inicio;
    for (int indice = indice_inicio + 1;
                     indice < elem_usados; indice++)
        if (a[indice] < min)
        {
            min = a[indice];
            indice_de_min = indice;
            //min es el menor de a[indice_inicio] a
            //a[indice]
        }
    return indice_de_min;
}
```

La función `indice_del_menor` tampoco depende de alguna forma significativa del tipo base del arreglo. Si sustituyéramos las dos ocurrencias resaltadas del tipo `int` por el tipo `double`, habríamos transformado la función `indice_del_menor` de modo que aplique a arreglos cuyo tipo base es `double`.

Para modificar la función `ordenar` de modo que pueda servir para ordenar arreglos cuyo tipo base es `double`, lo único que necesitamos es reemplazar unas cuantas ocurrencias del nombre de tipo `int` por el nombre de tipo `double`. Es más, el tipo `double` nada tiene de especial, así que podríamos hacer una sustitución similar por muchos otros tipos. Lo único que necesitamos saber acerca del tipo es que el operador < está definido para ese tipo. Ésta es la situación perfecta para usar plantillas de funciones. Si sustituimos unas cuantas ocurrencias del nombre de tipo `int` (en las funciones `ordenar` e `indice_del_menor`) por un parámetro de tipo, la función `ordenar` podrá ordenar un arreglo de valores de cualquier tipo, siempre que los valores de ese tipo se puedan comparar usando el operador <. En el cuadro 14.2 hemos escrito precisamente esa plantilla de función.

Cabe señalar que la plantilla de función `ordenar` que se muestra en el cuadro 14.2 puede usarse con arreglos de valores que no sean números. En el programa de demostración, que se muestra en el cuadro 14.3, se invoca la plantilla de función `ordenar` para ordenar un arreglo de caracteres. Los caracteres pueden compararse con el operador <. Aunque el significado exacto del operador < aplicado a valores de carácter podría variar un poco de una implementación a otra, siempre se cumplen algunas cosas acerca de la forma en que < aplica a las letras del alfabeto. Cuando se aplica a dos letras mayúsculas, el operador < prueba si la primera está antes de la segunda en el orden alfabético. También, si se aplica a dos letras minúsculas, el operador < prueba si la primera está antes de la segunda en el orden alfabético. Cuando mezclamos letras mayúsculas y minúsculas la situación no es tan predecible, pero el programa del cuadro 14.3 sólo maneja letras mayúsculas. En ese programa se coloca en orden alfabético un arreglo de letras mayúsculas con una llamada a la plantilla de función `ordenar`. (La plantilla `ordenar` incluso ordena un arreglo de objetos de una clase definida por nosotros, siempre que sobrecarguemos el operador < de modo que aplique a objetos de la clase.)

CUADRO 14.2 *Una función de ordenamiento genérica*

```
//Éste es el archivo ordenamientofunc.cpp

template<class T>
void intercambiar_valores(T& variable1, T& variable2)
                    <El resto de la definición para intercambiar_valores se encuentra en el cuadro 14.1.>

template<class BaseType>
int indice_del_menor(const BaseType a[], int indice_inicio, int elem_usados)
{
    BaseType min = a[indice_inicio];
    int indice_de_min = indice_inicio;

    for(int indice = indice_inicio + 1; indice < elem_usados; indice ++)
       if(a[indice] < min)
       {
          min = a[indice];
          indice_de_min = indice;
          // min es el más pequeño de a[indice_inicio] a[indice]
       }
    return indice_de_min;
}

template<class BaseType>
void ordenar(BaseType a[], int elem_usados)
{
int indice_del_siguiente_menor;
for(int indice = 0; indice < elem_usados - 1; indice++)
    {//Coloca correctamente los valores en a[indice]:
          indice_del_siguiente_menor =
                  indice_del_menor(a, indice, elem_usados);
          intercambiar_valores(a[indice], a[indice_del_siguiente_menor]);
          //a[0] <= a[1] <= ... <= a[indice] son los elementos más pequeños
          //del arreglo. El resto de los elementos permanecen en sus posiciones.
    }
}
```

CUADRO 14.3 *Uso de una función de ordenamiento genérica (parte 1 de 2)*

```
//Demuestra una función de ordenamiento genérica
#include<iostream>
using namespace std;

//El archivo ordenamientofunc.cpp define la siguiente función:
//template<class BaseType>
//void ordenar(BaseType a[], int elem_usados);
//Precondición: elem_usados <= tamaño declarado del arreglo a.
//Los elementos del arreglo a[0] hasta a[elem_usados-1] tienen valores
//Postcondición: Los valores a[0] hasta a[elem_usados-1] se han
//reacomodado de modo que a[0] <= a[1] <= ... <= a[elem_usados-1].

#include "ordenamientofunc.cpp"

int main()
{
    int i;
    int a[10] = {9, 8, 7, 6, 5, 1, 2, 3, 0, 4};
    cout << "Enteros no ordenados:\n";
    for (i = 0; i < 10; i++)
       cout  << a[i] << " ";
    cout << endl;
    ordenar(a, 10);
    cout << " De menor a mayor los enteros son:\n";
    for (i=0; i<10; i++)
       cout << a[i] <<" ";
    cout << endl;

    double b[5] = {5.5, 4.4, 1.1, 3.3, 2.2};
    cout << " Dobles no ordenados:\n";
    for (i=0; i<5; i++)
       cout << b[i] << " ";
    cout << endl;
    ordenar(b,5);
    cout << " De menor a mayor los dobles son:\n";
    for (i=0; i<5; i++)
       cout << b[i] << " ";
    cout <<endl;

    char c[7] = {'G', 'E', 'N', 'E', 'R', 'I', 'C'};
    cout << "Caracteres no ordenados: \n";
    for(i = 0; i<7; i++)
       cout << c[i] << " ";
    cout << endl;
```

Muchos compiladores permitirán que esta declaración de función aparezca como tal y no solamente como comentario. Sin embargo incluir la declaración de función no es necesario desde que la definición de la función está en el archivo `ordenamientofunc.cpp` *y la definición efectivamente aparece antes de la parte* `main`.

CUADRO 14.3 *Uso de una función de ordenamiento genérica (parte 2 de 2)*

```
    ordenar(c,7);
    cout << "De menor a mayor los caracteres son:\n";
    for (i=0; i <7; i++)
         cout << c[i] << " ";
    cout << endl;

    return 0;
}
```

Salida

```
Enteros no ordenados:
9 8 7 6 5 1 2 3 0 4
De menor a mayor los enteros son:
0 1 2 3 4 5 6 7 8 9
Dobles no ordenados:
5.5 4.4 1.1 3.3 2.2
De menor a mayor los dobles son:
1.1 2.2 3.3 4.4 5.5
Caracteres no ordenados:
G E N E R I C
De menor a mayor los caracteres son:
C E E G I N R
```

TIP DE PROGRAMACIÓN

Cómo definir plantillas

Cuando definimos las plantillas de función en el cuadro 14.2, partimos de una función que ordena un arreglo de elementos de tipo `int`. Luego creamos una plantilla sustituyendo el tipo base del arreglo por el parámetro de tipo `T`. Ésta es una buena estrategia general para escribir plantillas. Si queremos escribir una plantilla de función, primero escribimos una versión que sea una función ordinaria, no una plantilla. Luego depuramos perfectamente la función ordinaria, y por último la convertimos en una plantilla sustituyendo algunos nombres de tipo por un parámetro de tipo. Este método tiene dos ventajas. Primera, cuando estamos definiendo la función ordinaria estamos manejando un caso mucho más concreto, lo que facilita la visualización del problema. Segunda, hay menos detalles que verificar en cada etapa; cuando nos estamos concentrando en el algoritmo, no hay necesidad de preocuparnos por las reglas de sintaxis de las plantillas.

RIESGO *Uso de una plantilla con el tipo inapropiado*[2]

Podemos utilizar una plantilla de función con cualquier tipo con el cual el código en una definición de función tenga sentido. Sin embargo, todo el código en una plantilla de función debe tener sentido y funcionar de manera apropiada. Por ejemplo, no puede utilizar la plantilla `intercambiar_valores` (cuadro 14.1) con cualquier parámetro de tipo reemplazado por un tipo con el cual el operador de asignación no trabaje por completo o no trabaje correctamente.

Un ejemplo más concreto sería que su programa defina una plantilla de función `intercambiar_valores` como en el cuadro 14.1. No puede agregar lo siguiente a su programa.

```
int a[10], b[10];
<algo de código para llenar el arreglo>
intercambiar_valores (a, b);
```

Este código no funcionará debido a que la asignación no funciona con los tipos de arreglos.

Ejercicios de AUTOEVALUACIÓN

5. El cuadro 10.10 muestra una función llamada `buscar`, que busca un entero dado en un arreglo. Escriba una versión de plantilla de la función `buscar` que pueda servir para buscar en un arreglo de elementos de cualquier tipo. Dé tanto la declaración de función como la definición de función para la plantilla. *Sugerencia:* Es casi idéntica a la función del cuadro 10.10.
6. En el proyecto de programación 9 del capítulo 3 se pidió sobrecargar la función `abs` de modo que el nombre `abs` funcionara con varios de los tipos predefinidos que se habían estudiado hasta entonces. Compare y contraste la sobrecarga del nombre de función `abs` con el uso de plantillas para este fin en el ejercicio de autoevaluación 2.

14.2 Plantillas para abstracción de datos

Igualdad en riqueza e igualdad en oportunidades de cultura...
nos han hecho a todos simplemente miembros de una sola clase.

Edward Bellamy, *Looking Backward, 2000-1887*

Como vimos en la sección anterior, podemos hacer más generales las definiciones de funciones usando plantillas. En esta sección veremos que las plantillas también pueden hacer más generales las definiciones de clases.

[2]El ejemplo en esta sección de Riesgo utiliza arreglos. Si no ha leído al capítulo de arreglos (capítulo 10). Debería saltarse esta sección de Riesgo y leerla nuevamente una vez que haya leído el capítulo 10.

Sintaxis de plantillas de clases

La sintaxis de las plantillas de clase es básicamente la misma que la de las plantillas de funciones. Lo que sigue se coloca antes de la definición de la plantilla:

```
template<class T>
```

parámetro de tipo

El parámetro de tipo `T` se usa en la definición de clase exactamente como cualquier otro tipo. Igual que en las plantillas de funciones, el parámetro de tipo `T` representa un tipo que puede ser cualquiera; el parámetro de tipo no se tiene que sustituir por un tipo de clase. Al igual que con las plantillas de función, podemos usar cualquier identificador (que no sea palabra clave) en lugar de `T`.

Por ejemplo, la que sigue es una plantilla de clase. Un objeto de esta clase contiene un par de valores de tipo `T`: si `T` es `int`, los valores de los objetos son pares de enteros; si `T` es `char`, los valores de los objetos son pares de caracteres, etcétera.

```
//Clase para un par de valores de tipo T:
template<class T>
class Par
{
public;
    Par();
    Par(T primer_valor, T segundo_valor);
    void fijar_elemento(int posicion, T valor);
    //Precondición: posicion es 1 o 2.
    //Postcondición: Se ha asignado valor a la posición indicada.

    T obtener_elemento(int posicion) const;
    //Precondición: posicion es 1 o 2.
    //Devuelve el valor que está en la posición indicada.
private:
    T primero;
    T segundo;
};
```

declaración de objetos

Una vez definida la plantilla de clase, podemos declarar objetos de esta clase. La declaración debe especificar qué tipo sustituirá a `T`. Por ejemplo, lo que sigue declara el objeto `marcador` de manera que pueda registrar un par de enteros y declara el objeto `asientos` de modo que pueda registrar un par de caracteres:

```
Par<int> marcador;
Par<char> asientos;
```

Después, los objetos se usan como cualquier otro objeto. Por ejemplo, lo que sigue hace que el `marcador` sea `3` para el primer equipo y `0` para el segundo equipo:

```
marcador.fijar_elemento(1, 3);
marcador.fijar_elemento(2, 0);
```

definición de funciones miembro

Las funciones miembro de una plantilla de clase se definen de la misma manera que las de clases ordinarias. La única diferencia es que las definiciones de funciones miembro

también son plantillas. Por ejemplo, las que siguen son definiciones apropiadas para la función miembro `fijar_elemento` y para el constructor que recibe dos argumentos:

```
//Usa iostream y cstdlib:
template<class T>
void Par<T>::fijar_elemento(int posicion, T valor)
{
     if (posicion == 1)
         primero = valor;
     else if (posicion == 2)
         segundo = valor;
     else
     {
         cout << "Error: posición de par no válida.\n";
         exit(1);
     }
}

template<class T>
Par<T>::Par(T primer_valor, T segundo_valor)
        : primero(primer_valor), segundo(segundo_valor)
{
    //Cuerpo intencionalmente vacío.
}
```

Sintaxis de plantillas de clase

La definición de clase y las definiciones de las funciones miembro van precedidas de lo siguiente:

```
template<class Parametro_de_Tipo>
```

Por lo demás, las definiciones de clase y de funciones miembro son iguales que las de cualquier clase ordinaria, excepto que se puede usar el *Parametro_de_Tipo* en lugar de un tipo.

Por ejemplo, lo que sigue es el principio de una definición de plantilla de clase:

```
template<class T>
class Par
{
public:
    Par();
    Par(T primer_valor, T segundo_valor);
void fijar_elemento(int posicion, T valor);
       ...
```

Las funciones miembro y operadores sobrecargados se definen entonces como plantillas de funciones. Por ejemplo, la definición del constructor de dos argumentos para la plantilla de clase anterior iniciaría así:

```
template<class T>
void Par<T>::fijar_elemento(int posicion, T valor)
{
     ...
```

Observe que el nombre de clase que está antes del operador de resolución de alcance es `Par<T>`, no simplemente `Par`.

Podemos usar el nombre de una plantilla de clase como tipo de un parámetro de función. Por ejemplo, lo que sigue es una posible declaración de una función que tiene un parámetro que es un par de enteros:

```
int sumar(const Par<int>& el_par);
//Devuelve la suma de los dos enteros de el_par.
```

Observe que especificamos el tipo, en este caso `int`, que debe sustituir al parámetro de tipo `T`.

Incluso podemos usar una plantilla de clase dentro de una plantilla de función. Por ejemplo, en lugar de definir la función especializada `sumar` que dimos antes, podríamos definir la siguiente plantilla de función de modo que la función aplique a todo tipo de números:

```
template<class T>
T sumar(const Par<T>& el_par);
//Precondición: El operador + está definido para valores de tipo T.
//Devuelve la suma de los dos enteros de el_par.
```

Definiciones de tipos

Podemos especializar una plantilla de clase dando un argumento de tipo con el nombre de la clase, como en el siguiente ejemplo:

```
Par<int>
```

El nombre de clase especializado, como `Par<int>`, se puede usar entonces igual que cualquier nombre de clase. Podemos usarlo para declarar objetos o para especificar el tipo de un parámetro formal.

Podemos definir un nuevo nombre de tipo de clase que tenga el mismo significado que un nombre de plantilla de clase especializado, como `Par<int>`. La sintaxis de semejante nombre de tipo de clase definido es la siguiente:

```
typedef Nombre_de_Clase<Argumento_de_Tipo> Nuevo_Nombre_de_Tipo;
```

Por ejemplo:

```
typedef Par<int> ParDeInt;
```

El nombre de tipo `ParDeInt` se puede usar entonces para declarar objetos de tipo `Par<int>`, como en el siguiente ejemplo:

```
ParDeInt par1, par2;
```

El nombre de tipo `ParDeInt` también puede servir para especificar el tipo de un parámetro formal.

EJEMPLO DE PROGRAMACIÓN
Una clase de arreglo

El cuadro 14.4 contiene la interfaz de una plantilla de clase cuyos objetos son listas. Puesto que esta definición de clase es una plantilla de clase, las listas pueden ser listas de elementos de cualquier tipo. Podemos tener objetos que sean listas de valores de tipo `int`, o listas de valores de tipo `double`, o listas de objetos del tipo `cstring`, o listas de elementos de cualquier otro tipo.

El cuadro 14.5 contiene un programa de demostración que usa esta plantilla de clase. Aunque este programa no hace mucho que digamos, sí ilustra el uso de la plantilla de clase. Una vez que el lector entienda los detalles de la sintaxis, podrá usar la plantilla de clase en cualquier programa que necesite una lista de valores. La implementación de la plantilla de clase se da en el cuadro 14.6.

Observe que sobrecargamos el operador de inserción << a fin de poder usarlo para enviar a la salida un objeto de la plantilla de clase `ListaGenerica`. Para ello hicimos al operador << una función friend de la clase. Para tener un parámetro que sea del mismo tipo que la clase, usamos la expresión `ListaGenerica<ItemType>` para el tipo de parámetro. Cuando el tipo de parámetro sea sustituido por, digamos, el tipo `int`, este parámetro de lista será de tipo `ListaGenerica<int>`.

función friend

CUADRO 14.4 *Interfaz para la plantilla de clase* `ListaGenerica` *(parte 1 de 2)*

```
//Éste es el ARCHIVO DE ENCABEZADO listagenerica.h. Es la interfaz de la
//clase ListaGenerica. Los objetos de tipo ListaGenerica pueden ser una lista de
//cosas de cualquier tipo para el que estén definidos los operadores << y =.
//Todos los elementos de una lista deben ser del mismo tipo. Una lista que
//puede contener hasta máximo de elementos de tipo Nombre_Tipo se declara así:
//     ListaGenerica<Nombre_Tipo> el_objeto(max);
#ifndef LISTAGENERICA_H
#define LISTAGENERICA_H
#include <iostream>
using namespace std;

namespace listsavitch
{
    template<class ItemType>
    class ListaGenerica
    {
    public:
        ListaGenerica(int max);
        //Inicializa el objeto con una lista vacía que puede contener
        //hasta un máximo de cosas de tipo ItemType.
```

CUADRO 14.4 *Interfaz para la plantilla de clase* `ListaGenerica` *(parte 2 de 2)*

```
        ~ListaGenerica();
        //Devuelve al montículo toda la memoria dinámica usada por el objeto.

        int longitud() const;
        //Devuelve el número de elementos que hay en la lista.

        void agregar(ItemType elem_nuevo);
        //Precondición: La lista no está llena.
        //Postcondición: Se agregó elem_nuevo a la lista.

        bool llena() const;
        //Devuelve true si la lista está llena.

        void borrar();
        //Elimina todos los elementos de la lista, la cual queda vacía.

        friend ostream& operator <<(ostream& sale,
                                    const ListaGenerica<ItemType>& la_lista);
        //Sobrecarga el operador << de modo que sirva para exhibir el
        //contenido de la lista. Se exhibe un elemento por línea.
        //Precondición: Si sale es un flujo de archivo de salida, ya
        //se conectó a un archivo.
    private:
        ItemType *elemento; //puntero al arreglo dinámico que contiene la lista.
        int long_max; //número máximo de elementos que caben en la lista.
        int long_actual; //número de elementos que hay ahora en la lista.
    };
}//listsavitch
#endif  // LISTAGENERICA_H
```

CUADRO 14.5 *Programa que usa la plantilla de clase* ListaGenerica

```
//Programa para demostrar el uso de la plantilla de clase ListaGenerica.
#include <iostream>
#include "ListaGenerica.h"
#include "ListaGenerica.cpp"
using namespace std;
using namespace listsavitch;

int main()
{
    ListaGenerica<int> primera_lista(2);
    primera_lista.agregar(1);
    primera_lista.agregar(2);
    cout << "primera_lista = \n"
         << primera_lista;

    ListaGenerica<char> segunda_lista(10);
    segunda_lista.agregar('A');
    segunda_lista.agregar('B');
    segunda_lista.agregar('C');
    cout << "segunda_lista = \n"
         << segunda_lista;

    return 0;
}
```

Desde que se incluye el archivo `listagenerica.cpp` *no necesita compilar este archivo que contiene la parte* `main`.

Salida

```
primera_lista =
1
2
segunda_lista =
A
B
C
```

CUADRO 14.6 *Implementación de* ListaGenerica *(parte 1 de 3)*

```
//Éste es el ARCHIVO DE IMPLEMENTACIÓN: listagenerica.cpp
//Ésta es la IMPLEMENTACIÓN de la plantilla de clase llamada ListaGenerica.
//La interfaz para ListaGenerica está en el archivo de encabezado listagenerica.h.
#ifndef LISTAGENERICA_CPP
#define LISTAGENERICA_CPP
#include <iostream>
#include <cstdlib>
#include "listagenerica.h"//Esto no es necesario cuando este archivo se utiliza
                          //como lo estamos haciendo ahora, pero la directiva
                          //#ifndef en listagenerica.h lo hace más seguro.
using namespace std;

namespace listsavitch
{
    //Usa cstdlib:
    template<class ItemType>
    ListaGenerica<ItemType>:: ListaGenerica(int max):long_max = (max),
                                                     long_actual = 0

    {
            elemento = new ItemType[max];
    }

    template<class ItemType>
    ListaGenerica<ItemType>::~ListaGenerica()
    {
          delete [] elemento;
    }
```

CUADRO 14.6 *Implementación de* ListaGenerica *(parte 2 de 3)*

```
template<class ItemType>
int ListaGenerica<ItemType>::longitud() const
{
    return (long_actual);
}
//Usa iostream y cstdlib:
template<class ItemType>
void ListaGenerica<ItemType>::agregar(ItemType elem_nuevo)
{
    if ( llena() )
    {
        cout << "Error: La lista ya esta llena.\n";
        exit(1);
    }
    else
    {
        elemento[long_actual] = elem_nuevo;
        long_actual = long_actual + 1;
    }
}

template<class ItemType>
bool ListaGenerica<ItemType>::llena() const
{
    return (long_actual == long_max);
}

template<class ItemType>
void ListaGenerica<ItemType>::borrar()
{
    long_actual = 0;
}
```

CUADRO 14.6 *Implementación de* `ListaGenerica` *(parte 3 de 3)*

```
    //Usa iostream:
    template<class ItemType>
    ostream& operator <<(ostream& sale, const ListaGenerica<ItemType>& la_lista)
    {
          for (int i = 0; i < la_lista.long_actual; i++)
               sale << la_lista.elemento[i] << endl;

          return sale;
    }
}//listsavitch
#endif  //LISTAGENERICA_CPP   Observe que encerramos todas las
        //definiciones de plantillas entre #ifndef y #endif.
```

Debemos incluir una nota acerca de la compilación del código de los cuadros 14.4, 14.5 y 14.6. Una solución segura es poner con `#include` las declaraciones de la clase plantilla y las definiciones de las funciones plantilla antes de usarlos, como hicimos aquí. En ese caso sólo es necesario compilar el archivo del cuadro 14.5. Asegúrese de usar el mecanismo `#ifndef #define #endif` para evitar que se incluyan varias veces los archivos indicados con `#include`.

Ejercicios de AUTOEVALUACIÓN

7. Dé la definición de la función miembro `obtener_elemento` para la plantilla de clase `Par` que vimos en la sección "Sintaxis de plantillas de clase".

8. Dé la definición del constructor con cero argumentos para la plantilla de clase `Par` que vimos en la sección "Sintaxis de plantillas de clase".

9. Dé la definición de una plantilla de clase llamada `ParHeterogeneo` que es como la plantilla de clase `Par` que se discutió en la sección "Sintaxis de plantillas de clase", con la excepción de que con `ParHeterogeneo`, la primera y segunda posición probablemente almacenarán valores de diferentes tipos. Utilice dos parámetros `T1` y `T2`; todos los elementos en la primera posición serán del tipo `T1` y todos los elementos en la segunda posición serán del tipo `T2`. El método `fijar_elemento` en la plantilla de clases `Par` debe ser reemplazado por dos métodos llamados `fijar_primero` y `fijar_segundo` en la plantilla de clases `ParHeterogeneo`. De igual manera, el método de acceso `obtener_elemento` en la platilla de clases `Par` debe ser reemplazado por dos métodos de acceso llamados `obtener_primero` y `obtener_segundo` en la plantilla de clases `ParHeterogeneo`.

10. Comente brevemente la siguiente afirmación, indicando si es verdad o no, y justifique su opinión:

    Las funciones friend se usan exactamente igual con clases plantilla y no plantilla.

Resumen del capítulo

- Si usamos plantillas de funciones podemos definir funciones que tienen un parámetro para un tipo.
- Si usamos plantillas de clases podemos definir una clase con un parámetro de tipo para subpartes de la clase.

Respuestas a los ejercicios de autoevaluación

1. Declaración:

```
template<class T>
T maximo(T primero, T segundo);
//Precondición: El operador < está definido para el tipo T.
//Devuelve el mayor de primero y segundo.
```

Definición:

```
template<class T>
T maximo(T primero, T segundo)
{
    if (primero < segundo)
        return segundo;
    else
       return primero;
}
```

2. Declaración de función:

```
template<class T>
T absoluto(T valor);
//Precondición: Las expresiones x < 0 y -x están definidas
//siempre que x es de tipo T.
//Devuelve el valor absoluto de su argumento.
```

Definición:

```
template<class T>
T absoluto(T valor)
{
    if (valor < 0)
        return -valor;
    else
       return valor;
}
```

3. Las plantillas son un recurso que permite definir funciones y clases que tienen parámetros para los nombres de tipo.
4. d) Cualquier tipo, sea primitivo (provisto por C++) o definido por el usuario (un tipo `class` o `struct`, un tipo `enum` o un arreglo definido como tipo, `int`, `float`, `double`, etcétera).

5. A continuación se dan la declaración y la definición de la función. Básicamente son idénticos a los dados en el cuadro 10.10, excepto que dos ocurrencias de *int* se cambian a BaseType en la lista de parámetros.

Declaración de Función:

```
template<class BaseType>
int buscar(const BaseType                                                    a[],
                      int elem_usados, BaseType buscado);
//Precondición: elem_usados es <= el tamaño declarado de a.
//También a[0] a a[elem_usados−1] tienen valores
//Devuelve el primer índice tal que a[indice] == buscado,
//siempre que exista ese índice; de lo contrario, devuelve −1.
```

Definición:

```
template<class BaseType>
int buscar(const BaseType a[], int elem_usados,
                                      BaseType buscado)
{
    int indice = 0, hallado = false;
    while ((!hallado) && (indice < elem_usados))
    if (buscado == a[indice])
        hallado = true;
    else
        indice++;
    if (hallado)
        return indice;
    else
        return −1;
}
```

6. La sobrecarga de funciones sólo funciona con tipos para los que se provee una sobrecarga. La sobrecarga podría funcionar con tipos que se convierten automáticamente a algún tipo para el que se proveyó una sobrecarga, pero podría no hacer lo que se espera. La solución de plantilla funciona con cualquier tipo que esté definido en el momento de la investigación, siempre que se cumplan los requisitos para una definición de <.

7.
```
//Usa iostream y cstdlib:
template<class T>
T Par<T>::obtener_elemento(int posicion) const
{
    if (posicion == 1)
        return primero;
    else if (posicion == 2)
        return segundo;
    else
    {
       cout << "Error: Posicion de par no valida.\n";
       exit(1);
    }
}
```

8. No hay candidatos naturales para los valores de inicialización predeterminada así que este constructor no hace nada, aunque sí nos permite declarar objetos (no inicializados) sin dar argumentos para el constructor.

```
template<class T>
Par<T>::Par()
{
//No hacer nada.
}
```

9.
```
//Clase para un par de valores, el primero de tipo T1
//y el segundo de tipo T2:
template<class T1, class T2>
class ParHeterogeneo
{
public;
    ParHeterogeneo();
    ParHeterogeneo(T1 primer_valor, T2 segundo_valor);
    void fijar_primero(T1 valor);
    void fijar_segundo(T2 valor);
    T1 obtener_primero( ) const;
    T2 obtener_segundo( ) const;
private:
    T1 primero;
    T2 segundo;
};
```

La definición de función miembro es como sigue:

```
template<class T1, class T2>
ParHeterogeneo<T1, T2>::ParHeterogeneo()
{
//No hace nada.
}

template<class T1, class T2>
ParHeterogeneo<T1, T2>::ParHeterogeneo
    (T1 primer_valor, T2 segundo_valor)
          : primero(primer_valor), segundo(segundo_valor);
{
     //Cuerpo intencionalmente vacío.
}

template<class T1, class T2>
T1 ParHeterogeneo<T1, T2>::obtener_primero() const
{
    return primero;
}
```

```
template<class T1, class T2>
T2 ParHeterogeneo<T1, T2>::obtener_segundo() const
{
     return segundo;
}

template<class T1, class T2>
void ParHeterogeneo<T1, T2>::establecer_primero(T1 value)
{
    primero = value;
}

template<class T1, class T2>
void ParHeterogeneo<T1, T2>::establecer_segundo(T2 value)
{
    segundo = value;
}
```

10. `True`

Proyectos de programación

1. Escriba una plantilla para una función que tiene parámetros para un arreglo parcialmente lleno y para un valor del tipo base del arreglo. Si el valor está en el arreglo, la función devuelve el índice de la primera variable indizada que contiene el valor. Si el valor no está en el arreglo, la función devuelve `-1`. El tipo base del arreglo es un parámetro de tipo. Recuerde que se necesitan dos parámetros para dar el arreglo parcialmente lleno: uno para el arreglo y otro para el número de variables indizadas que se usan. Escriba también un programa de prueba apropiado para esta plantilla de función.

2. 

Reescriba la definición de la plantilla de clase `ListaGenerica` que se da en los cuadros 14.4 y 14.6 de modo que sea más general. Esta versión más general tiene la característica adicional de que podemos procesar los elementos de la lista en orden. Un elemento siempre es el elemento actual. Se puede pedir el elemento actual, se puede hacer que el elemento actual sea el siguiente elemento, se puede hacer que el elemento actual sea el elemento anterior, se puede comenzar al principio de la lista haciendo que el primer elemento sea el elemento actual, y se puede pedir el *n*-ésimo elemento de la lista. Para ello es preciso añadir los siguientes miembros: una variable miembro adicional que registre la posición del elemento actual en la lista, una función miembro que devuelva el elemento actual como valor, una función miembro que haga al siguiente elemento el elemento actual, una función miembro que haga al elemento anterior el elemento actual, una función miembro que haga al primer elemento de la lista el elemento actual, y una función miembro que devuelva el *n*-ésimo elemento de la lista si se le da `n` como argumento. (Numere los elementos igual que en los arreglos de modo que el primer elemento sea el 0, el siguiente sea el elemento 1, etcétera.)

   Cabe señalar que hay situaciones en las que algunas de las acciones de estas funciones no son posibles. Por ejemplo, una lista vacía no tiene primer elemento, y no hay ningún elemento después del último elemento de cualquier lista. Asegúrese de determinar si la lista está vacía y de manejarla correctamente. No olvide probar si el elemento actual es el primero o el último de la lista y manejar debidamente esos casos. Escriba un programa adecuado para probar esta plantilla de clase.

3. Escriba una plantilla para una función que tiene parámetros para una lista de elementos y para un posible elemento de la lista. Si el elemento está en la lista, la función devuelve la posición de la primera ocurrencia de ese elemento. Si el elemento no está en la lista, la función devuelve `-1`. La primera posición de la lista es la 0, la siguiente es la posición `1`, etcétera. El tipo de los elementos de la lista es un parámetro de tipo. Use la plantilla de clase `ListaGenerica` que definió en el Proyecto 2. Escriba un programa apropiado para probar esta plantilla de función.

4. Realice nuevamente el proyecto de programación 3 del capítulo 10, pero en esta ocasión haga que la función `eliminar_repetidos` sea una plantilla de función con un tipo de parámetro para el tipo base del arreglo. Sería útil hacer primero la versión que no sea plantilla, es decir, elabore primero el proyecto de programación 3 del capítulo 10, en caso de que no lo haya elaborado ya.

5. El cuadro 14.3 proporciona una plantilla de función para el ordenamiento de un arreglo por medio del algoritmo de ordenamiento. Escriba una plantilla de función similar para el ordenamiento de un arreglo, pero en esta ocasión utilice el algoritmo de inserción descrito en el proyecto de programación 6 del capítulo 10. Si no ha elaborado este proyecto, sería buena idea que primero elaborara la versión sin plantilla, es decir, primero lleve a cabo el proyecto de programación 6 del capítulo 10.

6. Escriba una versión de plantilla de la búsqueda binaria iterativa del cuadro 13.8. Especifique los requisitos para el tipo parámetro de la plantilla. Comente los requisitos que debe satisfacer dicho tipo.

7. Escriba una versión de plantilla de la búsqueda binaria recursiva del cuadro 13.6. Especifique los requisitos para el tipo parámetro de la plantilla. Comente los requisitos que debe satisfacer dicho tipo.

8. (Este proyecto requiere que sepa qué es una pila y cómo utilizar arreglos dinámicos. El tema de pilas se cubre en el capítulo 13 y los arreglos dinámicos en el capítulo 12. Éste es un proyecto apropiado sólo si ha leído ambos capítulos.)

   Escriba la versión de una plantilla de la clase pila. Utilice un parámetro de tipo para el tipo de datos que se almacenará en la pila. Utilice un arreglo dinámico para permitir que la pila crezca y pueda almacenar cualquier número de elementos.

15 Apuntadores y listas enlazadas

15 Apuntadores y listas enlazadas

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

GILBERT Y SULLIVAN, *Ruddigore*

Introducción

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

Prerrequisitos

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

15.1 Nodos y listas enlazadas

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

Nodos

estructuras de nodos

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

definición de tipo de nodo

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

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

typedef  NodoLista*  NodoListaPtr;
```

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

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

```
NodoListaPtr  cabeza;
```

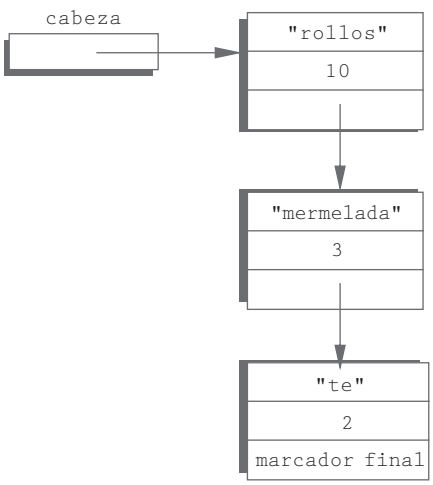
Aunque hemos ordenado las definiciones de tipos para evitar ciertas formas ilegales de circularidad, es evidente que la definición del `NodoLista` tipo *struct* sigue siendo circular. La definición del tipo `NodoLista` utiliza el nombre de tipo `NodoLista` para definir la variable miembro `enlace`. No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

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

modificación de los datos de un nodo

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

CUADRO 15.1 *Nodos y apuntadores*



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

el operador −>

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

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

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

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

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

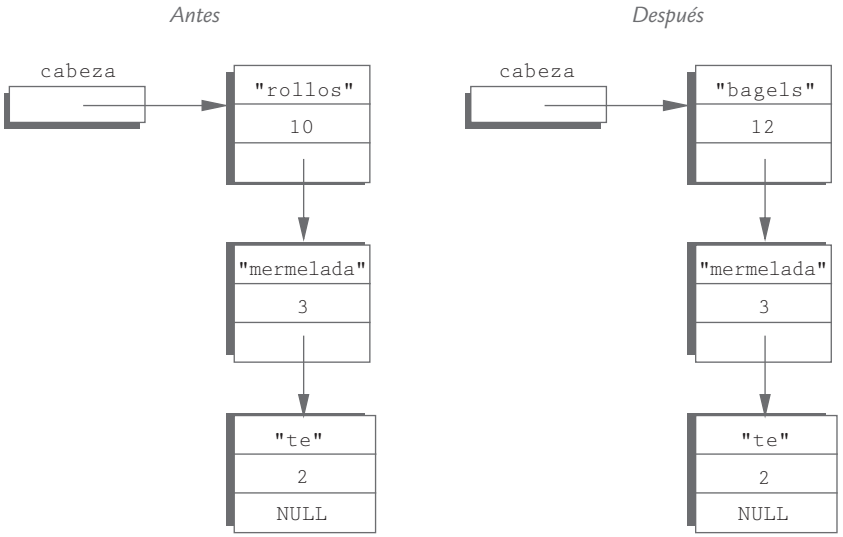
El resultado de estas modificaciones al primer nodo en la lista se muestra como diagrama en el cuadro 15.2

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

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

CUADRO 15.2 *Acceso a los datos de un nodo*

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



El operador flecha ->

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

Variable_apuntador->Nombre_miembro

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

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

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

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

NULL es 0

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

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

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

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

NULL

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

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

Ejercicios de AUTOEVALUACIÓN

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

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

typedef Caja* CajaPtr;
```

¿Cuál es la salida que produce el siguiente código?

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

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

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

4. Dada la siguiente definición de una estructura:

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

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

Listas enlazadas

lista enlazada

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

cabeza

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

definición de tipo de nodo

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

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

typedef Nodo* NodoPtr;
```

una lista enlazada de un nodo

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

```
NodoPtr cabeza;
```

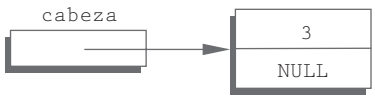
Para crear nuestro primer nodo utilizamos el operador *new* para crear una nueva variable dinámica que se convertirá en el primer nodo de nuestra lista enlazada.

```
cabeza = new Nodo;
```

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

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

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



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

Inserción de un nodo en la cabeza de una lista

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

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

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

Listas enlazadas como argumentos

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

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

algoritmo

Pseudocódigo para la función `insercion_cabeza`

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

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

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

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

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

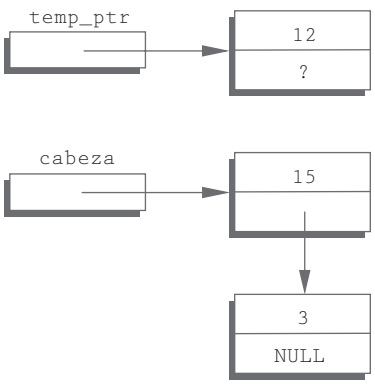
lista vacía

```
cabeza = NULL;
```

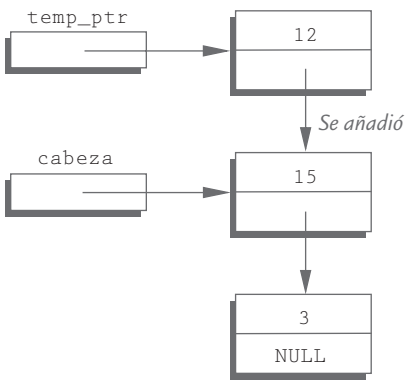
Cada vez que diseñe una función para manipular una lista enlazada, siempre deberá comprobar si funciona con la lista vacía. Si no funciona, tal vez pueda agregar un caso

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

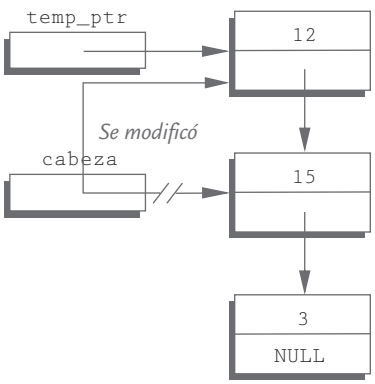
1. Se configura el nuevo nodo



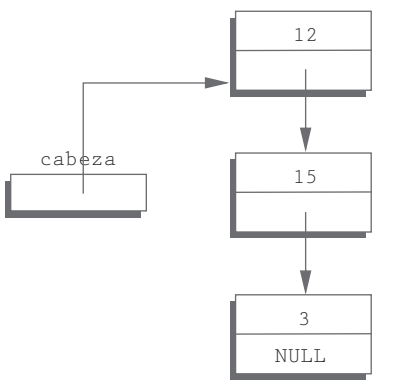
2. temp_ptr->enlace = cabeza;



3. cabeza = temp_ptr;



4. Después de la llamada a la función



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

Declaración de la función

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

typedef Nodo* NodoPtr;

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

Definición de la función

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

      temp_ptr->datos = el_numero;

      temp_ptr->enlace = cabeza;
      cabeza = temp_ptr;

}
```

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

RIESGO *Pérdida de nodos*

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

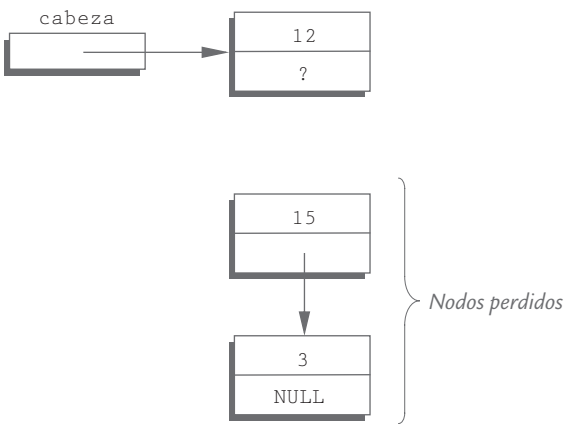
```
cabeza = new Nodo;
cabeza->datos = el_numero;
```

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

```
cabeza->enlace
```

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

CUADRO 15.5 *Nodos perdidos*



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

Búsqueda en una lista enlazada

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

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

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

algoritmo

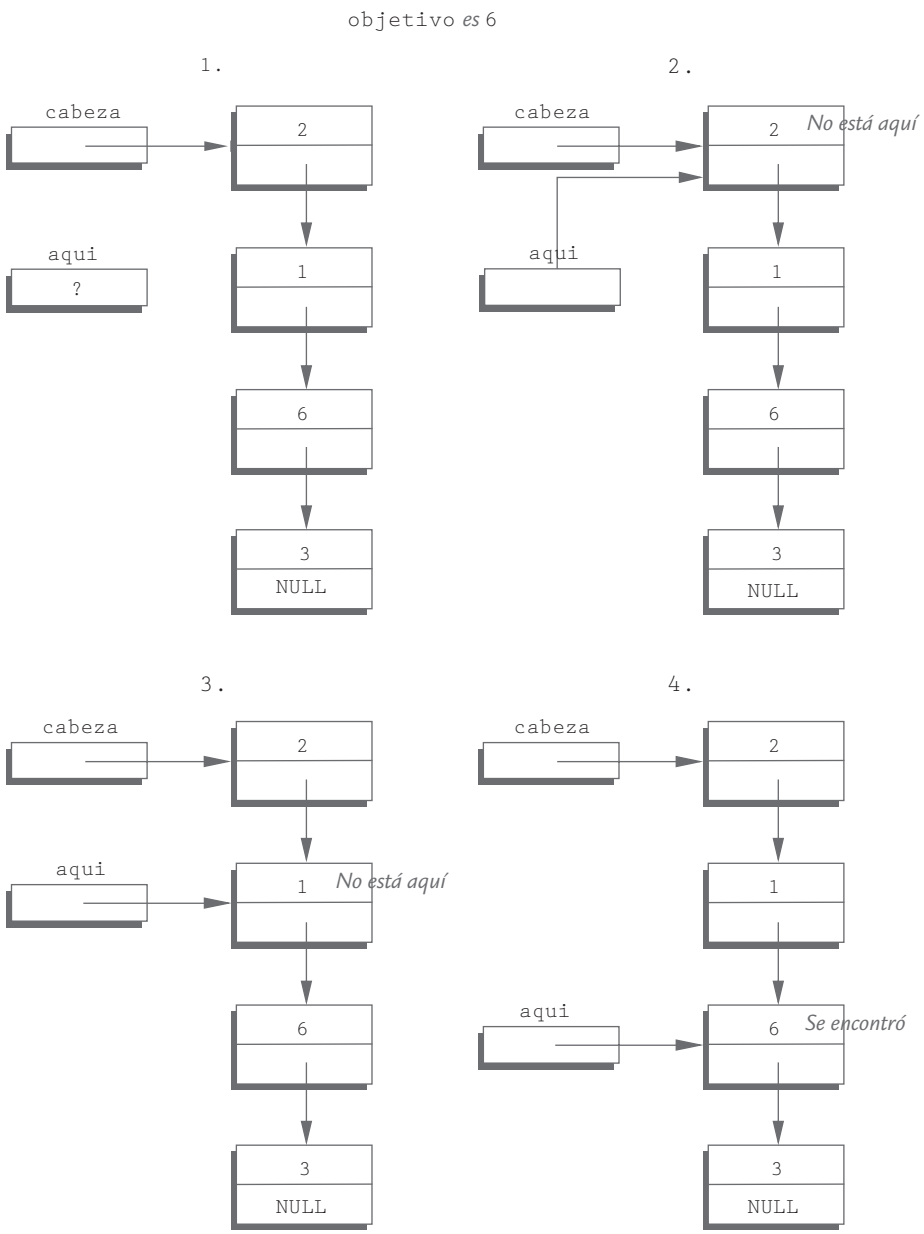
Pseudo-código para la función `busca`

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

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

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

CUADRO 15.6 *Búsqueda en una lista enlazada*



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

```
aqui->enlace
```

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

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

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

refinación del algoritmo

```
Versión preliminar del código para la función busca

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

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

lista vacía

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

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

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

Los apuntadores como iteradores

iterador

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

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

Declaración de la función

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

typedef Nodo* NodoPtr;

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

Definición de la función

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

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

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

Caso de lista vacía

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

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

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

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

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

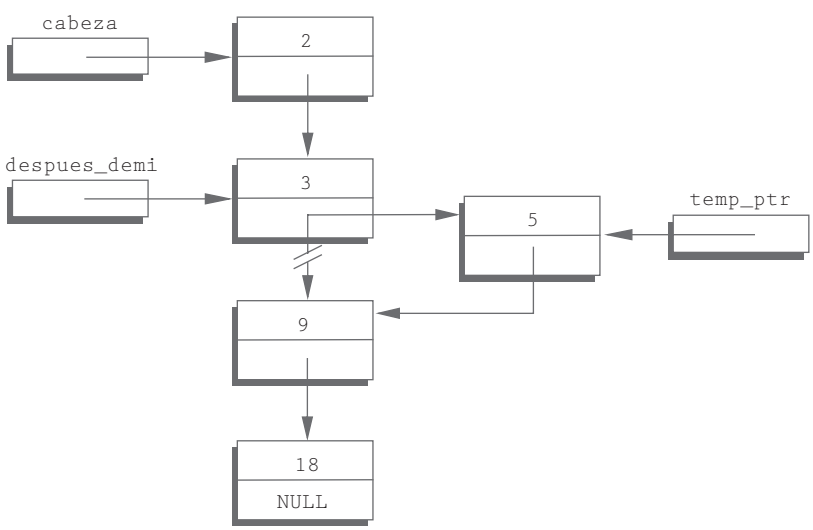
Las definiciones de `Nodo` y `NodoPtr` se proporcionan en el cuadro 15.7.

Inserción y remoción de nodos dentro de una lista

inserción en medio de una lista

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

CUADRO 15.8 *Inserción en medio de una lista enlazada*



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

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

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

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

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

inserción en los extremos

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

comparación con los arreglos

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

remoción de un nodo

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

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

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

```
delete descarta;
```

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

Declaración de la función

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

typedef Nodo* NodoPtr;

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

Definición de la función

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

    temp_ptr->datos = el_numero;

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

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

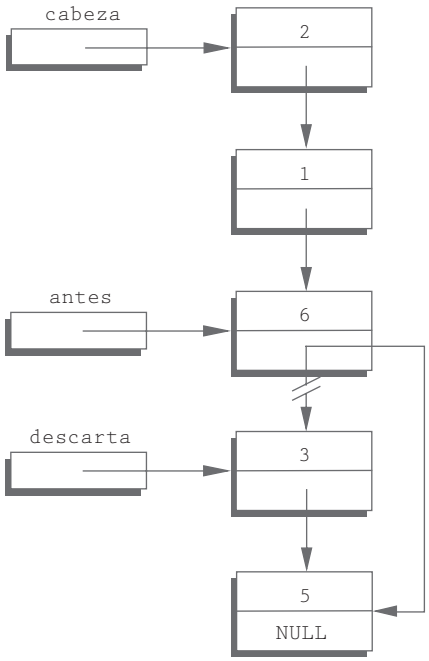
Si `cabeza1` y `cabeza2` son variables apuntadores y `cabeza1` apunta al nodo cabeza de una lista enlazada, la siguiente instrucción hará que `cabeza2` apunte al mismo nodo cabeza y por ende a la misma lista enlazada:

```
cabeza2 = cabeza1;
```

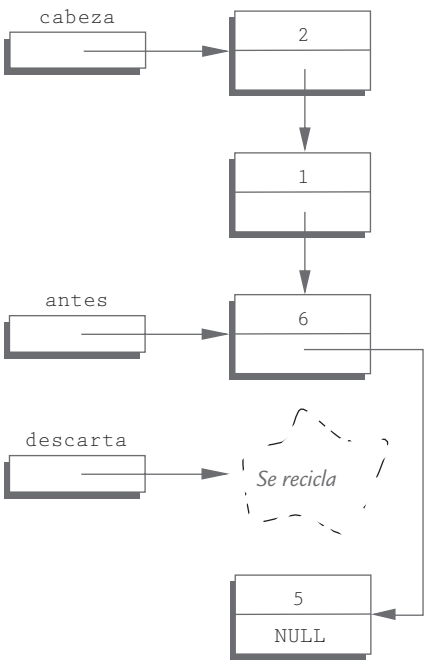
No obstante, debe recordar que sólo hay una lista enlazada y no dos. Si modifica la lista enlazada a la que apunta `cabeza1`, entonces también modificará la lista enlazada a la que apunta `cabeza2` ya que son la misma lista enlazada.

CUADRO 15.10 *Remoción de un nodo*

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



3. `delete descarta;`



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

Ejercicios de AUTOEVALUACIÓN

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

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

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

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

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

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

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

9. Seleccione una respuesta y explíquela.

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

   a) Más eficiente.

   b) Menos eficiente.

   c) Casi lo mismo.

   d) Dependiente del tamaño de las dos listas.

Variaciones en listas enlazadas

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

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

lista doblemente enlazada

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

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

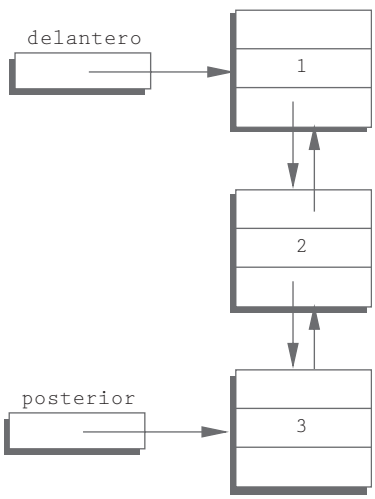
En vez de un solo apuntador hacia el nodo cabeza, una lista doblemente enlazada tiene un apuntador hacia cada uno de los dos nodos de los extremos. Podemos llamar a estos apuntadores `delantero` y `posterior`, aunque la elección de cuál es `delantero` y cuál `posterior` es arbitraria.

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

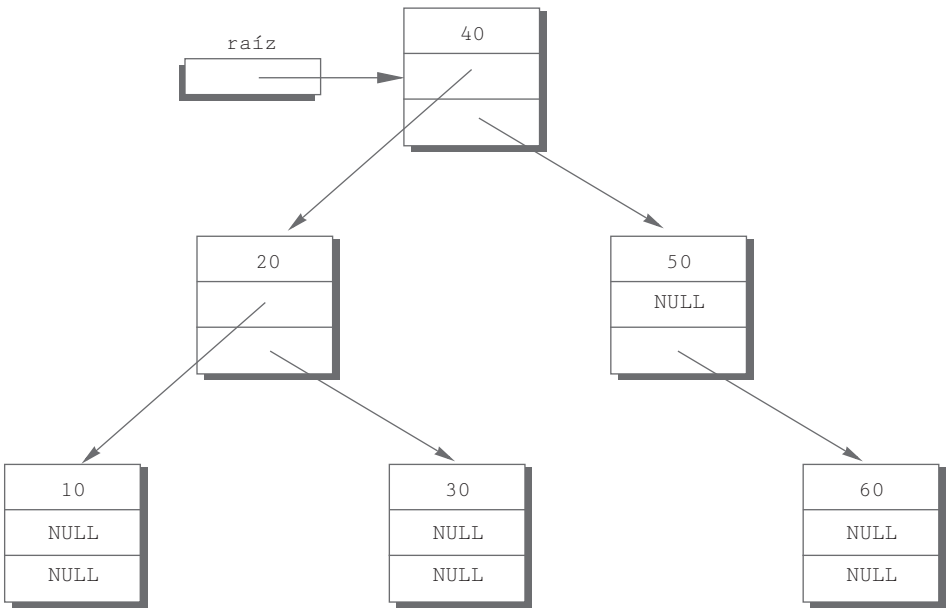
Un **árbol** es una estructura de datos que se estructura como se muestra en el cuadro 15.12. En especial, en un árbol podemos llegar a cualquier nodo desde el nodo superior (raíz) mediante cierta ruta que sigue los enlaces. Observe que no hay ciclos en un árbol. Si

árbol

CUADRO 15.11 *Una lista doblemente enlazada*



CUADRO 15.12 *Un árbol binario*



árbol binario

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

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

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

nodo raíz

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

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

nodos hoja

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

15.2 Pilas y colas

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

Mateo 19:30

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

Pilas

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

último en entrar, primero en salir

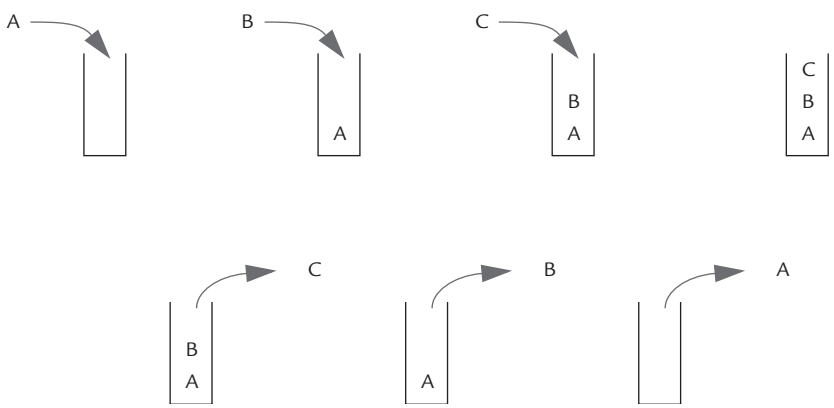
Las pilas se utilizan para muchas tareas de procesamiento de lenguajes. En el capítulo 13 vimos cómo el sistema computacional utiliza una pila para llevar el registro de las llamadas a funciones de C++. Sin embargo, aquí sólo realizaremos una aplicación muy simple. Nuestro objetivo en este ejemplo es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

EJEMPLO DE PROGRAMACIÓN
Una clase tipo pila

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

interfaz

CUADRO 15.13 *Una pila*



`mete` **y** `saca`

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

programa de aplicación

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

implementación

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

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

constructor predeterminado

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

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

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

    typedef PilaEstructura* PilaEstructuraPtr;

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

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

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

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

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

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

#endif //PILA_H
```

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

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

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

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

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

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

    return 0;
}
```

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

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

Diálogo de ejemplo

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

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

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

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

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

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

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

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

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

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

    temp_ptr->datos = el_simbolo;

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

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

    char resultado = superior->datos;

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

    delete temp_ptr;

    return resultado;
}
}//pilasavitch
```

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

constructor de copia

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

saca

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

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

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

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

```
delete temp_ptr;
```

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

destructor

Ejercicios de AUTOEVALUACIÓN

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

Colas

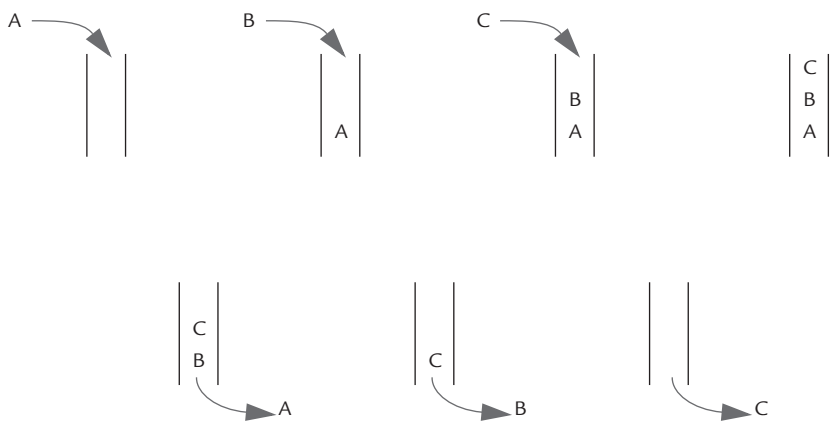
Una pila es una estructura de datos tipo "último en entrar/primero en salir". La **cola** es otra estructura de datos común, la cual maneja datos de la forma "primero en entrar/primero en salir" (PEPS). Una cola se comporta de igual forma que una línea de personas que esperan ser atendidas por el cajero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

cola
primero en entrar/primero en salir
delantera
posterior

Cola

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

CUADRO 15.17 *Una cola*



EJEMPLO DE PROGRAMACIÓN
Una clase tipo cola

interfaz

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

programa de aplicación

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

implementación

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

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

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

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

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

    typedef NodoCola* NodoColaPtr;

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

        Cola(const Cola& unaCola);

        ~Cola();

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

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

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

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

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

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

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

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

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

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

    return 0;
}
```

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

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

Diálogo de ejemplo

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

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

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

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

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

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

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

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

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

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

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

        char resultado = delantera->datos;

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

        delete descarta;

        return resultado;
    }
}
```

Ejercicios de AUTOEVALUACIÓN

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

Resumen del capítulo

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

Respuestas a los ejercicios de autoevaluación

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

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

2. La mejor respuesta es

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

No obstante, lo siguiente es también correcto:

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

3.
```
delete cabeza;
```

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

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

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

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

7. `p1 = p1->siguiente;`

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

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

```
delete descarta;
```

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

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

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

     temp_ptr->datos = el_simbolo;

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

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

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

       temp = temp->enlace;
```

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

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

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

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

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

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

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

Proyectos de programación

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

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

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

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

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

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

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

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

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

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

puede introducirse como

3x^4 + 7x^2 + 5

También podría introducirse como

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

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

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

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

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

12x^8 + 3x^2 + 0

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

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

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

   Los miembros de datos privados deberán incluir:

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

   y tal vez un apuntador más.

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

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

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

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

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

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

16 Herencia

16 Herencia

Con todos los dispositivos y medios para cargar.

WILLIAM SHAKESPEARE, *El rey Enrique IV, Parte III*

Introducción

La programación orientada a objetos es una técnica de programación popular y poderosa. Entre otras cosas, provee una nueva dimensión de abstracción conocida como *herencia*. Esto significa que puede definirse y compilarse una forma muy general de una clase. Después, pueden definirse versiones más especializadas de esa clase y pueden heredar todas las propiedades de la clase anterior. En todas las versiones de C++ hay herramientas para la herencia.

Prerrequisitos

La sección 16.1 utiliza material de los capítulos 2 al 11. Las secciones 16.2 y 16.3 utilizan material de los capítulos 12 y 15, además de los capítulos del 2 al 11 y la sección 16.1.

16.1 Fundamentos de la herencia

Si hay algo que deseamos cambiar en el niño, primero debemos examinarlo y ver si no es algo que podríamos cambiar mejor en nosotros mismos.

Carl Gustav Jung, *La integración de la personalidad*

herencia
clase derivada
clase base

Una de las características más poderosas de C++ es el uso de la herencia para derivar una clase de otra. La **herencia** es el proceso mediante el cual una nueva clase (conocida como **clase derivada**) se crea a partir de otra clase, conocida como **clase base**. Una clase derivada tiene de manera automática todas las variables y funciones miembro de la clase base, además de que puede tener funciones miembro adicionales y/o variables miembro adicionales.

En el capítulo 5 vimos que el decir que la clase `D` se deriva de otra clase `B` significa que la clase `D` tiene todas las características de la clase `B`, además de algunas características adicionales que pueden agregarse. Cuando una clase `D` se deriva de una clase `B`, decimos que `B` es la clase base y `D` es la clase derivada. También decimos que `D` es la **clase hija** y `B` es la **clase madre**.[1]

clase hija
clase madre

Por ejemplo, ya vimos el hecho de que la clase predefinida `ifstream` se deriva de la clase (predefinida) `istream` al agregar funciones miembro como `open` y `close`. El flujo `cin` pertenece a la clase de todos los flujos de entrada (es decir, a la clase `istream`), pero no pertenece a la clase de flujos de entrada de archivos (es decir, no pertenece a `ifstream`), en parte porque carece de las funciones miembro `open` y `close` de la clase derivada `ifstream`.

Clases derivadas

Supongamos que estamos diseñando un programa contable que tiene registros para los empleados asalariados y los empleados por horas. Hay una jerarquía natural para agrupar estas clases. Todas son clases de personas que comparten la propiedad de ser empleados.

Los empleados que reciben un sueldo por horas son un subconjunto de empleados. Otro subconjunto consiste en los empleados que reciben un sueldo fijo cada mes o cada semana. Aunque tal vez el programa no necesite ningún tipo que corresponda al conjunto de todos los empleados, pude ser útil pensar en términos del concepto más general de empleados. Por ejemplo, todos los empleados tienen nombres y números de seguro social, y las funciones miembro para establecer y modificar los nombres y los números de seguro social serán las mismas para los empleados asalariados y los empleados por horas.

Dentro de C++ podemos definir una clase llamada `Empleado` que incluya a todos los empleados, ya sean asalariados o por horas, y después podemos usar esta clase para definir clases para los empleados por horas y los empleados asalariados. Los cuadros 16.1 y 16.2 muestran una definición posible para la clase `Empleado`.

Podemos tener un objeto `Empleado` (sin diferenciación), pero nuestro motivo para definir la clase `Empleado` es que de esta forma podemos definir clases derivadas para distintos tipos de empleados. En especial, la definición de la función `imprime_cheque` siempre variará en las clases derivadas para que los distintos empleados puedan tener distintos tipos de cheques. Esto se refleja en la definición de la función `imprime_cheque` para la

[1] Algunos autores hablan de una *subclase* `D` y de la *superclase* `B` en vez de la clase derivada `D` y la clase base `B`. No obstante, hemos encontrado que los términos *clase derivada* y *clase base* son menos confusos. Sólo mencionamos esto como un esfuerzo por ayudarlo cuando lea otros libros.

CUADRO 16.1 *Interfaz para la clase base* Empleado

```
//Este es el archivo de encabezado empleado.h.
//Es la interfaz para la clase Empleado.
//Su principal uso es como clase base para derivar
//clases para distintos tipos de empleados.
#ifndef EMPLEADO_H
#define EMPLEADO_H

#include <string>
using namespace std;

namespace empleadosavitch
{

    class Empleado
    {
    public:
        Empleado( );
        Empleado(string el_nombre, string el_nss);
        string obtiene_nombre( ) const;
        string obtiene_nss( ) const;
        double obtiene_salario_neto( ) const;
        void asigna_nombre(string nuevo_nombre);
        void asigna_nss(string nuevo_nss);
        void asigna_salario_neto(double nuevo_salario_neto);
        void imprime_cheque( ) const;
    private:
        string nombre;
        string nss;
        double salario_neto;
    };

}//empleadosavitch

#endif //EMPLEADO_H
```

CUADRO 16.2 *Implementación para la clase base* `Empleado` ***(parte 1 de 2)***

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

namespace empleadosavitch
{
    Empleado::Empleado( ) : nombre("Sin nombre todavia"), nss("Sin numero todavia"),
    salario_neto(0)
    {
        //vacío de manera intencional
    }

    Empleado::Empleado(string el_nombre, string el_numero)
       : nombre(el_nombre), nss(el_numero), salario_neto(0)
    {
        //vacío de manera intencional
    }

    string Empleado::obtiene_nombre( ) const
    {
        return nombre;
    }

    string Empleado::obtiene_nss( ) const
    {
        return nss;
    }

    double Empleado::obtiene_salario_neto( ) const
    {
        return salario_neto;
    }

   void Empleado::asigna_nombre(string nuevo_nombre)
    {
        nombre = nuevo_nombre;
    }
```

CUADRO 16.2 *Implementación para la clase base* `Empleado` *(parte 2 de 2)*

```
void Empleado::asigna_nss(string nuevo_nss)
{
    nss = nuevo_nss;
}

void Empleado::asigna_salario_neto (double nuevo_salario_neto)
{
    salario_neto = nuevo_salario_neto;
}

void Empleado::imprime_cheque( ) const
{
    cout << "ERROR: SE LLAMO A LA FUNCION imprime_cheque PARA UN "
         << "EMPLEADO SIN DIFERENCIACION. Se abortara el programa."
         << "Verifique este error con el autor del programa.";
    exit(1);
}

}//empleadosavitch
```

clase `Empleado` (cuadro 16.2). No tiene mucho sentido imprimir un cheque para un `Empleado` tan generalizado. No sabemos nada acerca de los detalles del salario de este empleado. Por consecuencia, implementamos la función `imprime_cheque` de la clase `Empleado` de manera que el programa se detenga con un mensaje de error si se hace una llamada a `imprime_cheque` para un objeto de la clase base `Empleado`. Como verá, las clases derivadas tendrán suficiente información para redefinir la función `imprime_cheque`, de manera que pueda producir cheques de empleados que tengan sentido.

Una clase que se derive de la clase `Empleado` tendrá de manera automática todas las variables miembro de la clase `Empleado` (`nombre`, `nss` y `salario_neto`). Una clase que se derive de la clase `Empleado` también tendrá todas las funciones miembro de la clase `Empleado` tales como `imprime_cheque`, `obtiene_nombre`, `asigna_nombre` y las demás funciones miembro que se listan en el cuadro 16.1. Esto por lo general se expresa diciendo que la clase derivada **hereda** las variables y funciones miembro.

herencia

En los cuadros 16.3 (`EmpleadoPorHoras`) y 16.4 (`EmpleadoAsalariado`) se proporcionan los archivos de interfaz con las definiciones de dos clases derivadas de la clase `Empleado`. Hemos colocado a esta clase y a sus dos clases derivadas en el mismo espacio de nombres. C++ no requiere que se encuentren en el mismo espacio de nombres, pero como son clases relacionadas tiene sentido colocarlas en el mismo espacio de nombres. Primero hablaremos sobre la clase derivada `EmpleadoPorHoras` que se proporciona en el cuadro 16.3.

CUADRO 16.3 *Interfaz para la clase derivada* EmpleadoporHoras

```
//Este es el archivo de encabezado empleadoporhoras.h.
//Es la interfaz para la clase EmpleadoPorHoras.
#ifndef EMPLEADOPORHORAS_H
#define EMPLEADOPORHORAS_H

#include <string>
#include "empleado.h"

using namespace std;

namespace empleadosavitch
{

    class EmpleadoPorHoras : public Empleado
    {
    public:
        EmpleadoPorHoras( );
        EmpleadoPorHoras(string el_nombre, string el_nss,
                         double la_tarifa_sueldo, double las_horas);
        void asigna_tarifa(double nueva_tarifa_sueldo);
        double obtiene_tarifa( ) const;
        void asigna_horas(double horas_trabajadas);
        double obtiene_horas( ) const;
        void imprime_cheque( ) ;
    private:
        double tarifa_sueldo;
        double horas;
    };

}//empleadosavitch

#endif //EMPLEADOPORHORAS_H
```

Sólo debe listar la declaración de una función miembro heredada si desea modificar la definición de la función.

CUADRO 16.4 *Interfaz para la clase derivada* EmpleadoAsalariado

```
//Este es el archivo de encabezado empleadoasalariado.h.
//Es la interfaz para la clase EmpleadoAsalariado.
#ifndef EMPLEADOASALARIADO_H
#define EMPLEADOASALARIADO_H

#include <string>
#include "empleado.h"

using namespace std;

namespace empleadosavitch
{

    class EmpleadoAsalariado : public Empleado
    {
    public:
        EmpleadoAsalariado( );
        EmpleadoAsalariado (string el_nombre, string el_nss,
                                    double el_salario_semanal);
        double obtiene_salario( ) const;
        void asigna_salario(double nuevo_salario);
        void imprime_cheque( );
    private:
        double salario;//semanal
    };

}//empleadosavitch

#endif //EMPLEADOASALARIADO_H
```

Observe que la definición de una clase derivada empieza de la misma forma que cualquier otra definición de clase, sólo que se agregan dos puntos, la palabra reservada *public* y el nombre de la clase base en la primera línea de la definición de la clase, como en el siguiente ejemplo (del cuadro 16.3);

```
class EmpleadoPorHoras : public Empleado
{
```

La clase derivada (como EmpleadoPorHoras) recibe de manera automática todas las variables y funciones miembro de la clase base (tal como Empleado); además puede agregar variables y funciones miembro adicionales.

La definición de la clase `EmpleadoPorHoras` no menciona a las variables miembro `nombre`, `nss` y `salario_neto`, pero todo objeto de la clase `EmpleadoPorHoras` tiene variables miembro llamadas `nombre`, `nss` y `salario_neto`. Estas variables miembro se heredan de la clase `Empleado`. La clase `EmpleadoPorHoras` declara dos variables miembro adicionales llamadas `tarifa_sueldo` y `horas`. Por ende, todo objeto de la clase `EmpleadoPorHoras` tiene cinco variables miembro llamadas `nombre`, `nss`, `salario_neto`, `tarifa_sueldo` y `horas`. Hay que tener en cuenta que en la definición de una clase derivada (tal como `EmpleadoPorHoras`) sólo se listan las variables miembro agregadas. Las variables miembro definidas en la clase base no se mencionan. Éstas se proporcionan de manera automática a la clase derivada.

La clase `EmpleadoPorHoras` hereda las variables miembro y también las funciones miembro de la clase `Empleado`. Por lo tanto, la clase `EmpleadoPorHoras` hereda a las funciones miembro `obtiene_nombre`, `obtiene_nss`, `obtiene_salario_neto`, `asigna_nombre`, `asigna_nss`, `asigna_salario_neto` e `imprime_cheque` de la clase `Empleado`.

Además de las variables y funciones miembro heredadas, una clase derivada puede agregar nuevas variables y funciones miembro. Las nuevas variables miembro y las declaraciones de las nuevas funciones se listan en la definición de la clase. Por ejemplo, la clase derivada `EmpleadoPorHoras` agrega las dos variables miembro `tarifa_sueldo` y `horas`, y agrega las funciones miembro llamadas `asigna_tarifa`, `obtiene_tarifa`, `asigna_horas` y `obtiene_horas`. Esto se muestra en el cuadro 16.3. Observe que no tiene que dar las declaraciones de las funciones miembro heredadas, sólo aquellas definiciones que desee modificar; esta es la razón por la cual sólo listamos la función miembro `imprime_cheque` de la clase base `Empleado`. Por ahora no se preocupe por los detalles acerca de la definición del constructor para la clase derivada. En la siguiente subsección hablaremos sobre los constructores.

En el archivo de implementación para la clase derivada, tal como la implementación de `EmpleadoPorHoras` en el cuadro 16.5, se proporcionan las definiciones de todas las

Clases madre e hija

Al hablar sobre clases derivadas es común utilizar la terminología derivada de las relaciones familiares. A menudo, a una clase base se le conoce como **clase madre**. Por consecuencia, a una clase derivada se le llama **clase hija**. Esto suaviza mucho el lenguaje de la herencia. Por ejemplo, podemos decir que una clase hija hereda las variables y funciones miembro de su clase madre. Por lo general esta analogía se lleva un paso más allá. A una clase que sea la madre de una madre de una madre de otra clase (o de algún otro número de iteraciones tipo "madre de") se le conoce por lo común como **clase ancestro**. Si la clase A es un ancestro de la clase B, entonces a la clase B a menudo se le conoce como **descendiente** de la clase A.

Miembros heredados

Una clase derivada recibe de manera automática todas las variables miembro y todas las funciones miembro ordinarias de la clase base. (Como veremos más adelante en este capítulo, existen algunas funciones miembro especializadas como los constructores, que no se heredan de manera automática.) Se dice que estos miembros de la clase base son **heredados**. Estas funciones y variables miembro heredadas (con una excepción) no se mencionan en la definición de la clase derivada, sino que son miembros de la clase derivada de manera automática. Como explicamos en el libro, si deseamos modificar la definición de una función miembro heredada, se debe mencionar a esa función miembro heredada en la definición de la clase derivada.

CUADRO 16.5 ***Implementación para la clase derivada*** `EmpleadoporHoras` ***(parte 1 de 2)***

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

namespace empleadosavitch
{

    EmpleadoPorHoras::EmpleadoPorHoras( ) : Empleado( ), tarifa_sueldo(0), horas(0)
    {
        //vacío de manera intencional
    }

    EmpleadoPorHoras::EmpleadoPorHoras(string el_nombre, string el_numero,
                                       double la_tarifa_sueldo, double las_horas)
: Empleado(el_nombre, el_numero), tarifa_sueldo(la_tarifa_sueldo), horas(las_horas)
    {
        //vacío de manera intencional
    }

    void EmpleadoPorHoras::asigna_tarifa(double nueva_tarifa_sueldo)
    {
        tarifa_sueldo = nueva_tarifa_sueldo;
    }

    double EmpleadoPorHoras::obtiene_tarifa( ) const
    {
        return tarifa_sueldo;
    }
```

CUADRO 16.5 ***Implementación para la clase derivada*** `EmpleadoporHoras` ***(parte 2 de 2)***

```
void EmpleadoPorHoras::asigna_horas(double horas_trabajadas)
{
    horas = horas_trabajadas;
}

double EmpleadoPorHoras::obtiene_horas( ) const
{
    return horas;
}

void EmpleadoPorHoras::imprime_cheque( )
{
    asigna_salario_neto(horas * tarifa_sueldo);

    cout << "\n_____\n";
    cout << "Paguese a la orden de " << obtiene_nombre( ) << endl;
    cout << "La suma de " << obtiene_salario_neto( ) << " pesos\n";
    cout << "_____\n";
    cout << "Talon de cheque: NO NEGOCIABLE\n";
    cout << "Numero de empleado: " << obtiene_nss( ) << endl;
    cout << "Empleado por horas. \nHoras trabajadas: " << horas
         << " Tarifa: " << tarifa_sueldo << " Salario: "
         << obtiene_salario_neto( ) << endl;
    cout << "_____\n";
}

}//empleadosavitch
```

Hemos optado por establecer `salario_neto` *como parte de la función* `imprime_cheque`*, ya que es cuando se utiliza pero, en cualquier caso, ésta es una pregunta de contabilidad y no de programación.*
Pero tenga en cuenta que C++ nos permite eliminar la palabra `const` *en la función* `imprime_cheque` *cuando la redefinimos en una clase derivada.*

funciones miembro agregadas. No debemos proporcionar definiciones para las funciones miembro heredadas, a menos que se modifique la definición de la función miembro en la clase derivada, un punto que discutiremos a continuación.

La definición de una función miembro heredada puede modificarse en la definición de una clase derivada, de manera que tenga un significado en la clase derivada que sea distinto de lo que está en la clase base. A esto se le llama **redefinir** la función miembro heredada. Por ejemplo, la función miembro `imprime_cheque( )` se redefine en la definición de la clase derivada `EmpleadoPorHoras`. Para redefinir la definición de una función miembro, sólo necesitamos listarla en la definición de la clase y darle una nueva definición, de igual forma que como se haría con una función miembro que se agregue en la clase derivada. Esto se ilustra mediante la función predefinida `imprime_cheque( )` de la clase `EmpleadoPorHoras` (cuadros 16.3 y 16.5).

redefinir

`EmpleadoAsalariado` es otro ejemplo de una clase derivada de la clase `Empleado`. En el cuadro 16.4 se muestra la interfaz para la clase `EmpleadoAsalariado`. Un objeto que se declara de tipo `EmpleadoAsalariado` tiene todas las funciones y variables miembro de `Empleado`, junto con los nuevos miembros que se proporcionan en la definición de la clase `EmpleadoAsalariado`. Esto es cierto aún y cuando la clase `EmpleadoAsalariado` no liste a ninguna de las variables heredadas y sólo liste una función de la clase `Empleado`, la función `imprime_cheque`, cuya definición se modifica en la clase `EmpleadoAsalariado`. Esta clase tiene las tres variables miembro `nombre`, `nss` y `salario_neto`, junto con la variable miembro `salario`. Observe que no tiene que declarar las variables y funciones miembro de la clase `Empleado` como `nombre` y `asigna_nombre` para que un objeto `EmpleadoAsalariado` tenga estos miembros. La clase `EmpleadoAsalariado` obtiene estos miembros heredados de manera automática, sin necesidad de que el programa haga algo.

Observe que la clase `Empleado` tiene todo el código que es común para las dos clases `EmpleadoPorHoras` y `EmpleadoAsalariado`. Esto nos ahorra el problema de tener que escribir código idéntico dos veces, una para la clase `EmpleadoPorHoras` y otra para la clase `EmpleadoAsalariado`. La herencia nos permite reutilizar el código de la clase `Empleado`.

Constructores en clases derivadas

Un constructor en una clase base no se hereda en la clase derivada, pero puede invocar a un constructor de la clase base dentro de la definición de un constructor de la clase derivada, y eso es todo lo que necesita, o lo que se desea por lo general. Un constructor para una clase derivada utiliza un constructor de la clase base de una forma especial. Un constructor para la clase base inicializa todos los datos que se heredan de la clase base. Por ende, un constructor para una clase derivada empieza con una invocación de un constructor para la clase base.

Hay una sintaxis especial para invocar el constructor de la clase base, la cual se ilustra mediante las definiciones del constructor para la clase `EmpleadoPorHoras` que aparece en el cuadro 16.5. En el siguiente código hemos reproducido (con pequeños cambios en las interrupciones de línea para ajustar la columna de texto) una de las definiciones del constructor para la clase `EmpleadoPorHoras`, la cual tomamos de ese cuadro:

```
EmpleadoPorHoras::EmpleadoPorHoras(string el_nombre,
     string el_numero,double la_tarifa_sueldo,
   double las_horas)
```

```
    : Empleado(el_nombre, el_numero),
                tarifa_sueldo(la_tarifa_sueldo), horas(las_horas)
{
     //vacío de manera intencional
}
```

La porción que va después de los dos puntos es la sección de inicialización de la definición para el constructor `EmpleadoPorHoras::EmpleadoPorHoras`. La parte `Empleado(el_nombre,el_numero)` es una invocación del constructor de dos argumentos para la clase base `Empleado`. Observe que la sintaxis para invocar el constructor de la clase base es análoga a la sintaxis que se utiliza para establecer el valor de las variables miembro: la entrada `tarifa_sueldo(la_tarifa_sueldo)` asigna el valor de `la_tarifa_sueldo` a la variable miembro `tarifa_sueldo`; la entrada `Empleado(el_nombre, el_numero)` invoca al constructor de la clase base `Empleado` con los argumentos `el_nombre` y `el_numero`. Como todo el trabajo se realiza en la sección de inicialización, el cuerpo de la definición del constructor está vacío.

A continuación reproduciremos el otro constructor para la clase `EmpleadoPorHoras` del cuadro 16.5:

```
EmpleadoPorHoras::EmpleadoPorHoras( ) : Empleado( ), tarifa_sueldo(0),
                                                          horas(0)
{
    //vacío de manera intencional
}
```

En esta definición del constructor, se hace una llamada a la versión predeterminada (cero argumentos) del constructor de la clase base para inicializar las variables miembro heredadas. Siempre debemos incluir una invocación de uno de los constructores de la clase base en la sección de inicialización del constructor de una clase derivada.

Si una definición del constructor para una clase derivada no incluye una invocación de un constructor para la clase base, entonces se invocará de manera automática la versión

Un objeto de una clase derivada tiene más de un tipo

En la experiencia diaria, un empleado por horas es un empleado. En C++ es lo mismo. Como `EmpleadoPorHoras` es una clase derivada de la clase `Empleado`, todo objeto de la clase `EmpleadoPorHoras` puede utilizarse en cualquier parte en la que se puede usar un objeto de la clase `Empleado`. En especial, podemos usar un argumento de tipo `EmpleadoPorHoras` cuando una función requiera un argumento de tipo `Empleado`. Puede asignar un objeto de la clase `EmpleadoPorHoras` a una variable de tipo `Empleado`. (Pero esté advertido: no puede asignar un objeto `Empleado` a una variable de tipo `EmpleadoPorHoras`.) Desde luego que se aplica lo mismo para cualquier clase base y su clase derivada. Podemos usar un objeto de una clase derivada en cualquier parte en la que se permita un objeto de su clase base.

En forma más general, un objeto de un tipo de clase puede utilizarse en cualquier parte en que pueda utilizarse un objeto de cualquiera de sus clases ancestros. Si la clase `Hijo` se deriva de la clase `Ancestro` y la clase `Nieto` se deriva de la clase `Hijo`, entonces un objeto de la clase `Nieto` podrá usarse en cualquier parte en la que pueda usarse un objeto de la clase `Hijo` y el objeto de la clase `Nieto` también podrá usarse en cualquier parte en la que pueda usarse un objeto de la clase `Ancestro`.

predeterminada (cero argumentos) del constructor de la clase base. Por lo tanto, la siguiente definición del constructor predeterminado para la clase `EmpleadoPorHoras` (si se omite `Empleado( )`) será equivalente a la versión que acabamos de describir:

```
EmpleadoPorHoras::EmpleadoPorHoras( ) : tarifa_sueldo(0), horas(0)
{
      //vacío de manera intencional
}
```

No obstante, es preferible incluir siempre de manera explícita una llamada a un constructor de la clase base, aún y cuando se invoque de manera automática.

Un objeto de la clase derivada tiene todas las variables miembro de la clase base. Cuando se hace la llamada a un constructor de la clase derivada, hay que asignar memoria a estas variables miembro e inicializarlas. Esta asignación de memoria para las variables miembro heredadas debe realizarse por medio de un constructor para la clase base, ya que el constructor de la clase base es el lugar más conveniente para inicializar estas variables miembro heredadas. Por eso siempre debemos incluir una llamada a uno de los constructores de la clase base cuando defina un constructor para una clase derivada. Si no incluye una llamada a un constructor de la clase base (en la sección de inicialización de la definición de un constructor de la clase derivada), entonces se llamará de manera automática al constructor de la clase base. (Si no hay un constructor determinado para la clase base, se producirá una condición de error.)

orden de llamadas al constructor

La llamada al constructor de la clase base es la primera acción que realiza el constructor de una clase derivada. Por lo tanto, si la clase B se deriva de la clase A y la clase C se deriva de la clase B, cuando se crea un objeto de la clase C primero se hace la llamada a un constructor de la clase A, después se hace la llamada a un constructor de la clase B y por último se realizan las acciones incluidas en el constructor de la clase C.

Constructores en clases derivadas

Una clase derivada no hereda los constructores de su clase base. No obstante, cuando definamos un constructor para la clase derivada podemos y debemos incluir una llamada a un constructor de la clase base (dentro de la sección de inicialización de la definición del constructor).

Si no incluimos una llamada a un constructor de la clase base, entonces se llamará automáticamente al constructor predeterminado (cero argumentos) de la clase base cuando se llame al constructor de la clase derivada.

RIESGO *Uso de variables miembro privadas de la clase base*

Un objeto de la clase `EmpleadoPorHoras` (cuadros 16.3 y 16.5) hereda una variable miembro llamada `nombre` de la clase `Empleado` (cuadros 16.1 y 16.2). Por ejemplo, el siguiente código asigna el valor `"Josefina"` a la variable miembro `nombre` del objeto `pepe`. (Este código también asigna `"123-45-6789"` a la variable miembro `nss` y asigna un `0` a las variables `tarifa_sueldo` y `horas`.)

```
EmpleadoPorHoras pepe("Josefina", "123-45-6789", 0, 0);
```

Si desea modificar `pepe.nombre` a `"Pepe el Toro"` puede hacerlo de la siguiente manera:

```
pepe.asigna_nombre("Pepe el Toro");
```

Pero debe tener cuidado en cuanto a la manera en que manipula las variables miembro heredadas tales como `nombre`. La variable miembro `nombre` de la clase `EmpleadoPorHoras` se heredó de la clase `Empleado`, pero la variable miembro `nombre` es una variable miembro privada en la definición de la clase `Empleado`. Eso significa que se puede acceder de manera directa a `nombre` sólo dentro de la definición de una función miembro de la clase `Empleado`. No se puede acceder a una variable miembro (o función miembro) que sea privada en una clase base *por nombre* en la definición de una función miembro para *cualquier otra clase*, *no incluso en la definición de una función miembro de una clase derivada*. Por ende y aunque la clase `EmpleadoPorHoras` tiene una variable miembro llamada `nombre` (que hereda de la clase base `Empleado`), no es válido acceder de manera directa a la variable miembro `nombre` en la definición de cualquier función miembro en la definición de la clase `EmpleadoPorHoras`.

Por ejemplo, las siguientes líneas son del cuerpo de la función miembro `EmpleadoPorHoras::imprime_cheque` (que aparecen en el cuadro 16.5):

```
void EmpleadoPorHoras::imprime_cheque( )
{
    asigna_salario_neto(horas * tarifa_sueldo);

    cout << "\n_____\n";
    cout << "Paguese a la orden de" << obtiene_nombre( ) << endl;
    cout << "La suma de " << obtiene_salario_neto( ) << " pesos\n";
```

Tal vez se pregunte por qué necesitamos usar la función miembro `asigna_salario_neto` para asignar el valor de la variable miembro `salario_neto`. Podría verse tentado a reescribir el principio de la definición de la función miembro de la siguiente manera:

```
void EmpleadoPorHoras::imprime_cheque( )
{                                          uso inválido de salario_neto
   salario_neto = horas * tarifa_sueldo;
```

Como el comentario indica, esto no funcionará. La variable miembro `salario_neto` es una variable miembro privada en la clase `Empleado`, y aunque una clase derivada tal como `EmpleadoPorHoras` emplea la variable `salario_neto`, no puede acceder a ella en forma directa. Debemos utilizar alguna función miembro pública para acceder a la variable miembro `salario_neto`. La forma correcta de realizar la definición de `imprime_cheque` en la clase `EmpleadoPorHoras` es como lo hicimos en el cuadro 16.5 (parte del cual se mostró antes).

El hecho de que `nombre` y `salario_neto` sean variables heredadas privadas en la clase base también explica el por qué necesitamos usar las funciones de acceso `obtiene_nombre` y `obtiene_salario_neto` en la definición de `EmpleadoPorHoras::imprime_cheque`, en vez de sólo usar los nombres de las variables `nombre` y `salario_neto`. No puede mencionar una variable miembro privada heredada por su nombre. Debe utilizar las funciones miembro de acceso público y de mutación (como `obtiene_nombre` y `asigna_nombre`) que estén definidas en la clase base. (Recuerde que una *función de acceso* es una función que le permite acceder a las variables

miembro de una clase, y una *función de mutación* es una que le permite modificar las variables miembro de una clase. En el capítulo 6 vimos estos dos tipos de funciones.)

Para algunas personas, el hecho de que no se pueda acceder a una variable miembro privada de una clase base en la definición de una función miembro de una clase derivada a menudo les parece algo malo. Después de todo, si usted es un empleado por horas y desea cambiar su nombre, nadie dice: "Lo siento, `nombre` es una variable miembro privada de la clase `Empleado`". Después de todo, si usted es un empleado por horas, también es un empleado. En Java esto también es cierto; un objeto de la clase `EmpleadoPorHoras` también es un objeto de la clase `Empleado`. No obstante, las leyes sobre el uso de las variables miembro y las funciones miembro privadas deben ser como lo describimos antes, o de lo contrario se comprometería su privacidad. Si las variables miembro privadas de una clase fueran accesibles en las definiciones de las funciones miembro de una clase derivada, entonces cada vez que quisieramos acceder a una variable miembro privada sólo tendríamos que crear una clase derivada y acceder a esa variable miembro en una función miembro de esa clase, lo cual significaría que cualquiera podría acceder a todas las variables miembro privadas, si pusiera un poco de esfuerzo adicional. Este escenario contradictorio ilustra el problema, pero el principal problema son los errores no intencionales, no la subversión intencional. Si las variables miembro privadas de una clase fueran accesibles en las definiciones de las funciones miembro de una clase derivada, entonces podrían modificarse por error o de maneras no apropiadas. (Recuerde que las funciones de acceso y de mutación pueden proteger contra las modificaciones inapropiadas a las variables miembro.)

Más adelante, en la subsección titulada "El calificador *`protected`*" de este capítulo, hablaremos sobre una posible forma de evadir esta restricción sobre las variables miembro privadas de la clase base.

RIESGO *Las funciones miembro privadas en efecto no se heredan*

Como vimos en la sección de Riesgo anterior, no se puede acceder de manera directa a una variable miembro (o función miembro) que sea privada en una clase base fuera de la interfaz y la implementación de la clase base, *ni siquiera en la definición de una función miembro para una clase derivada*. Tenga en cuenta que las funciones miembro privadas son idénticas a las variables privadas en términos de no estar accesibles de manera directa. Pero en el caso de las funciones miembro, la restricción es más dramática. Se puede acceder a una variable miembro de manera indirecta por medio de una función miembro de acceso o de mutación. Una función miembro privada tan sólo no está disponible. Es tan simple como si la función miembro privada no se heredara.

Esto no debe ser un problema. Las funciones miembro privadas sólo deben usarse como funciones de ayuda, por lo que su uso debe limitarse a la clase en la que están definidas. Si deseamos utilizar una función miembro como función miembro de ayuda en una variedad de clases heredadas, entonces no es *sólo* una función de ayuda, y deberíamos hacerla pública.

El calificador *`protected`*

Como hemos visto, no se puede acceder a una variable miembro privada o a una función miembro privada en la definición o implementación de una clase derivada. Hay una clasificación de variables y funciones miembro que nos permite acceder a ellas por nombre sólo

en una clase derivada y en ningún otro lugar más, como por ejemplo en alguna clase que no sea derivada. Si utiliza el calificador *protected* en vez de *private* o *public* antes de una variable o función miembro de una clase, entonces para cualquier clase o función que no sea una clase derivada, el efecto será el mismo que como si la variable miembro se etiquetara como *private*; no obstante, en una clase derivada podrá acceder a la variable por su nombre.

Por ejemplo, consideremos la clase EmpleadoPorHoras que se derivó de la clase base Empleado. Es necesario que utilicemos funciones miembro de acceso y de mutación para manipular las variables miembro heredadas en la definición de EmpleadoPorHoras::imprime_cheque. Si todas las variables miembro privadas en la clase Empleado se identificaran con la palabra clave *protected* en vez de *private*, la definición de EmpleadoPorHoras::imprime_cheque en la clase derivada Empleado podría simplificarse de la siguiente manera:

protected

```
void EmpleadoPorHoras::imprime_cheque( )
//Sólo funciona si las variables miembro de Empleado se marcan como
//protected en vez de private.
{
    salario_neto = horas * tarifa_sueldo;

    cout << "\n_____________________________________________\n";
    cout << "Paguese a la orden de " << nombre( ) << endl;
    cout << "La suma de " << salario_neto( ) << " pesos\n";
    cout << "_____________________________________________\n";
    cout << "Talon de cheque: NO NEGOCIABLE\n";
    cout << "Numero de empleado: " << obtiene_nss( ) << endl;
    cout << "Empleado por horas.\nHoras trabajadas: " << horas
         << " Tarifa: " << tarifa_sueldo << " Salario: "
         << salario_neto
         << endl;
    cout << "_____________________________________________\n";
}
```

En la clase derivada EmpleadoPorHoras, se puede acceder a las variables miembro heredadas nombre, salario_neto y nss por su nombre, siempre y cuando se marquen como *protected* (en vez de *private*) en la clase base Empleado. No obstante, en cualquier clase que no se derive de la clase Empleado, estas variables miembro se tratarán como si fueran *private*.

Las variables miembro que se protegen en la clase base actúan como si también se marcaran como *protected* en cualquier clase derivada. Por ejemplo, supongamos que se define una clase llamada EmpleadoPorHorasMedioTiempo que se deriva de la clase EmpleadoPorHoras. La clase EmpleadoPorHorasMedioTiempo hereda todas las variables miembro de la clase EmpleadoPorHoras, incluyendo las variables miembro que EmpleadoPorHoras hereda de la clase Empleado. Por ende, la clase EmpleadoPorHorasMedioTiempo tendrá las variables miembro salario_neto, nombre y nss. Si estas variables miembro se marcaran como *protected* en la clase Empleado, entonces podrían usarse por su nombre en las definiciones de las funciones de la clase EmpleadoPorHorasMedioTiempo.

Una variable miembro que se marca como *protected* se trata de la misma forma como si se hubiera marcado *private*, excepto en las clases derivadas (y en las clases derivadas de clases derivadas, etcétera).

En primer lugar, incluimos una discusión de las variables miembro *protected* debido a que las verá en uso y deberá estar familiarizado con ellas. Muchas, pero no todas, las autoridades de programación dicen que el uso de variables miembro *protected* es un mal estilo. Dicen que compromete el principio de ocultar la implementación de la clase. Dicen que todas las variables miembro deberían marcarse como *private*. Si todas las variables miembro se marcan como *private*, no se podrá acceder a las variables miembro heredadas

por su nombre en las definiciones de funciones de las clases derivadas. Sin embargo, esto no es tan malo como parece. Se puede acceder de manera indirecta a las variables miembro `private` heredadas mediante la invocación de funciones heredadas que leen o modifican esas variables heredadas `private`. Como las autoridades difieren en cuanto a si se deben utilizar miembros protegidos o no, usted tendrá que decidir por su cuenta si desea o no utilizarlos.

Miembros protected

Si utiliza el calificador `protected` en vez de `private` o `public`, antes de una variable miembro de una clase, entonces para cualquier clase o función que no sea una clase derivada (o una clase derivada de una clase derivada, etcétera) la situación será la misma que si la variable miembro se hubiera marcado como `private`. No obstante, en la definición de una función miembro de una clase derivada, se puede acceder a la variable por su nombre. De manera similar, si utiliza el calificador `protected` antes de una función miembro de una clase, entonces para cualquier clase o función que no sea una clase derivada (o una clase derivada de una clase derivada, etcétera), será lo mismo que si la función miembro se hubiera marcado como `private`. Sin embargo, en la definición de una función miembro de una clase derivada puede utilizarse la función protected.

Los miembros protected heredados se heredan en la clase derivada como si se hubieran marcado como `protected` en la clase derivada. En otras palabras, si un miembro se marca como `protected` en una clase base, se podrá acceder a él por su nombre en las definiciones de todas las clases descendientes, no sólo en esas clases que se deriven de manera directa de la clase base.

Ejercicios de AUTOEVALUACIÓN

1. ¿Es válido el siguiente programa (suponga que se agregan las directivas `#include` y `using` apropiadas)?

```
void muestra_datos_empleado(const Empleado objeto);

int main( )
{
    EmpleadoPorHoras pepe("Pepe el Toro",
                                  "123-45-6789", 20.50, 40);
    EmpleadoAsalariado jefe("El Gran Jefe",
                                  "987-65-4321", 10500.50);
    muestra_datos_empleado(pepe);
    muestra_datos_empleado(jefe);

    return 0;
}

void muestra_datos_empleado(const Empleado objeto)
{
    cout << "Nombre: " << objeto.obtiene_nombre(  ) << endl;
    cout << "Numero de seguro social: "
         << objeto.obtiene_nss( ) << endl;
}
```

2. Proporcione la definición para una clase llamada `TipoListo` que sea una clase derivada de la clase base `Listo`, la cual reproduciremos a continuación. No se preocupe por los detalles relacionados con las directivas `#include` o los espacios de nombres.

```
class Listo
{
public:
    Listo(  );
    void imprime_respuesta(  ) const;
protected:
    int a;
    int b;
};
```

Esta clase deberá tener un campo de datos adicional llamado `loco` que sea de tipo `bool`, una función miembro adicional que no reciba argumentos y devuelva un valor de tipo `bool`, y los constructores apropiados. La nueva función deberá llamarse `esta_loco`. No necesitamos proporcionar las implementaciones, sólo la definición de la clase.

3. ¿Es válida la siguiente definición de la función miembro `esta_loco` en la clase derivada `TipoListo` que describimos en el ejercicio de autoevaluación 2? Explique su respuesta. (Recuerde que estamos preguntando si es válida, no si es una definición con sentido.)

```
bool TipoListo::esta_loco( ) const
{
   if (a > b)
      return loco;
   else
      return true;
}
```

Redefinición de funciones miembro

En la definición de la clase derivada `EmpleadoPorHoras` (cuadro 16.3) proporcionamos las declaraciones para las nuevas funciones miembro `asigna_tarifa`, `obtiene_tarifa`, `asigna_horas` y `obtiene_horas`. También proporcionamos la declaración de sólo una de las funciones miembro heredadas de la clase `Empleado`. Las funciones miembro heredadas cuyas declaraciones no proporcionamos (como `asigna_nombre` y `asigna_nss`) se heredan sin modificaciones. Tienen la misma definición en la clase `EmpleadoPorHoras` que en la clase base `Empleado`. Cuando definimos una clase derivada como `EmpleadoPorHoras`, sólo se listan las declaraciones para las funciones miembro heredadas cuyas definiciones deseamos modificar para que tengan una definición distinta en la clase derivada. Si analiza la implementación de la clase `EmpleadoPorHoras` que aparece en el cuadro 16.5, podrá ver que hemos redefinido la función miembro heredada `imprime_cheque`. La clase `EmpleadoAsalariado` también proporciona una nueva definición para la función miembro `imprime_cheque`, como se muestra en el cuadro 16.6. Lo que es más, las dos clases proporcionan distintas definiciones una de la otra. La función `imprime_cheque` se **redefine** en las clases derivadas.

función redefinida

CUADRO 16.6 ***Implementación para la clase derivada*** `EmpleadoAsalariado` ***(parte 1 de 2)***

```
//Éste es el archivo empleadoasalariado.cpp.
//Es la implementación de la clase EmpleadoAsalariado.
//La interfaz para la clase EmpleadoAsalariado está en el
//archivo de encabezado empleadoasalariado.h.
#include <iostream>
#include <string>
#include "empleadoasalariado.h"
using namespace std;

namespace empleadosavitch
{
    EmpleadoAsalariado::EmpleadoAsalariado( ) : Empleado( ), salario(0)
    {
        //vacío de manera intencional
    }

    EmpleadoAsalariado::EmpleadoAsalariado(string el_nombre, string el_numero,
                                    double el_salario_semanal)
                      : Empleado(el_nombre, el_numero), salario(el_salario_semanal)
    {
        //vacío de manera intencional
    }

    double EmpleadoAsalariado::obtiene_salario( ) const
    {
        return salario;
    }

    void EmpleadoAsalariado::asigna_salario(double nuevo_salario)
    {
        salario = nuevo_salario;
    }
```

CUADRO 16.6 *Implementación para la clase derivada* `EmpleadoAsalariado` *(parte 2 de 2)*

```
    void EmpleadoAsalariado::imprime_cheque( )
    {
        asigna_salario_neto(salario);
        cout << "\n_____________________________________________________\n";
        cout << "Paguese a la orden de " << obtiene_nombre( ) << endl;
        cout << "La suma de " << obtiene_salario_neto( ) << " pesos\n";
        cout << "_____________________________________________________\n";
        cout << "Talon de cheque: NO NEGOCIABLE \n";
        cout << "Numero de empleado: " << obtiene_nss( ) << endl;
        cout << "Empleado asalariado. Salario regular: "
             << salario << endl;
        cout << "_____________________________________________________\n";
    }
}//empleadosavitch
```

Redefinición de una función heredada

Una clase derivada hereda todas las funciones miembro (y variables miembro también) que pertenecen a la clase base. No obstante, si una clase derivada requiere una implementación distinta para una función miembro heredada, la función puede redefinirse en la clase derivada. Cuando definimos una función miembro, debemos listar su declaración en la definición de la clase derivada aunque la declaración sea la misma que en la clase base. Si no desea redefinir una función miembro que se hereda de la clase base, entonces no debe listarla en la definición de la clase derivada.

El cuadro 16.7 contiene un programa de demostración en el que se ilustra el uso de las clases derivadas `EmpleadoPorHoras` y `EmpleadoAsalariado`.

Comparación entre redefinición y sobrecarga

No confunda la *redefinición* de la definición de una función en una clase derivada con la *sobrecarga* del nombre de una función. Cuando se redefine la definición de una función, la nueva función que aparece en la clase derivada tiene el mismo número y tipo de parámetros. Por otro lado, si la función en la clase derivada tuviera un número distinto de parámetros o un parámetro de un tipo distinto al de la función en la clase base, entonces

CUADRO 16.7 *Uso de clases derivadas (parte 1 de 2)*

```
#include <iostream>
#include "empleadoporhoras.h"
#include "empleadoasalariado.h"
using std::cout;
using std::endl;

int main( )
{
     using namespace empleadosavitch;
     EmpleadoPorHoras pepe;
     pepe.asigna_nombre("Pepe el Toro");
     pepe.asigna_nss("123-45-6789");
     pepe.asigna_tarifa(20.50);
     pepe.asigna_horas(40);
     cout << "Cheque para " << pepe.obtiene_nombre( )
          << " por " << pepe.obtiene_horas( ) << " horas.\n";
     pepe.imprime_cheque( );
     cout << endl;

     EmpleadoAsalariado jefe("El Gran Jefe", "987-65-4321", 10500.50);
     cout << "Cheque para " << jefe.obtiene_nombre( ) << endl;
     jefe.imprime_cheque( );

     return 0;
}
```

Las funciones `asigna_nombre`, `asigna_nss`, `asigna_tarifa`, `asigna_horas` *y* `obtiene_nombre` *se heredan sin modificarse de la clase* `Empleado`.
La función `imprime_cheque` *se redefinió.*
La función `obtiene_horas` *se agregó a la clase derivada* `EmpleadoPorHoras`.

CUADRO 16.7 *Uso de clases derivadas (parte 2 de 2)*

Diálogo de ejemplo

```
Cheque para Pepe el Toro por 40 horas.

_____________________________________________
Paguese a la orden de Pepe el Toro
La suma de 820 pesos
_____________________________________________
Talon de cheque: NO NEGOCIABLE
Numero de empleado: 123-45-6789
Empleado por horas.
Horas trabajadas: 40 Tarifa: 20.5 Salario: 820
_____________________________________________

Cheque para El Gran Jefe

_____________________________________________
Paguese a la orden de El Gran Jefe
La suma de 10500.5 pesos
_____________________________________________
Talon de cheque: NO NEGOCIABLE
Numero de empleado: 987-65-4321
Empleado asalariado. Salario regular: 10500.5
_____________________________________________
```

la clase derivada tendría ambas funciones. Eso sería sobrecarga. Por ejemplo, suponga que agregamos una función con la siguiente declaración a la definición de la clase `EmpleadoPorHoras`:

```
void asigna_nombre(string primer_nombre, string apellido);
```

La clase `EmpleadoPorHoras` tendría su función de dos argumentos `asigna_nombre`, y también heredaría la siguiente función de un argumento `asigna_nombre`:

```
void asigna_nombre(string nuevo_nombre);
```

La clase `EmpleadoPorHoras` tendría dos funciones llamadas `asigna_nombre`. A esto le llamaríamos *sobrecargar* el nombre de función `asigna_nombre`.

Por otro lado, tanto la clase `Empleado` como la clase `EmpleadoPorHoras` definen una función con la siguiente declaración:

```
void imprime_cheque( );
```

En este caso, la clase `EmpleadoPorHoras` sólo tiene una función llamada `imprime_cheque`, pero la definición de la función `imprime_cheque` para la clase `EmpleadoPorHoras` es distinta de su definición para la clase `Empleado`. En este caso, se ha *redefinido* la función `imprime_cheque`.

Si confunde los términos redefinir y sobrecargar, tiene una consolación. Ambos son válidos. Eso quiere decir que es más importante aprender a utilizarlos que aprender a diferenciarlos, pero en algún momento dado tendrá que aprender a hacerlo.

Firma

La **firma** de una función es su nombre con la secuencia de tipos en la lista de parámetros, sin incluir la palabra clave *const* y sin incluir el signo `&`. Al sobrecargar el nombre de una función, las dos definiciones del nombre de la función deben tener distintas firmas que utilicen esta definición de firma.[2]

Si una función tiene el mismo nombre en una clase derivada que en la clase base pero tiene una firma distinta, se trata de sobrecarga, no de redefinición.

Acceso a una función base redefinida

Suponga que redefinimos una función de manera que tengamos una definición distinta en la clase derivada de la que teníamos en la clase base. La definición que se proporcionó en la clase base no se pierde completamente para los objetos de la clase derivada. No obstante, si deseamos invocar la versión de la función que se proporciona en la clase base con un objeto en la clase derivada, necesitamos alguna forma de decir "utiliza la definición de esta función como se da en la clase base (aunque soy un objeto de la clase derivada)". La manera de decir esto es mediante el uso del operador de resolución de alcance con el nombre de la clase base. A continuación veremos un ejemplo para que quede más claro.

Consideremos la clase base `Empleado` (cuadro 16.1) y la clase derivada `EmpleadoPorHoras` (cuadro 16.3). La función `imprime_cheque( )` se define en ambas clases. Ahora supongamos que tenemos un objeto de cada clase, como en

```
Empleado isabel_r;
EmpleadoPorHoras arlae_r;
```

Entonces

```
isabel_r.imprime_cheque( );
```

utiliza la definición de `imprime_cheque` que se proporciona en la clase `Empleado`, y

```
arlae_r.imprime_cheque( );
```

utiliza la definición de `imprime_cheque` que aparece en la clase `EmpleadoPorHoras`.

[2] Algunos compiladores pueden permitir la sobrecarga sin necesidad de excluir *const*, pero no podemos depender de esto y por lo tanto no deberíamos hacerlo. Por esta razón, algunas definiciones de *firma* incluyen el modificador `const`, pero ésta es una cuestión complicada que será mejor evitar hasta que se convierta en un experto.

Ahora supongamos que deseamos invocar la versión de `imprime_cheque` que se da en la definición de la clase base `Empleado` con el objeto `arlae_r` de la clase derivada como el objeto que hace la llamada para `imprime_cheque`. Para ello hay que hacer lo siguiente:

```
arlae_r.Empleado::imprime_cheque(  );
```

Desde luego que es muy poco probable que desee utilizar la versión de `imprime_cheque` que aparece en la clase particular `Empleado`, pero con otras clases y otras funciones tal vez sea conveniente algunas veces utilizar la definición de una función de una clase base con un objeto de una clase derivada. En el ejercicio de autoevaluación 6 se muestra un ejemplo.

Ejercicios de AUTOEVALUACIÓN

4. La clase `EmpleadoAsalariado` hereda las funciones `obtiene_nombre` e `imprime_cheque` (entre otras cosas) de la clase base `Empleado`, pero sólo aparece la declaración de la función `imprime_cheque` en la definición de la clase `EmpleadoAsalariado`. ¿Por qué no aparece la declaración de la función `obtiene_nombre` en la definición de `EmpleadoAsalariado`?

5. Proporcione una definición para una clase llamada `EmpleadoTitulado` que sea una clase derivada de la clase base `EmpleadoAsalariado` que se proporciona en el cuadro 16.4. La clase `EmpleadoTitulado` tiene una variable miembro adicional de tipo `string` llamada `titulo`. También tiene dos funciones miembro adicionales: `obtiene_titulo`, que no recibe argumentos y devuelve un valor `string`; y `asigna_titulo`, que es una función *`void`* que recibe un argumento de tipo `string`. También redefine la función miembro `asigna_nombre`. No necesita proporcionar las implementaciones, sólo la definición de la clase. No obstante, debe proporcionar todas las directivas *`#include`* y todas las directivas *`using namespace`* necesarias. Coloque la clase `EmpleadoTitulado` en el espacio de nombres `empleadosavitch`.

6. Proporcione las definiciones de los constructores para la clase `EmpleadoTitulado` que dio como respuesta al ejercicio de autoevaluación 5. Proporcione también la redefinición de la función miembro `asigna_ nombre`. Esta función debería insertar el título en el nombre. No se preocupe por los detalles relacionados con las directivas `#include` o los espacios de nombres.

16.2 Detalles acerca de la herencia

El demonio está en los detalles.

Dicho común

En esta sección presentaremos algunos de los detalles más sutiles sobre la herencia. La mayoría de los temas son relevantes sólo para las clases que utilizan arreglos dinámicos o apuntadores y otros datos dinámicos.

Funciones que no se heredan

Como regla general, si `Derivada` es una clase derivada con la clase base `Base`, entonces todas las funciones "normales" en la clase `Base` se heredan como miembros de la clase `Derivada`. No obstante, hay algunas funciones especiales que, para todos los fines prácticos, no se heredan. Hemos visto que, como cuestión práctica, los constructores no se heredan y que las funciones miembro privadas no se heredan. Los destructores tampoco se heredan.

En el caso del constructor de copia, éste no se hereda pero si no define un constructor de copia en una clase derivada (o en cualquier clase relacionada), C++ generará de manera automática un constructor de copia por usted. No obstante, este constructor de copia predeterminado sólo copia el contenido de las variables miembro y no funciona de manera correcta para las clases con apuntadores o datos dinámicos en sus variables miembro. Por ende, si las variables miembro de su clase involucran el uso de apuntadores, arreglos dinámicos u otros datos dinámicos, entonces debe definir un constructor de copia para la clase. Esto se aplica sin importar si la clase es derivada o no.

El operador de asignación = tampoco se hereda. Si la clase base `Base` define el operador de asignación pero la clase derivada `Derivada` no lo define, entonces la clase `Derivada` tendrá un operador de asignación pero será el operador de asignación predeterminado que C++ crea (cuando usted no define =); no tendrá nada que ver con el operador de asignación de la clase base definido en `Base`.

Es natural que los constructores, los destructores y el operador de asignación no se hereden. Para realizar sus tareas en forma correcta, necesitan información que la clase base no posee. Para realizar sus funciones en forma correcta, necesitan saber acerca de las nuevas variables miembro que se introducen en la clase derivada.

Operadores de asignación y constructores de copia en clases derivadas

Los operadores de asignación y los constructores sobrecargados no se heredan. Sin embargo, pueden (y en casi todos los casos deben) usarse en las definiciones de los operadores de asignación y los constructores de copia sobrecargados en las clases derivadas.

Cuando se sobrecarga el operador de asignación en una clase derivada, por lo general se utiliza el operador de asignación sobrecargado de la clase base. Presentaremos un esquema de cómo se escribe el código para hacer esto. Para ayudar a comprender el esquema de código, recuerde que un operador de asignación sobrecargado debe definirse como una función miembro de la clase.

Si `Derivada` es una clase que se deriva de `Base`, entonces la definición del operador de asignación sobrecargado para la clase `Derivada` comenzaría por lo general con algo así como esto:

```
Derivada& Derivada::operator =(const Derivada& lado_derecho)
{
    Base::operator =(lado_derecho);
```

La primera línea de código en el cuerpo de la definición es una llamada al operador de asignación sobrecargado de la clase `Base`. Esto se encarga de las variables miembro heredadas y sus datos. La definición del operador de asignación sobrecargado pasaría entonces a establecer las nuevas variables miembro que se introdujeron en la definición de la clase `Derivada`. Existe una situación similar para definir el constructor de copia en una clase derivada.

Si `Derivada` es una clase que se deriva de `Base`, entonces la definición del constructor de copia para la clase `Derivada` utilizaría por lo general el constructor de copia para la clase `Base`, para establecer las variables miembro heredadas y sus datos. Lo más común sería que el código empezara con algo así:

```
Derivada::Derivada(const Derivada& objeto)
                        : Base(objeto), <tal vez haya más inicializaciones>
{
```

La invocación del constructor de copia `Base(objeto)` de la clase base establece las variables miembro heredadas del objeto de la clase `Derivada` que se va a crear. Tenga en cuenta que como `objeto` es de tipo `Derivada`, también es de tipo `Base`; por lo tanto, `objeto` es un argumento válido para el constructor de copia de la clase `Base`.

Desde luego que estas técnicas no funcionarán a menos que usted tenga un operador de asignación y un constructor de copia para la base clase que funcionen a la perfección. Esto significa que la definición de la clase base deberá incluir un constructor de copia y que el operador de asignación predeterminado que se crea automáticamente deberá funcionar de forma correcta para la clase base, o ésta deberá tener una definición sobrecargada apropiada del operador de asignación.

Destructores en clases derivadas

Si una clase base tiene un destructor que funcione sin problemas, entonces es fácil definir un destructor que funcione en forma correcta, en una clase derivada de la clase base. Cuando se invoca el destructor para la clase derivada, éste invoca de manera automática al destructor de la clase base, por lo que no hay necesidad de escribir una llamada al destructor de la clase base en forma explícita; siempre ocurre en forma automática. Por ende, el destructor de la clase derivada sólo necesita preocuparse por usar *`delete`* en las variables miembro (y en cualquier dato hacia el que apunten) que se agreguen en la clase derivada. El trabajo del destructor de la clase base es invocar a *`delete`* sobre las variables miembro heredadas.

Si la clase B se deriva de la clase A y la clase C se deriva de la clase B entonces, cuando un objeto de la clase C queda fuera de alcance, primero se hace la llamada al destructor de la clase C, después al destructor de la clase B y por último al destructor de la clase A. Observe que el orden en el que se llaman los destructores es el orden inverso en el que se llaman los constructores.

Ejercicios de AUTOEVALUACIÓN

7. Sabemos que un operador de asignación y un constructor de copia sobrecargados no se heredan. ¿Significa esto que si no se define un operador de asignación o un constructor de copia sobrecargado para una clase derivada, entonces esa clase derivada no tendrá operador de asignación ni constructor de copia?

8. Suponga que `Hijo` es una clase derivada de la clase `Padre`, y que `Nieto` es una clase derivada de la clase `Hijo`. Esta pregunta se relaciona con los constructores y los destructores para las tres clases `Padre`, `Hijo` y `Nieto`. Cuando se invoca un constructor para la clase `Nieto`, ¿qué constructores se invocan y en qué orden? Cuando se invoca el destructor para la clase `Nieto`, ¿qué destructores se invocan y en qué orden?

9. Proporcione las definiciones para la función miembro `agrega_valor`, para el constructor de copia, el operador de asignación sobrecargado y el destructor para la siguiente clase. Se pretende usar esta clase para un arreglo lleno en forma parcial. La variable miembro `numero_usado` contiene el número de posiciones en el arreglo que están llenas. Le daremos la definición del otro constructor para ayudarlo a empezar.

```
#include <iostream>
#include <cstdlib>
using namespace std;

class ArregloLlenoParcial
{
```

```
public:
    ArregloLlenoParcial(int tamanio_arreglo);
    ArregloLlenoParcial(const ArregloLlenoParcial& objeto);
    ~ArregloLlenoParcial();
    void operator = (const ArregloLlenoParcial& lado_derecho);
    void agrega_valor(double nueva_entrada);
    //Tal vez habría más funciones miembro
    //pero son irrelevantes para este ejercicio.
protected:
    double *a;
    int numero_maximo;
    int numero_usado;
};

ArregloLlenoParcial::ArregloLlenoParcial(int tamanio_arreglo)
                  : numero_maximo(tamanio_arreglo), numero_usado(0)
{
    a = new double[numero_maximo];
}
```

(Muchas de las autoridades dirían que las variables miembro deberían ser privadas en vez de protegidas. Nosotros estamos de acuerdo, pero el uso de `protected` nos ayudará a obtener una mejor práctica; además debe tener experiencia con las variables protegidas, ya que algunos programadores las utilizan.)

10. Defina una clase llamada `ArregloLlenoParcialCMax` que sea una clase derivada de la clase `ArregloLlenoParcial`. La clase `ArregloLlenoParcialCMax` deberá tener una variable miembro adicional llamada `valor_maximo` que contenga el valor máximo almacenado en el arreglo. Defina una función miembro de acceso llamada `obtiene_max` que devuelva el valor máximo almacenado en el arreglo. Redefina la función miembro `agrega_valor` y defina dos constructores, uno de los cuales deberá contener un argumento `int` para el número máximo de entradas en el arreglo. Defina también un constructor de copia, un operador de asignación sobrecargado y un destructor. (Una clase real tendría más funciones miembro, pero con éstos bastará para el ejercicio.)

16.3 Polimorfismo

Lo hice a mi manera.

Frank Sinatra

función del polimorfismo

El polimorfismo se refiere a la habilidad de asociar múltiples significados a un nombre de función. Según el uso que se le da hoy en día, el **polimorfismo** se refiere a una manera muy particular de asociar varios significados a un solo nombre de función. Es decir, el `polimorfismo` se refiere a la habilidad de asociar múltiples significados a un nombre de función por medio de un mecanismo especial conocido como *vinculación postergada*. El polimorfismo es uno de los componentes clave de una filosofía de programación conocida como *programación orientada a objetos*. El tema de esta sección es la vinculación postergada y, por lo tanto, también el polimorfismo.

Vinculación postergada

Una *función virtual* es una función que, en cierto sentido, puede usarse antes de definirse. Por ejemplo, un programa de gráficos puede tener varios tipos de figuras como rectángulos,

círculos, óvalos, etcétera. Cada figura podría ser un objeto de una clase distinta. Por ejemplo, la clase `Rectangulo` podría tener variables miembro para la altura, la anchura y el punto central, mientras que la clase `Circulo` podría tener variables miembro para un punto central y un radio. En un proyecto de programación bien diseñado, es probable que todas esas figuras fueran descendientes de una clase madre individual llamada, por ejemplo, `Figura`. Ahora suponga que desea que una función dibuje una figura en la pantalla. Para dibujar un círculo se necesitan distintas instrucciones de las que se necesitan para dibujar un rectángulo. Por lo tanto, cada clase necesita tener una función distinta para dibujar su tipo de figura. No obstante y como las funciones pertenecen a las clases, todas pueden llamarse `dibuja`. Si `r` es un objeto `Rectangulo` y `c` es un objeto `Circulo`, entonces `r.dibuja( )` y `c.dibuja( )` pueden ser funciones que se implementen con distinto código. Todo esto no es nuevo, pero ahora veremos algo nuevo: las *funciones virtuales* definidas en la clase madre `Figura`.

Ahora, la clase madre `Figura` puede tener funciones que se apliquen a todas las figuras. Por ejemplo, podría tener una función llamada `centro` que desplace una figura hacia el centro de la pantalla al borrarla y después volver a dibujarla en el centro de la pantalla. `Figura::centro` podría utilizar la función `dibuja` para volver a dibujar la figura en el centro de la pantalla. Si piensa en utilizar la función heredada `centro` con figuras de las clases `Rectangulo` y `Circulo`, empezará a ver que hay complicaciones.

Para aclarar el punto y hacerlo más dramático, supongamos que la clase `Figura` ya está escrita y en uso, y que cierto tiempo después agregamos una clase para un nuevo tipo de figura, por decir, la clase `Triangulo`. Ahora, `Triangulo` puede ser una clase derivada de la clase `Figura` y por lo tanto heredará la función `centro`; de esta forma, la función `centro` debería aplicarse a (y funcionar en forma correcta para) todos los objetos `Triangulo`. Pero existe una complicación. La función `centro` utiliza `dibuja` y esta función es distinta para cada tipo de figura. La función heredada `centro` (si no se hace nada especial) utilizará la definición de la función `dibuja` que se proporciona en la clase `Figura`, y esa función `dibuja` no funciona en forma correcta para los objetos `Triangulo`. Queremos que la función heredada `centro` utilice la función `Triangulo::dibuja` en vez de la función `Figura::dibuja`. Pero la clase `Triangulo`, y por consecuencia la función `Triangulo::dibuja`, ¡ni siquiera se había escrito cuando la función `centro` (definida en la clase `Figura`) se escribió y se compiló! ¿Cómo es posible que la función `centro` pueda trabajar en forma correcta para los objetos `Triangulo`? El compilador no sabía nada acerca de `Triangulo::dibuja` cuando se compiló `centro`. La respuesta es que se puede aplicar siempre y cuando `dibuja` sea una *función virtual*.

función virtual

Cuando hacemos una **función virtual**, le indicamos al compilador "No queremos saber cómo se implementa esta función. Espera hasta que se utilice en un programa y luego obtén la implementación de la instancia del objeto". La técnica de esperar hasta el tiempo de ejecución para determinar la implementación de un procedimiento se conoce como **vinculación postergada** o **vinculación dinámica**. Las funciones virtuales son la forma en que C++ proporciona la vinculación postergada. Pero ya fue suficiente introducción. Necesitamos un ejemplo para ver cómo funciona esto (y para enseñarle a utilizar funciones virtuales en sus programas). Para poder explicar los detalles de las funciones virtuales en C++, utilizaremos un ejemplo simplificado de un área de aplicación distinta al dibujo de figuras.

vinculación postergada
vinculación dinámica

Funciones virtuales en C++

Suponga que diseñaremos un programa de contabilidad para una tienda de refacciones automotrices. Deseamos hacer que el programa sea versátil, pero no estamos seguros de poder tomar en cuenta todas las situaciones posibles. Por ejemplo, deseamos llevar el registro de las ventas pero no podemos anticipar todos los tipos de ventas. Al principio sólo habrá

ventas regulares para los clientes al detalle que vayan a la tienda a comprar una refacción específica. No obstante, más adelante tal vez queramos agregar ventas con descuentos, o ventas por correspondencia con un cargo por envío. Todas estas ventas serán para un artículo con un precio básico y en última instancia producirán cierta cuenta. Para una venta simple la cuenta será sólo el precio básico, pero si más adelante agregamos descuentos, entonces ciertos tipos de cuentas dependerán también del tamaño del descuento. Nuestro programa necesitará calcular las ventas totales diarias, que de manera intuitiva sólo deberían ser la suma de todas las cuentas de ventas individuales. Tal vez también necesitemos calcular las ventas mayor y menor del día, o la venta promedio para ese día. Todo esto puede calcularse a partir de las cuentas individuales, pero las funciones para calcular las cuentas no se agregarán sino hasta después, cuando decidamos con qué tipos de ventas estaremos tratando. Para dar cabida a esto, haremos que la función para calcular la cuenta sea una función virtual. (Por cuestión de simpleza en este primer ejemplo, supondremos que cada venta es sólo para un artículo, aunque con las clases derivadas y las funciones virtuales podríamos (pero no lo haremos aquí) contabilizar las ventas de varios artículos.)

Los cuadros 16.8 y 16.9 contienen la interfaz y la implementación para la clase `Venta`. Todos los tipos de ventas serán clases derivadas de la clase `Venta`. Esta clase corresponde a las ventas simples de un solo artículo, sin agregar descuentos o cargos. Observe la palabra reservada *`virtual`* en la declaración de la función `cuenta` (cuadro 16.8). Observe (cuadro 16.9) que la función miembro `ahorros` y el operador sobrecargado `<` utilizan la función `cuenta`. Ya que `cuenta` se declara como una función virtual, más adelante podremos definir clases derivadas de la clase `Venta` y sus versiones de la función `cuenta`, y las definiciones de la función miembro `ahorros` y el operador sobrecargado `<` (que proporcionamos con la clase `Venta`) utilizarán la versión de la función `cuenta` que corresponda con el objeto de la clase derivada.

Por ejemplo, el cuadro 16.10 muestra la clase derivada `VentaDescuento`. Observe que la clase `VentaDescuento` requiere una definición distinta para su versión de la función `cuenta`. Sin embargo, cuando la función miembro `ahorros` y el operador sobrecargado `<` se utilicen con un objeto de la clase `VentaDescuento`, utilizarán la versión de la definición de la función `cuenta` que se proporcionó con la clase `VentaDescuento`. Es evidente que esto es un truco elegante de C++. Considere la llamada a la función `d1.ahorros(d2)` para los objetos `d1` y `d2` de la clase `VentaDescuento`. La definición de la función `ahorros` (incluso para un objeto de la clase `VentaDescuento`) se proporciona en el archivo de implementación para la clase base `Venta`, que se compiló antes de que hubiéramos pensado siquiera en la clase `VentaDescuento`. Aún así, en la llamada a la función `d1.ahorros(d2)` la línea que llama a la función `cuenta` sabe lo suficiente como para utilizar la definición de la función `cuenta` que se proporciona en la clase `VentaDescuento`.

¿Cómo funciona esto? Para poder escribir programas en C++ tan sólo puede suponer que ocurre por obra de magia, pero la verdadera explicación se proporcionó en la introducción a esta sección. Cuando etiqueta una función como *`virtual`*, le está indicando al entorno de C++ que "espere a que esta función se utilice en un programa y después obtenga la implementación que corresponda al objeto que hace la llamada".

El cuadro 16.11 muestra un programa de ejemplo en el que se ilustra cómo trabajan la función virtual `cuenta` y las funciones que utilizan a `cuenta` en un programa completo.

Hay una variedad de detalles técnicos que necesita conocer para poder utilizar funciones virtuales en C++. A continuación los enlistaremos:

- Si una función va a tener una definición distinta en una clase derivada a la definición de la clase base y usted quiere que sea una función virtual, debe agregar la palabra clave *`virtual`* a la declaración de la función en la clase base. No necesita agregar la palabra reservada *`virtual`* a la declaración de la función en la clase

CUADRO 16.8 *Interfaz para la clase* `Venta`

```
//Este es el archivo de encabezado venta.h.
//Es la interfaz para la clase Venta.
//Venta es una clase para ventas simples.
#ifndef VENTA_H
#define VENTA_H

#include <iostream>
using namespace std;

namespace ventasavitch
{

    class Venta
    {
    public:
        Venta();
        Venta(double el_precio);
        virtual double cuenta() const;
        double ahorros(const Venta& otro) const;
        //Devuelve los ahorros si compra otro en vez del objeto que hace la llamada.
    protected:
        double precio;
    };

    bool operator < (const Venta& primera, const Venta& segunda);
    //Compara dos ventas para ver cuál es mayor.

}//ventasavitch

#endif // VENTA_H
```

derivada. Si una función es virtual en la clase base, entonces de manera predeterminada es virtual en la clase derivada. (No obstante, es buena idea etiquetar la declaración de la función en la clase derivada como `virtual`, aún y cuando no se requiera.)

- La palabra reservada `virtual` se agrega a la declaración de la función y no a la definición de la función.
- No se obtiene una función virtual y los beneficios de las funciones virtuales, a menos que utilice la palabra clave `virtual`.

CUADRO 16.9 *Implementación de la clase base* Venta

```
//Este es el archivo de implementación: sale.cpp
//Es la implementación para la clase Venta.
//La interfaz para la clase Venta está en
//el archivo de encabezado venta.h.
#include "venta.h"

namespace ventasavitch
{

    Venta::Venta() : precio(0)
    {}

    Venta::Venta(double el_precio) : precio(el_precio)
    {}

    double Venta::cuenta() const
    {
        return precio;
    }

    double Venta::ahorros(const Venta& otro) const
    {
        return ( cuenta() - otro.cuenta() );
    }

    bool operator < (const Venta& primera, const Venta& segunda)
    {
        return (primera.cuenta() < segunda.cuenta());
    }

}//ventasavitch
```

CUADRO 16.10 *La clase derivada* VentaDescuento

```
            //Ésta es la interfaz para la clase VentaDescuento.
#ifndef VENTADESCUENTO_H
#define VENTADESCUENTO_H
#include "venta.h"

namespace ventasavitch
{
    class VentaDescuento : public Venta
    {
    public:
        VentaDescuento();
        VentaDescuento(double el_precio, double el_descuento);
        //El descuento se expresa como un porcentaje del precio.
        virtual double cuenta() const;
    protected:
        double descuento;
    };
}//ventasavitch
#endif //VENTADESCUENTO_H
```

Este es el archivo ventadescuento.h.

La palabra clave virtual *no se requiere aquí, pero es buen estilo incluirla.*

```
              //Ésta es la implementación para la clase VentaDescuento.
#include "ventadescuento.h"

namespace ventasavitch
{
    VentaDescuento::VentaDescuento() : Venta(), descuento(0)
    {}
    VentaDescuento::VentaDescuento(double el_precio, double el_descuento)
              : Venta(el_precio), descuento(el_descuento)
    {}
    double VentaDescuento::cuenta() const
    {
        double fraccion = descuento/100;
        return (1 - fraccion)*precio;
    }
}//ventasavitch
```

Éste es el archivo ventadescuento.cpp.

CUADRO 16.11 *Uso de una función virtual*

```
//Demuestra el funcionamiento de la función virtual cuenta.
#include <iostream>
#include "venta.h" //En realidad no se necesita, pero es seguro gracias a ifndef.
#include "ventadescuento.h"
using namespace std;
using namespace ventasavitch;

int main()
{
    Venta simple(10.00);//Un artículo a $10.00
    VentaDescuento descuento(11.00, 10);//Un artículo a $11.00 con un descuento del 10%.

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);

    if (descuento < simple)
    {
        cout << "El articulo con descuento es mas barato.\n";
        cout << "El ahorro es de $" << simple.ahorros(descuento) << endl;
    }
    else
        cout << "El articulo con descuento no es mas barato.\n";

    return 0;
}
```

Diálogo de ejemplo

```
El articulo con descuento es mas barato.
El ahorro es de $0.10
```

Ya que las funciones virtuales son tan grandiosas, ¿por qué no hacer todas las funciones virtuales? Casi la única razón por la que no siempre se deben usar funciones virtuales es la eficiencia. El compilador y el entorno en tiempo de ejecución necesitan hacer mucho más trabajo para las funciones virtuales, por lo que si usted marca más funciones `virtual` de las que necesita, su programa será menos eficiente.

Sobrescritura

Cuando la definición de una función virtual se modifica en una clase derivada, los programadores dicen que la definición de la función se **sobrescribió**. En la literatura de C++, algunas veces se hace una distinción entre los términos *redefinir* y *sobrescribir*. Ambos términos se refieren a la modificación de la función en una clase derivada. Si la función es virtual, se llama *sobrescritura*. Si la función no es virtual, se llama *redefinición*. Tal vez a usted como programador esto le parezca una distinción sin importancia, pero el compilador trata los dos casos de manera distinta.

Polimorfismo

El término **polimorfismo** se refiere a la habilidad de asociar varios significados a un nombre de función por medio de la vinculación postergada. Por ende, el polimorfismo, la vinculación postergada y las funciones virtuales en realidad son parte del mismo tema.

Ejercicios de AUTOEVALUACIÓN

11. Suponga que debe modificar las definiciones de la clase `Venta` (cuadro 16.8) y para ello elimina la palabra reservada `virtual`. ¿Cómo cambiaría eso la salida producida por el programa en el cuadro 16.11?

Funciones virtuales y compatibilidad extendida de tipos

Analizaremos algunas consecuencias adicionales de declarar una función miembro de una clase como `virtual`; para ello realizaremos un ejemplo que utilice algunas de esas características.

C++ es un lenguaje estricto repecto a los tipos. Esto significa que siempre se comprueban los tipos de los elementos y se despliega un mensaje de error si hay un conflicto de tipos, como el caso entre un argumento y un parámetro formal en el que no hay conversión que pueda invocarse en forma automática. Esto también significa que, por lo general, el valor que se asigna a una variable debe concordar con el tipo de esa variable, aunque en algunos casos bien definidos C++ realizará una conversión explícita de tipos (llamada *coerción*), lo que da la apariencia de que puede asignar un valor de un tipo a una variable de otro tipo.

Por ejemplo, C++ le permite asignar un valor de tipo `char` o `int` a una variable de tipo `double`. No obstante, C++ no le permite asignar un valor de tipo `double` o `float` a una variable de cualquier tipo entero (`char`, `short`, `int`, `long`).

Sin embargo aunque los tipos estrictos son importantes, esta verificación estricta de tipos interfiere con la misma idea de la herencia en la programación orientada a objetos. Suponga que tiene definidas las clases `A` y `B`, y que ha definido objetos de tipo clase `A`

y clase B. No siempre podrá realizar asignaciones entre objetos de esos tipos. Por ejemplo, supongamos que un programa o unidad contiene las siguientes declaraciones de tipos:

```
class Mascota
{
public:
    virtual void imprime( );
    string nombre;
};

class Perro : public Mascota
{
public:
    virtual void imprime(); //La palabra clave virtual no se
            //necesita, pero se incluye aquí por claridad.
            //(También es un buen estilo.)

    string raza;
};

Perro perrov;
Mascota mascotav;
```

Ahora concéntrese en los miembros de datos, nombre y raza. (Para que este ejemplo sea sencillo, hemos hecho las variables miembro *public*. En una aplicación real, deberían ser *private* y contar con funciones para su manipulación.)

Cualquier cosa que sea un Perro también es una Mascota. Tal vez tendría sentido permitir que los programas consideraran los valores de tipo Perro como si fueran también valores de tipo Mascota, y por lo tanto debería permitirse lo siguiente:

```
perrov.nombre = "Chiquillo";
perrov.raza = "Gran Danes";
mascotav = perrov;
```

C++ permite este tipo de asignación. Puede asignar un valor, como el valor de perrov, a una variable de un tipo padre tal como mascotav, pero no puede realizar la asignación inversa. Aunque la asignación anterior está permitida, el valor que se asigna a la variable mascotav pierde su campo raza. A esto se le llama el **problema de la pérdida de datos**. El siguiente intento de acceso producirá un mensaje de error:

problema de la pérdida de datos

```
cout << mascotav.raza;
        //No permitido: la clase Mascota no tiene un miembro
        //llamado raza
```

Puede argumentar que esto tiene sentido, ya que una vez que un Perro se mueve hacia una variable de tipo Mascota, debería tratarse como cualquier otra Mascota y no tener propiedades peculiares para los objetos Perro. Esto da cabida a un debate muy filosófico, pero por lo general sólo resulta ser una molestia al programar. El perro llamado Chiquillo sigue siendo un Gran Danés y nos gustaría hacer referencia a su raza, incluso aunque lo tratáramos como Mascota en algunas ocasiones.

Por fortuna, C++ nos ofrece una manera de tratar a un Perro como una Mascota sin necesidad de descartar el nombre de la raza. Para ello utilizaremos apuntadores a instancias de objetos dinámicos.

Suponga que agregamos las siguientes declaraciones:

```
Mascota *pmascota;
Perro   *pperro;
```

Si utilizamos apuntadores y variables dinámicas, podemos tratar a `Chiquillo` como una `Mascota` sin perder su raza. Lo siguiente está permitido:

```
pperro = new Perro;
pperro->nombre = "Chiquillo";
pperro->raza = "Gran Danes";
pmascota = pperro;
```

Lo que es más, aún podemos tener acceso al campo `raza` del nodo al que apunta `pmascota`. Suponga que

```
Perro::imprime();
```

se define de la siguiente manera:

```
//usa iostream
void Perro::imprime()
{
    cout << "nombre: " << nombre << endl;
    cout << "raza: " << raza << endl;
}
```

La instrucción

```
pmascota->imprime();
```

hará que se imprima lo siguiente en la pantalla:

```
nombre: Chiquillo
raza: Gran Danes
```

Esto debido a que `imprime()` es una función miembro *`virtual`*. En el cuadro 16.12 hemos incluido el código de prueba.

RIESGO *El problema de la pérdida de datos*

Aunque es válido asignar un objeto de una clase derivada en una variable de una clase base, si se asigna un objeto de la clase derivada a un objeto de la clase base se perderán datos. Cualquier miembro de datos en el objeto de la clase derivada que no se encuentre también en la clase base se perderá con la asignación, y cualquier función miembro que no esté definida en la clase base tampoco estará disponible para el objeto resultante de la clase base.

Si hacemos las siguientes declaraciones y asignaciones:

```
Perro perrov;
Mascota mascotav;
perrov.nombre = “Chiquillo”
perrov.raza = “Gran Danes”
mascotav = perrov;
```

entonces el objeto `mascotav` no podrá llamar a una función miembro que se introduzca en `Perro` y se perderá el dato miembro `Perro::raza`.

CUADRO 16.12 *Más herencia con funciones virtuales (parte 1 de 2)*

```
//Programa que ilustra el uso de una función virtual
//para vencer el problema de la pérdida de datos.

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

class Mascota
{
public:
     virtual void imprime();
     string nombre;
};

class Perro : public Mascota
{
public:
     virtual void imprime(); //no se necesita la palabra clave virtual, pero se
                             //incluye por claridad. (Además es buen estilo!)
     string raza;
};

int main()
{
     Perro perrov;
     Mascota mascotav;

     perrov.nombre = "Chiquillo";
     perrov.raza = "Gran Danes";
     mascotav = perrov;

     //mascotav.raza; es ilegal ya que la clase Mascota no tiene un miembro llamado raza

     Perro *pperro;
     pperro = new Perro;
```

CUADRO 16.12 *Más herencia con funciones virtuales (parte 2 de 2)*

```
    pperro->nombre = "Chiquillo";
    pperro->raza = "Gran Danes";

    Mascota *pmascota;
    pmascota = pperro;
    pmascota->imprime(); // Estas dos instrucciones imprimen lo mismo:
    pperro->imprime(); // nombre: Chiquillo raza: Gran Danes

    //Lo siguiente, que accede a las variables miembro en forma directa
    //en vez de hacerlo mediante funciones virtuales, produciría un error:
    //cout << "nombre: " << pmascota->nombre << "  raza: "
    //<< pmascota->raza << endl;
    //genera un mensaje de error: 'class Mascota' has no member
    //named 'raza' .
    //Consulte la sección de Riesgo titulada "No utilizar funciones miembro virtuales"
    //para más detalles sobre esto.

    return 0;
}

void Perro::imprime()
{
    cout << "nombre: " << nombre << endl;
    cout << "raza: " << raza << endl;
}

void Mascota::imprime()
{
    cout << "nombre: " << endl;//Nota no se menciona la raza
}
```

Diálogo de ejemplo

```
nombre: Chiquillo
raza: Gran Danes
nombre: Chiquillo
raza: Gran Danes
```

RIESGO *No utilizar funciones miembro virtuales*

Para poder obtener el beneficio de la compatibilidad extendida de tipos que vimos antes, debe usar funciones miembro *virtual*. Por ejemplo, suponga que no hubiéramos utilizado funciones miembro en el cuadro 16.12. Suponga que en lugar de

```
pmascota->imprime();
```

utilizáramos lo siguiente:

```
cout << "nombre: " << pmascota->nombre
     << "raza; " << pmascota->raza << endl;
```

Este código hubiera desencadenado un mensaje de error. La razón de ello es que el tipo de la expresión

```
*pmascota
```

se determina en base al tipo apuntador de `pmascota`. Es un tipo apuntador para el tipo `Mascota`, y el tipo `Mascota` no tiene un campo llamado `raza`.

Pero `imprime()` se declaró *virtual* en la clase base `Mascota`. Por ello, cuando el compilador ve la llamada

```
pmascota->imprime();
```

verifica la tabla *virtual* para las clases `Mascota` y `Perro`, y ve que `pmascota` apunta a un objeto de tipo `Perro`. Por lo tanto, utiliza el código generado para

```
Perro::imprime()
```

en vez de usar el código para

```
Mascota::imprime().
```

La programación orientada a objetos con variables dinámicas es una manera muy distinta de ver la programación. Al principio esto puede ser desconcertante. Las siguientes dos reglas le serán de mucha utilidad:

1. Si el tipo de dominio del apuntador `p_ancestro` es una clase base para el tipo de dominio del apuntador `p_descendiente`, entonces la siguiente asignación de apuntadores está permitida:

   ```
   p_ancestro = p_descendiente;
   ```

   Lo que es más, no se perderá ninguno de los miembros de datos o las funciones miembro de la variable dinámica a la que apunte `p_descendiente`.

2. Aunque todos los campos adicionales de la variable dinámica estén ahí, necesitará funciones miembro *virtual* para acceder a ellos.

RIESGO *Tratar de compilar definiciones de clases sin definiciones para todas las funciones miembro virtuales*

Es prudente escribir los programas en forma de incrementos. Esto significa que debe codificar un poco, luego probar un poco, después codificar un poco más y probar un poco más, y así sucesivamente. No obstante, si trata de compilar clases con funciones miembro `virtual` pero no implementa a cada uno de los miembros, podría encontrarse con algunos mensajes muy difíciles de entender, aún si no llama a las funciones miembro indefinidas.

Si cualquier función miembro virtual no se implementa antes de la compilación, este proceso fallará con mensajes de error como "referencia indefinida para la tabla virtual de *Nombre_clase*". Inclusive si *no hay clase derivada* y *sólo hay un* miembro `virtual`, este tipo de mensaje podrá seguir apareciendo si esa función no tiene una definición.

Lo que hace que los mensajes de error sean tan difíciles de descifrar es que sin definiciones para las funciones que se declaran como `virtual`, puede haber más mensajes de error que se quejen sobre una referencia indefinida a constructores predeterminados, aún si estos constructores en realidad sí están definidos.

TIP DE PROGRAMACIÓN

Haga los destructores virtuales

Es buena política siempre hacer que los destructores sean `virtual`, pero antes de explicar por qué necesitamos decir una palabra o dos acerca de cómo interactúan los destructores y los apuntadores y qué es lo que significa para un destructor ser virtual.

Considere el siguiente código, en donde `UnaClase` es una clase con un destructor que no es virtual:

```
UnaClase *p = new UnaClase;
   . . .
delete p;
```

Cuando se invoca `delete` con `p`, el destructor de la clase `UnaClase` se invoca de manera automática. Ahora veamos lo que ocurre cuando un destructor se marca como `virtual`.

La manera más sencilla de describir cómo interactúan los destructores con el mecanismo de las funciones virtuales es que los destructores se tratan como si todos tuvieran el mismo nombre (aún y cuando en realidad no es así). Por ejemplo, suponga que `Derivada` es una clase derivada de la clase `Base` y suponga que el destructor en la clase `Base` se marca como `virtual`. Ahora considere el siguiente código:

```
Base *pBase = new Derivada;
     . . .
delete pBase;
```

Cuando se invoca `delete` con `pBase`, se llama a un destructor. Como el destructor en la clase `Base` se marcó `virtual` y el objeto al que apunta es de tipo `Derivada`, se hace una llamada al destructor para la clase `Derivada` (que a su vez llama al destructor para

la clase Base.) Si el destructor en la clase Base no se hubiera declarado como *virtual*, entonces sólo debería llamarse al destructor de la clase Base.

Otro punto a tener en cuenta es que cuando un destructor se marca como *virtual*, entonces todos los destructores de las clases derivadas son virtuales de manera automática (se hayan marcado o no como *virtual*). De nuevo, este comportamiento es como si todos los destructores tuvieran el mismo nombre (aun cuando no es así).

Ahora estamos listos para explicar por qué todos los destructores deben ser virtuales. Suponga que la clase Base tiene una variable miembro llamada pB de tipo apuntador, que el constructor para la clase Base crea una variable dinámica a la que apunta pB y que el destructor para la clase Base elimina a la variable dinámica a la que apunta pB. Suponga también que el destructor para la clase Base *no* se marca como *virtual*, que la clase Derivada (que se deriva de Base) tiene una variable miembro llamada pD de tipo apuntador, que el constructor para la clase Derivada crea una variable miembro a la que apunta pD y que el destructor para la clase Derivada elimina a la variable dinámica a la que apunta pD. Considere el siguiente código:

```
Base *pBase = new Derivada;
     . . .
delete pBase;
```

Como el destructor en la clase Base no se marca como *virtual*, sólo se invocará a este destructor. Éste devolverá al almacén de memoria libre la memoria utilizada por la variable dinámica a la que apunta pB, pero la memoria para la variable dinámica a la que apunta pD nunca regresará al almacén de memoria libre (hasta que termine el programa).

Por otro lado, si el destructor para la clase Base se marcara como *virtual*, entonces cuando se aplicara *delete* a pBase se invocaría el destructor para la clase Derivada (ya que el objeto al que apunta es de tipo Derivada). El destructor para la clase Derivada eliminaría la variable dinámica a la que apunta pD y después invocaría en forma automática al destructor para la clase Base; eso eliminaría la variable dinámica a la que apunta pB. Por lo tanto, cuando el destructor de la clase base se marca como *virtual*, toda la memoria se devuelve al almacén de memoria libre. Para prepararnos en caso de eventualidades como éstas, es conveniente siempre marcar los destructores como virtuales.

Ejercicios de AUTOEVALUACIÓN

12. ¿Por qué no podemos asignar un objeto de una clase base a una variable de una clase derivada?

13. ¿Cuál es el problema con la asignación (válida) de un objeto de una clase derivada a una variable de una clase base?

14. Suponga que la clase base y la clase derivada tienen una función miembro con la misma firma. Si tiene un apuntador a un objeto de la clase base y llama al miembro de una función a través del apuntador, hable sobre lo que determina cuál es la función que se llama en realidad: la función miembro de la clase base o la función de la clase derivada.

Resumen del capítulo

- La herencia proporciona una herramienta para reutilizar código al hacer que una clase se derive de otra; además le agrega características a la clase derivada.
- Los objetos de una clase derivada heredan todos los miembros de la clase base y también pueden agregar miembros.
- Vinculación postergada significa que la decisión de cuál versión de una función miembro es la apropiada se toma en tiempo de ejecución. Las funciones virtuales son lo que C++ utiliza para lograr la vinculación postergada. El polimorfismo, la vinculación postergada y las funciones virtuales son en realidad el mismo tema.
- Un miembro *protected* en la clase base está disponible de manera directa para las funciones miembro de una clase pública derivada.

Respuestas a los ejercicios de autoevaluación

1. Sí. Puede utilizar un objeto de una clase derivada como parámetro del tipo de la clase base. Un `EmpleadoPorHoras` es un `Empleado`. Un `EmpleadoAsalariado` es un `Empleado`.

2. ```
   class TipoListo : public Listo
   {
   public:
       TipoListo( );
       TipoListo(int nuevo_a, int nuevo_b, bool nuevo_loco);
       bool esta_loco( ) const;
   private:
       bool loco;
   };
   ```

3. Es válida debido a que `a` y `b` se marcan como *protected* en la clase base `Listo` y por lo tanto pueden utilizarse por su nombre en una clase derivada. Si `a` y `b` se hubieran marcado como *private*, entonces esto sería inválido.

4. La declaración de la función `obtiene_nombre` no se proporciona en la definición de `EmpleadoAsalariado` debido a que no se redefine en esta clase. Se hereda sin cambios de la clase base `Empleado`.

5. ```
   #include<iostream>
   #include "empleadoasalariado.h"
   using namespace std;
   namespace empleadosavitch
   {
        class EmpleadoTitulado : public EmpleadoAsalariado
        {
        public:
            EmpleadoTitulado( );
            EmpleadoTitulado(string el_nombre, string el_titulo,
                             string_el_nss, double el _salario);
            string obtiene_titulo( ) const;
            void asigna_titulo(string el_titulo);
            void asigna_nombre(string el_nombre);
   ```

```
    private:
        string titulo;
    };
}//empleadosavitch
```

6.
```
namespace empleadosavitch
{
    EmpleadoTitulado::EmpleadoTitulado( )
           : EmpleadoAsalariado( ), titulo("Sin titulo todavia")
    {
         //vacío de manera intencional
    }

    EmpleadoTitulado::EmpleadoTitulado(string el_nombre,
                            string el_titulo,
                            string el_nss, double el_salario)
            : EmpleadoAsalariado(el_nombre, el_nss, el_salario),
                            titulo(el_titulo)
    {
           //vacío de manera intencional
    }

    void EmpleadoTitulado::asigna_nombre(string el_nombre)
    {
        Empleado::asigna_nombre(titulo + el_nombre);
    }
}//empleadosavitch
```

7. No. Si no define un operador de asignación sobrecargado o un constructor de copia para una clase derivada, entonces se definirá un operador de asignación predeterminado y un constructor de copia predeterminado para la clase derivada. No obstante, si la clase involucra el uso de apuntadores, arreglos dinámicos o cualquier otro tipo de datos dinámicos, es casi seguro que ni el operador de asignación predeterminado, ni el constructor de copia predeterminado se comportarán como queremos que lo hagan.

8. Los constructores se llaman en el siguiente orden: primero `Padre`, después `Hijo` y por último `Nieto`. Los destructores se llaman en el orden inverso: primero `Nieto`, después `Hijo` y por último `Padre`.

9.
```
//Usa iostream y cstdlib:
void ArregloLlenoParcial::agrega_valor(double nueva_entrada)
{
     if (numero_usado == numero_maximo)
     {
         cout << "Intento de agregar en un arreglo lleno.\n";
         exit(1);
     }
    else
    {
       a[numero_usado] = nueva_entrada;
       numero_usado++;
    }
}

ArregloLlenoParcial::ArregloLlenoParcial
            (const ArregloLlenoParcial& objeto)
                 : numero_maximo(objeto.numero_maximo),
```

```
                          numero_usado(objeto.numero_usado)
{
   a = new double[numero_maximo];

   for (int i = 0; i < numero_usado; i++)
       a[i] = objeto.a[i]
}
void ArregloLlenoParcial::operator =
                 (const ArregloLlenoParcial& lado_derecho)
{
     if (lado_derecho.numero_maximo > numero_maximo)
     {
         delete [] a;
         numero_maximo = lado_derecho.numero_maximo;
         a = new double[numero_maximo];
     }
     numero_usado = lado_derecho.numero_usado;

     for (int i = 0; i < numero_usado; i++)
         a[i] = lado_derecho.a[i];
}

ArregloLlenoParcial::~ArregloLlenoParcial()
{
       delete [] a;
}
```

10.
```
class ArregloLlenoParcialCMax : public ArregloLlenoParcial
{
public:
    ArregloLlenoParcialCMax(int tamanio_arreglo);
    ArregloLlenoParcialCMax(const ArregloLlenoParcialCMax& objeto);
    ~ArregloLlenoParcialCMax();
    void operator= (const ArregloLlenoParcialCMax& lado_derecho);
    void agrega_valor(double nueva_entrada);
    double obtiene_max();
private:
    double valor_maximo;
};

ArregloLlenoParcialCMax::ArregloLlenoParcialCMax(int tamanio_arreglo)
                                :ArregloLlenoParcial(tamanio_arreglo)
{
      //Cuerpo vacío de manera intencional.
      //valor_maximo sin inicializar ya que
      //no hay un valor predeterminado adecuado.
}

/*
Tenga en cuenta que lo siguiente no funciones, ya que llama al constructor
predeterminado para ArregloParcialLleno pero esta clase no tiene un
constructor predeterminado:
ArregloLlenoParcialCMax::ArregloLlenoParcialCMax (int tamanio_arreglo)
                     : numero_maximo(tamanio_arreglo), numero_usado(0)
```

```
{
     a = new double[numero_maximo];
}
*/

ArregloLlenoParcialCMax::ArregloLlenoParcialCMax
                  (const ArregloLlenoParcialCMax& objeto)
                      :ArregloLlenoParcial(objeto)
{
     if (objeto.numero_usado > 0)
     {
          valor_maximo = a[0];

          for (int i = 1; i < numero_usado; i++)
              if (a[i] > valor_maximo)
                  valor_maximo = a[i];
     }//de lo contrario, dejar a valor_max sin inicializar
}

//Esto es equivalente al constructor predeterminado que proporciona
//C++, por lo que puede omitir esta definición.
//Pero si la omite, también deberá omitir la declaración del
//destructor de la definición de la clase.
ArregloLlenoParcialCMax::~ArregloLlenoParcialCMax( )
{
      //Vacío de manera intencional.
}

void ArregloLlenoParcialCMax::operator =
                       (const ArregloLlenoParcialCMax& lado_derecho)
{
     ArregloLlenoParcial::operator = (lado_derecho);
     valor_maximo = lado_derecho.valor_maximo;
}

//Usa iostream y cstdlib;
void ArregloLlenoParcialCMax::agrega_valor(double nueva_entrada)
{
    if (numero_usado == numero_maximo)
    {
        cout << "Intento de agregar en un arreglo lleno.\n";
        exit(1);
}

     if ((numero_usado == 0) || (nueva_entrada >  valor_maximo))
         valor_maximo = nueva_entrada;

     a[numero_usado] = nueva_entrada;
     numero_usado++;
}

double ArregloLlenoParcialCMax::obtiene_max( )
{
     return valor_maximo;
}
```

11. La salida cambiaría a

```
El articulo con descuento no es mas barato.
```

12. No habría miembro para asignar a los miembros agregados de la clase derivada.

13. Aunque es válido asignar un objeto de una clase derivada a una variable de una clase base, se descartan las partes del objeto de la clase derivada que no son miembros de la clase base. Esta situación se conoce como el *problema de la pérdida de datos*.

14. Si la función de la clase base lleva el modificador `virtual`, entonces el tipo del objeto con el que se inicializó el apuntador es el que determina cuál función miembro se va a llamar. Si la función miembro de la clase base no tiene el modificador `virtual`, entonces el tipo del apuntador es el que determina cuál función miembro se va a llamar.

Proyectos de programación

1. Escriba un programa que utilice la clase `EmpleadoAsalariado` del cuadro 16.4. Su programa debe definir una clase llamada `Administrador`, que debe derivarse de la clase `EmpleadoAsalariado`. Puede cambiar todo lo que sea `private` en la clase base a `protected`. Debe suministrar los siguientes datos y miembros de función adicionales:

   Una variable miembro de tipo `string` que contenga el título del administrador (como Director o Vicepresidente).

   Una variable miembro de tipo `string` que contenga el área de responsabilidad de la compañía (tal como Producción, Contabilidad o Personal).

   Una variable miembro de tipo `string` que contenga el nombre del supervisor inmediato de este administrador.

   Una variable miembro `protected:` de tipo `double` que almacene el salario anual del administrador. Usted puede utilizar el miembro salario existente si hizo el cambio que recomendamos antes.

   Una función miembro llamada `asigna_supervisor`, que modifique el nombre del supervisor.

   Una función miembro para leer los datos de un administrador desde el teclado.

   Una función miembro llamada `imprime`, que muestre los datos del objeto en la pantalla.

   Una sobrecarga de la función miembro `imprime_cheque( )` con las anotaciones apropiadas en el cheque.

2. Agregue las clasificaciones temporal, administrativo, permanente y demás clasificaciones de empleado a la jerarquía de los cuadros 16.1, 16.3 y 16.4. Implemente y pruebe esta jerarquía. Pruebe todas las funciones miembro. Una interfaz de usuario con un menú sería un toque agradable para su programa de prueba.

3. Proporcione la definición de una clase llamada `Doctor`, cuyos objetos sean registros para los doctores de una clínica. Esta clase será una clase derivada de la clase `EmpleadoAsalariado` que aparece en el cuadro 16.4. Un registro de la clase `Doctor` tiene la especialidad del doctor (tal como "Pediatra", "Obstetra", "Medico General", etcétera, por lo que debe usar el tipo `string`) y el costo de la consulta (use el tipo `double`). Asegúrese de que su clase tenga un complemento razonable de constructores, funciones miembro de acceso y de mutación, un operador de asignación sobrecargado y un constructor de copia. Escriba un programa controlador para probar todas sus funciones.

4. Cree una clase base llamada `Vehiculo` que tenga el nombre del fabricante (tipo `string`), el número de cilindros en el motor (tipo `int`) y el propietario (tipo `Persona`, que se muestra a continuación). Después cree una clase llamada `Camion` que se derive de `Vehiculo` y tenga propiedades adicionales: la capacidad de carga en toneladas (tipo `double`, ya que puede contener una parte fraccionaria) y la capacidad de remolque en kilos (tipo `int`). Asegúrese de que sus clases tengan un complemento razonable de constructores, funciones miembro de acceso y de mutación, un operador de asignación sobrecargado y un constructor de copia. Escriba un programa controlador para probar todas sus funciones.

A continuación se muestra la definición de la clase `Persona`. La implementación de la clase es parte de este Proyecto de programación.

```
class Persona
{
public:
    Persona();
    Persona(string el_nombre);
    Persona(const Persona& el_objeto);
    string obtiene_nombre() const;
    Persona& operator=(const Persona& lado_der);
    friend istream& operator >>(istream& flujo_ent,
                                Persona& objeto_persona);
    friend ostream& operator <<(ostream& flujo_sal,
                           const Persona& objeto_persona);
    private:
        string nombre;
};
```

5. Defina una clase llamada `Auto` que se derive de la clase `Vehiculo` que aparece en el Proyecto de programación 4. Defina una clase llamada `AutoDeportivo` que se derive de la clase `Auto`. Tenga creatividad al elegir las variables y las funciones miembro. Escriba un programa controlador para probar las clases `Auto` y `AutoDeportivo`.

6. Proporcione la definición de dos clases, `Paciente` y `Facturacion`, cuyos objetos sean registros para una clínica. `Paciente` se derivará de la clase `Persona` que aparece en el Proyecto de programación 4. Un registro de la clase `Paciente` debe tener el nombre del paciente (heredado de la clase `Persona`) y el doctor principal, de tipo `Doctor` que se definió en el Proyecto de programación 3. Un objeto `Facturacion` contendrá un objeto `Paciente`, un objeto `Doctor` y un monto a pagar de tipo *`double`*. Asegúrese de que sus clases tengan un complemento razonable de constructores, funciones miembro de acceso y de mutación, un operador de asignación sobrecargado y un constructor de copia. Primero escriba un programa controlador para probar todas sus funciones miembro y después escriba un programa de prueba para crear por lo menos dos pacientes, dos doctores y dos registros de `Facturacion`, para que después imprima el ingreso total de los registros de `Facturacion`.

7. Considere un sistema de gráficos que tiene clases para diversas figuras: rectángulos, cuadrados, triángulos, círculos, y así sucesivamente. Por ejemplo, un rectángulo podría tener miembros de datos para la altura, la anchura y el punto central, mientras que un cuadrado y un círculo podrían tener sólo un punto central y una longitud de borde o radio, en forma respectiva. En un sistema bien diseñado, todo esto se derivaría de una clase común llamada `Figura`. Usted deberá implementar tal sistema.

   La clase `Figura` es la clase base. Debe agregar sólo las clases `Rectangulo` y `Triangulo` que se deriven de `Figura`. Cada clase deberá tener las funciones miembro `borra` y `dibuja` ficticias. Cada una de estas funciones miembro deberá mostrar un mensaje que indique qué función se ha llamado y cuál es la clase del objeto que hizo la llamada. Como sólo son funciones ficticias, no harán nada más que mostrar este mensaje. La función miembro `centro` llamará a las funciones `borra` y `dibuja` para borrar y volver a dibujar la figura en el centro. Como sólo tiene funciones ficticias para `borra` y `dibuja`, `centro` no hará ningún "centrado", sino que llamará a las funciones miembro `borra` y `dibuja`. Agregue también un mensaje de salida en la función miembro `centro` que anuncie que se está haciendo una llamada a esta función. Las funciones miembro no deberán tomar argumentos.

   Este proyecto consta de tres partes:

   a) Escriba las definiciones de las clases sin usar funciones virtuales. Compile y pruebe.

   b) Haga que las funciones miembro de la clase base sean virtuales. Compile y pruebe.

   c) Explique la diferencia en resultados.

Para un ejemplo real, tendría que sustituir la definición de cada una de estas funciones miembro con código para realizar los dibujos de verdad. En el Proyecto de programación 8 le pediremos que haga esto.

Use la siguiente función `main` para todas las pruebas:

```
//Programa de prueba para el Proyecto de programación 6.
#include <iostream>
#include "figura.h"
#include "rectangulo.h"
#include "triangulo.h"
using std::cout;

int main( )
{
    Triangulo tri;
    tri.dibuja( );
    cout <<
      "\nObjeto de clase derivada Triangulo llamando a centro( ).\n";
    tri.centro( ); //Llama a dibuja y centra

    Rectangulo rect;
    rect.dibuja( );
    cout <<
      "\nObjeto de clase derivada Rectangulo llamando a centro( ).\n";
    tri.centro( ); //Llama a dibuja y centra
    return 0;
}
```

8. Complete el Proyecto de programación 7. Proporcione nuevas definiciones para los diversos constructores y las funciones miembro `Figura::centro`, `Figura::dibuja`, `Figura::borra`, `Triangulo::dibuja`, `Triangulo::borra`, `Rectangulo::dibuja` y `Rectangulo::borra` de manera que las funciones `dibuja` en realidad dibujen figuras en la pantalla, colocando el carácter `'*'` en las posiciones apropiadas. Para las funciones `borra` sólo necesita limpiar la pantalla (muestre líneas en blanco o haga algo más sofisticado). Hay muchos detalles en este problema y tendrá que tomar decisiones sobre algunos de ellos por su cuenta.

9. Los bancos tienen muchos tipos distintos de cuentas, a menudo con distintas reglas para las cuotas asociadas con las transacciones tales como los retiros. Los clientes pueden transferir fondos entre cuentas, para lo cual deberán cubrir las cuotas apropiadas relacionadas con el retiro de fondos de una cuenta.

Escriba un programa con una clase base para una cuenta bancaria y dos clases derivadas (como se describe a continuación) que representen cuentas con distintas reglas para retirar fondos. Escriba además una función que transfiera fondos de una cuenta (de cualquier tipo) a otra. Una transferencia es un retiro de una cuenta y un depósito en la otra. Como la transferencia puede hacerse en cualquier momento y con cualquier tipo de cuenta, la función de retiro en las clases deberá ser virtual. Escriba un programa principal que cree tres cuentas (una de cada clase) y que pruebe la función de transferencia.

Para las clases, cree una clase base llamada `CuentaBancaria`, que tenga el nombre del propietario de la cuenta (un valor string) y el balance de la misma (*`double`*) como miembros de datos. Incluya las funciones miembro `deposita` y `retira` (cada una con un valor *`double`* para el monto como argumento), además de las funciones de acceso `obtieneNombre` y `obtieneBalance`. La función `deposita` sumará el monto al balance (suponiendo que la cantidad no sea negativa) y la función `retira` restará el monto del balance (suponiendo que el monto sea positivo y menor o igual que el balance). Cree además una clase llamada `CuentaMercadoMoneda` que se derive de `CuentaBancaria`. En una `CuentaMercadoMoneda` el usuario recibe 2 retiros gratis en un periodo de tiempo dado (no se preocupe por el tiempo en este problema). Una

vez que se hayan utilizado los retiros gratuitos, se deducirá una cuota de $1.50 del balance por cada retiro adicional. Por ende, la clase deberá tener un miembro de datos para llevar la cuenta del número de retiros. También deberá sobreescribir la función `retira`. Por último, cree una clase llamada `CuentaCD` (para modelar un Certificado de depósito) que se derive de `CuentaBancaria` y que además de tener el nombre y el balance también tenga una tasa de interés. Si se retiran los fondos antes de tiempo en un CD, se incurrirá en penalizaciones. Suponga que un retiro de fondos (cualquier monto) incurre en una penalización del 25% del interés anual que se obtenga de la cuenta. Suponga que se deducirá el monto retirado más la penalización del balance de la cuenta. De nuevo, la función retira deberá sobrescribir a la de la clase base. Para las tres clases, la función retira deberá devolver un entero que indique el estado (ya sea una aceptación o que haya fondos insuficientes para que se realice el retiro). Para los fines de este ejercicio, no se preocupe por las demás funciones y propiedades de estas cuentas (como la manera en que se paga el interés y cuándo se efectúa el pago).

17 Manejo de excepciones

17 Manejo de excepciones

La excepción confirma la regla.

FRASE POPULAR

Introducción

Una manera de escribir un programa es primero suponer que no ocurrirá nada inusual o incorrecto. Por ejemplo, si el programa toma una entrada de una lista, podríamos suponer que la lista no está vacía. Una vez que el programa esté funcionando para la situación básica en la que todo siempre sale bien, podemos entonces agregar código para hacernos cargo de los casos excepcionales. En C++ hay una forma de reflejar esta metodología en nuestro código. En esencia, necesitamos escribir nuestro código como si nunca ocurriera algo inusual. Después de eso podemos utilizar las herramientas para el manejo de excepciones de C++ para agregar código para esos casos inusuales.

Por lo general, el manejo de excepciones se utiliza para tratar con situaciones de error, pero tal vez una mejor forma de ver una excepción sea como una manera de tratar con las "situaciones excepcionales". Después de todo, si su código maneja un "error" de una forma correcta, entonces ya no es un error.

Tal vez el uso más importante de las excepciones sea el tratar con funciones que tienen cierto caso especial que se maneja en forma distinta, dependiendo de cómo se utilice la función. Tal vez la función se utilice en muchos programas, algunos de los cuales manejarán el caso especial de cierta forma y otros lo manejarán de otra manera distinta. Por ejemplo, si hay una división entre cero en la función, podría resultar que para ciertas invocaciones de la función el programa debería terminar, pero para otras invocaciones de la función debería ocurrir algo más. Pronto veremos que una función así puede definirse para lanzar una excepción si ocurre el caso especial, y que esa excepción permitirá manejar el caso especial fuera de la función. De esa manera, el caso especial podrá manejarse en distintas formas para distintas invocaciones de la función.

En C++, el manejo de excepciones funciona así: cierto software de biblioteca o cierto código suyo debe proporcionar un mecanismo que nos indique que ocurrió algo inusual. A esto se le conoce como *lanzar una excepción*. En otro lugar de su programa debe colocar el código que trate con ese caso excepcional. A esto se le conoce como *manejar la excepción*. Este método de programación hace que nuestro código esté mejor organizado. Desde luego que aún tenemos que explicar los detalles de cómo puede hacer esto en C++.

Prerrequisitos

Con la excepción de una subsección que se puede omitir, en la sección 17.1 sólo utilizamos material de los capítulos 2 al 8. En la subsección Riesgo de la sección 17.1 titulada "Especificación de una excepción en clases derivadas" utilizamos material del capítulo 16. Esta subsección Riesgo se puede omitir sin perder continuidad.

Con excepción de una subsección que se puede omitir, en la sección 17.2 sólo utilizamos material de los capítulos 2 al 11 y de la sección 16.1 del capítulo 16, además de la sección 17.1. La subsección de la sección 17.2 titulada "Prueba para la memoria disponible" utilizamos material del capítulo 15. Puede omitir esta subsección sin que haya pérdida de continuidad.

17.1 Fundamentos del manejo de excepciones

Bueno, el programa funciona para la mayoría de los casos. No sabía que tenía que trabajar para ese caso.

Estudiante de ciencias computacionales, en su apelación por una calificación.

El manejo de excepciones debe utilizarse con moderación y en situaciones que sean más complicadas de lo que sea razonable incluir en un ejemplo introductorio. Por esta razón, le enseñaremos los detalles sobre el manejo de excepciones en C++ mediante ejemplos simples en los que por lo general no se utilizaría el manejo de excepciones. Esto tiene mucho sentido cuando se desea aprender sobre el manejo de excepciones, pero no olvide que estos primeros ejemplos son pequeños y, en la práctica, no se utilizaría el manejo de excepciones para algo tan simple.

Un pequeño ejemplo sobre el manejo de excepciones

Para este ejemplo supondremos que la leche es un alimento tan importante en nuestra cultura, que casi nunca se agota, pero aun así nos gustaría que nuestros programas se adaptaran a la muy poco probable situación de que se acabara la leche. El código básico, en el que se supone que no se agotará la leche, podría ser el siguiente:

```
cout << "Escriba el numero de donas:\n";
cin >> donas;
cout << "Escriba el numero de vasos de leche:\n";
cin >> leche;
dpv = donas/static_cast<double>(leche);
cout << donas << " donas.\n"
     << leche << " vasos de leche.\n"
     << "Usted tiene " << dpv
     << " donas por cada vaso de leche.\n";
```

Si no hubiera leche, entonces este código incluiría una división entre cero, lo cual es un error. Para tratar con la situación especial en la que se agota la leche, podemos agregar una prueba para esta situación inusual. En el cuadro 17.1 se muestra el programa completo que contiene esta prueba para la situación especial; este programa no utiliza el manejo de excepciones. Ahora, veamos cómo podemos reescribir dicho programa mediante el uso de las herramientas de manejo de excepciones en C++.

CUADRO 17.1 *Manejo de un caso especial sin el manejo de excepciones*

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

int main()
{
    int donas, leche;
    double dpv;
    cout << "Escriba el numero de donas:\n";
    cin >> donas;
    cout << "Escriba el numero de vasos de leche:\n";
    cin >> leche;

    if (leche <= 0)
    {
        cout << donas << " donas, sin leche!\n"
             << "Vaya a comprar leche.\n";
    }
    else
    {
        dpv = donas/static_cast<double>(leche);
        cout << donas << " donas.\n"
             << leche << " vasos de leche.\n"
             << "Usted tiene " << dpv
             << " donas para cada vaso de leche.\n";
    }

    cout << "Fin del programa.\n";
    return 0;
}
```

Diálogo de ejemplo

```
Escriba el numero de donas:
12
Escriba el numero de vasos de leche:
0
12 donas, sin leche!
Vaya a comprar leche.
Fin del programa.
```

En el cuadro 17.2 hemos reescrito el programa del cuadro 17.1 y le agregamos el manejo de una excepción. Éste es sólo un pequeño ejemplo; en la práctica es muy probable que no se utilice una excepción en este caso, pero nos sirve como ejemplo. Aunque el programa en sí no es más simple, por lo menos la parte que va entre las palabras *try* y *catch* es más limpia, lo cual nos muestra la ventaja del uso de excepciones. Analice el código entre las palabras *try* y *catch*. Ese código es en esencia el mismo código que el del cuadro 17.1, pero en vez de la voluminosa instrucción *if-else* (del cuadro 17.1), este nuevo

CUADRO 17.2 *Mismo ejemplo pero con manejo de excepciones (parte 1 de 2)*

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

int main()
{
    int donas, leche;
    double dpv;

    try
    {
        cout << "Escriba el numero de donas:\n";
        cin >> donas;
        cout << "Escriba el numero de vasos de leche:\n";
        cin >> leche;

        if (leche <= 0)
                throw donas;

        dpv = donas/static_cast<double>(leche);
        cout << donas << " donas.\n"
             << leche << " vasos de leche.\n"
             << "Usted tiene " << dpv
             << " donas para cada vaso de leche.\n";
    }
    catch(int e)
    {
        cout << e << " donas, sin leche!\n"
             << "Vaya a comprar leche.\n";
    }

    cout << "Fin del programa.\n";
    return 0;
}
```

CUADRO 17.2 *Mismo ejemplo pero con manejo de excepciones (parte 2 de 2)*

Diálogo de ejemplo 1

```
Escriba el numero de donas:
12
Escriba el numero de vasos de leche:
6
12 donas.
6 vasos de leche.
Usted tiene 2 donas para cada vaso de leche.
Fin del programa.
```

Diálogo de ejemplo 2

```
Escriba el numero de donas:
12
Escriba el numero de vasos de leche:
0
12 donas, sin leche!
Vaya a comprar leche.
Fin del programa.
```

programa tiene la siguiente instrucción *if* más pequeña (además de algunas instrucciones simples sin bifurcación):

```
if (leche <= 0)
    throw donas;
```

Esta instrucción *if* dice que si no hay leche, entonces hay que hacer algo excepcional. Ese algo excepcional se incluye después de la palabra *catch*. La idea es que el código que va después de la palabra *try* maneje la situación normal, y que el código que va después de la palabra *catch* se utilice sólo en circunstancias excepcionales. Por lo tanto, hemos separado el caso normal del caso excepcional. En este pequeño ejemplo, la separación realmente no nos ayuda mucho, pero en otras situaciones demostrará ser muy útil. Veamos ahora los detalles.

La manera básica de manejar excepciones en C++ consiste en el trío *try-throw-catch*. Un **bloque** *try* tiene la siguiente sintaxis:

bloque *try*

```
try
{
   Algo_de_codigo
}
```

Este bloque *try* contiene el código para el algoritmo básico que indica a la computadora lo que debe hacer cuando todo sale bien. Se llama bloque *try* (probar) porque no estamos 100% seguros de que todo saldrá bien, pero queremos "probar lo que sucede".

Ahora, si algo *sale* mal debemos lanzar (throw) una excepción, que es una manera de indicar que algo salió mal. La estructura básica cuando agregamos una instrucción *throw* es la siguiente:

```
try
{
    Codigo_a_probar
    Tal_vez_lanzar_una_excepcion
    Mas_codigo
}
```

A continuación se muestra un ejemplo de un bloque *try* en el que se incluye una instrucción *throw* (se copió del cuadro 17.2):

```
try
{
     cout << "Escriba el numero de donas:\n";
     cin >> donas;
     cout << "Escriba el numero de vasos de leche:\n";
     cin >> leche;

     if (leche <= 0)
            throw donas;

     dpv = donas/static_cast<double>(leche);
     cout << donas << " donas.\n"
          << leche << " vasos de leche.\n"
          << "Usted tiene " << dpv
          << " donas para cada vaso de leche.\n";
}
```

instrucción throw

La siguiente instrucción lanza (**throw**) el valor *int* donas:

```
throw donas;
```

lanzamiento de excepciones

El valor que se lanza, en este caso **donas**, se denomina algunas veces **excepción**, y al proceso de ejecutar una instrucción *throw* se le conoce como **lanzar una excepción**. Usted puede lanzar un valor de cualquier tipo. En este caso se lanza un valor *int*.

Como el nombre sugiere, cuando algo se "lanza", se va de un lugar a otro. En C++, lo que pasa de un lugar a otro es el flujo de control (así como el valor lanzado). Cuando se lanza una excepción, el código dentro del bloque *try* circundante deja de ejecutarse y otra porción de código, conocida como **bloque** *catch*, comienza su ejecución. A este proceso de ejecutar el bloque *catch* se le conoce como atrapar o manejar la excepción. Cuando se lanza una excepción, debe haber un bloque *catch* para manejarla (atraparla). En el cuadro 17.2, el bloque *catch* apropiado va justo después del bloque *try*. A continuación repetimos ese bloque *catch*:

bloque *catch*

```
catch(int e)
{
    cout << e << " donas, sin leche!\n"
         << "Vaya a comprar leche.\n";
}
```

Instrucción `throw`

Sintaxis

`throw` *Expresion_para_el_valor_que_se_va_a_lanzar* ;

Cuando se ejecuta la instrucción `throw` se detiene la ejecución del bloque `try` circundante. Si el bloque `try` va seguido de un bloque `catch` apropiado, entonces el flujo de control se transfiere al bloque `catch`. Casi siempre una instrucción `throw` se integra en una instrucción de bifurcación tal como `if`. El valor que se lanza puede ser de cualquier tipo.

Ejemplo

```
if (leche <= 0)
    throw donas;
```

Este bloque `catch` se parece mucho a la definición de una función que tiene un parámetro de tipo `int`. No es la definición de una función pero, en cierta forma, un bloque `catch` es como una función. Es una pieza separada de código que se ejecuta cuando su programa encuentra (y ejecuta) lo siguiente (dentro del bloque `try` anterior):

```
throw Un_int;
```

Entonces, esta instrucción `throw` es similar a la llamada a una función, pero en vez de llamar a la función llama al bloque `catch` e indica que se debe ejecutar el código en el bloque `catch`. Por lo general, a un bloque `catch` se le conoce como **manejador de excepciones**, un término que sugiere que un bloque `catch` tiene una naturaleza similar a una función.

manejador de excepciones

¿Qué significa ese identificador `e` en la siguiente línea de un bloque `catch`?

```
catch(int e)
```

Ese identificador `e` se parece a un parámetro y actúa en forma muy parecida a un parámetro. Por lo tanto, le llamaremos **parámetro del bloque** `catch`. (Pero recuerde que esto no significa que el bloque `catch` sea una función.) El parámetro del bloque `catch` tiene dos funciones:

parámetro del bloque `catch`

1. Antes del parámetro del bloque `catch` se coloca un nombre de tipo, el cual especifica qué tipo de valor lanzado puede atrapar el bloque `catch`.
2. El parámetro del bloque `catch` nos proporciona un número para el valor lanzado que se atrapa, por lo que podemos escribir código en el bloque `catch` que trabaje con el valor lanzado que se atrapó.

Hablaremos sobre estas dos funciones del parámetro del bloque `catch` en orden inverso. En esta subsección veremos cómo usar el parámetro del bloque `catch` como un nombre para el valor que se lanzó y se atrapó. En la subsección titulada "Uso de varias instrucciones throw y catch", que veremos más adelante en este capítulo, hablaremos sobre cuál bloque `catch` (cuál manejador de excepciones) procesará un valor que se haya lanzado. Nuestro ejemplo actual sólo tiene un bloque `catch`. El identificador `e` es un nombre común para un parámetro del bloque `catch`, pero podemos utilizar cualquier identificador válido en lugar de `e`.

Veamos cómo funciona el bloque `catch` en el cuadro 17.2. Cuando se lanza un valor, la ejecución del código en el bloque `try` termina y el control se pasa al bloque `catch` (o

bloques) que se coloca justo después del bloque *try*. El bloque *catch* del cuadro 17.2 es el siguiente:

```
catch(int e)
{
     cout << e << " donas, sin leche!\n"
          << "Vaya a comprar leche.\n";
}
```

Cuando se lanza un valor, éste debe ser de tipo *int* para que pueda aplicarse este bloque *catch* específico. En el cuadro 17.2, el valor que se lanza lo proporciona la variable donas, y como es de tipo *int*, este bloque *catch* puede atrapar el valor lanzado.

Suponga que el valor de donas es 12 y que el valor de leche es 0, como en el segundo diálogo de ejemplo del cuadro 17.2. Como el valor de leche no es positivo, se ejecuta la instrucción *throw* que está dentro de la instrucción *if*. En ese caso, se lanza el valor de la variable donas. Cuando el bloque *catch* del cuadro 17.2 atrapa el valor de donas, éste se utiliza para el parámetro e del bloque *catch* y se ejecuta el código que está dentro de este bloque, lo cual produce la siguiente salida:

```
12 donas, y sin leche!
Vaya a comprar leche.
```

Si el valor de donas es positivo, no se ejecuta la instrucción *throw*. En este caso se ejecuta todo el bloque *try*. Después de ejecutar la última instrucción del bloque *try*, se ejecuta la instrucción que va después del bloque *catch*. Tenga en cuenta que si no se lanza una excepción, entonces el bloque *catch* se ignora.

Esto suena como si el conjunto *try-throw-catch* fuera equivalente a una instrucción *if-else*. Es casi equivalente, con la excepción del valor que se lanza. El conjunto de instrucciones *try-throw-catch* es similar a una instrucción *if-else*, pero *con la capacidad agregada de enviar un mensaje a una de las bifurcaciones*. Esto no suena muy distinto de una instrucción *if-else*, pero resulta ser una gran diferencia en la práctica.

Parámetro del bloque *catch*

El parámetro del bloque *catch* es un identificador en el encabezado de un bloque *catch* que sirve como receptáculo para una excepción (un valor) que podría lanzarse. Cuando se lanza un valor (adecuado) en el bloque *try* anterior, ese valor se utiliza para el parámetro del bloque *catch*. Puede usar cualquier identificador válido (palabra no reservada) para un parámetro del bloque *catch*.

Ejemplo

```
catch(int e)
{
    cout << e << " donas, y sin leche!\n"
         << "Vaya a comprar leche.\n";
}
```

e es el parámetro del bloque *catch*.

Para resumir en un tono más formal, un bloque `try` contiene cierto código que suponemos incluye una instrucción `throw`. Esta instrucción por lo general se ejecuta sólo en circunstancias excepcionales, y al ejecutarse lanza un valor de cierto tipo. Cuando se lanza una excepción (un valor como `donas` en el cuadro 17.2), es el final del bloque `try`. Se ignora todo el resto del código en el bloque `try` y el control pasa a un bloque `catch` apropiado. Un bloque `catch` se aplica sólo a un bloque `try` que esté justo antes de éste. Si se lanza la excepción, entonces ese objeto de excepción se utiliza para el parámetro del bloque `catch` y se ejecutan las instrucciones que están dentro del bloque `catch`. Por ejemplo, si analiza los diálogos en el cuadro 17.2, podrá ver que tan pronto como el usuario introduce un número no positivo, el bloque `try` se detiene y se ejecuta el bloque `catch`. Por ahora supondremos que todo bloque `try` va seguido de un bloque `catch` apropiado. Más adelante hablaremos sobre lo que ocurre cuando no hay un bloque `catch` apropiado.

Ahora resumiremos lo que ocurre cuando no se lanza una excepción en un bloque `try`. Si no se lanza una excepción (ningún valor) en el bloque `try`, entonces después de que éste se completa, la ejecución del programa continúa con el código que va después del

`try`–`throw`–`catch`

Éste es el mecanismo básico para lanzar y atrapar excepciones. La **instrucción** `throw` lanza la excepción (un valor). El **bloque** `catch` atrapa la excepción (el valor). Cuando se lanza una excepción, el bloque `try` termina y se ejecuta el código dentro del bloque `catch`. Una vez que se completa el bloque `catch`, se ejecuta el código que va después del (los) bloque(s) `catch` (siempre y cuando el bloque `catch` no haya finalizado el programa o realizado alguna otra acción especial).

Si no se lanza una excepción en el bloque `try` entonces, cuando termine de ejecutarse, el programa continuará ejecutándose desde el código que va después del (los) bloque(s) `catch`. (En otras palabras, si no se lanza una excepción, se ignora(n) el (los) bloque(s) `catch`.

Sintaxis

```
try
{
    Algunas_instrucciones
    <Cierto código con una instrucción throw o
        la invocación de una función que podría lanzar una excepción.>
    Algunas_instrucciones_mas
}
catch(Nombre_tipo e)
{
    <El código que se ejecutará si se lanza un valor
        del tipo del parámetro del bloque catch en el bloque try.>
}
```

Ejemplo

Vea el cuadro 17.2.

bloque *catch*. En otras palabras, si no se lanza una excepción, entonces el bloque *catch* se ignora. La mayor parte del tiempo que se ejecute el programa, la instrucción *throw* no se ejecutará y, en la mayoría de los casos, el código en el bloque *try* se ejecutará hasta completarse y el código en el bloque *catch* se ignorará por completo.

Ejercicios de AUTOEVALUACIÓN

1. ¿Qué salida produce el siguiente código?

```
int tiempo_espera = 46;

try
{
    cout << "Se entro al bloque try.\n";
    if (tiempo_espera > 30)
        throw tiempo_espera;
    cout << "Se salio del bloque try.\n";
}

catch(int valor_lanzado)
{

     cout << "La excepcion se lanzo con\n"
          << "tiempo_espera igual a " << valor_lanzado << endl;
}
cout << "Despues del bloque catch." << endl;
```

2. ¿Cuál sería la salida producida por el código del ejercicio de autoevaluación 1, si hacemos la siguiente modificación? Modifique la línea

```
int tiempo_espera = 46;
```

por

```
int tiempo_espera = 12;
```

3. En el código que se muestra en el ejercicio de autoevaluación 1, ¿cuál es la instrucción *throw*?

4. ¿Qué ocurre cuando se ejecuta una instrucción *throw*? Ésta es una pregunta general. Indique lo que ocurre en general, no sólo lo que ocurre en el código de la pregunta de autoevaluación 1, o en algún otro código de ejemplo.

5. En el código que se muestra en el ejercicio de autoevaluación 1, ¿cuál es el bloque *try*?

6. En el código que se muestra en el ejercicio de autoevaluación 1, ¿cuál es el bloque *catch*?

7. En el código que se muestra en el ejercicio de autoevaluación 1, ¿cuál es el parámetro del bloque *catch*?

Definición de sus propias clases de excepciones

Una instrucción *throw* puede lanzar un valor de cualquier tipo. Lo común es definir una clase cuyos objetos puedan llevar el tipo preciso de información que deseamos lanzar hacia el bloque *catch*. Una razón aún más importante para definir una clase de excepción especializada es para que podamos tener un tipo distinto que identifique cada tipo posible de situación excepcional.

Una clase de excepción es sólo una clase. Lo que hace que sea una clase de excepción es la manera en que se utiliza. Aun así, conviene tener cuidado al elegir el nombre de una clase de excepción, junto con algunos otros detalles.

El cuadro 17.3 contiene un ejemplo de un programa con una clase de excepción definida por el programador. Éste es sólo un pequeño programa para ilustrar algunos detalles de C++ relacionados con el manejo de excepciones. Utiliza demasiada maquinaria para una tarea tan sencilla, pero fuera de eso es un ejemplo impecable de algunos detalles sobre C++.

Observe la instrucción *throw* que reproducimos a continuación:

```
throw SinLeche(donas);
```

La parte `SinLeche(donas)` es la invocación de un constructor para la clase `SinLeche`. El constructor recibe un argumento *int* (en este caso `donas`) y crea un objeto de la clase `SinLeche`. Después, ese objeto se "lanza".

Uso de varios bloques throw y catch

Un bloque *try* puede lanzar cualquier número potencial de valores de excepción, los cuales pueden ser de distintos tipos. En cualquiera de las ejecuciones del bloque *try*, sólo se lanzará una excepción (ya que una excepción que se lanza termina la ejecución del bloque *try*), pero pueden lanzarse distintos tipos de valores de excepción en distintas ocasiones en las que se ejecute el bloque *try*. Cada bloque *catch* sólo puede atrapar valores de un tipo, pero se pueden atrapar valores de excepción de distintos tipos si se coloca más de un bloque *catch* después de un bloque *try*. Por ejemplo, el programa en el cuadro 17.4 tiene dos bloques *catch* después de su bloque *try*.

Observe que no hay parámetro en el bloque *catch* para `DivisionEntreCero`. Si no necesita un parámetro, sólo debe listar el tipo sin parámetro. Hablaremos un poco más sobre este caso en la sección Tip de programación titulada "Clases de excepciones que pueden ser triviales".

RIESGO *Atrape la excepción más especifica primero*

Al atrapar varias excepciones, el orden de los bloques *catch* puede ser importante. Cuando se lanza el valor de una excepción en un bloque *try*, los bloques *catch* que le siguen se prueban en orden, y el primero que concuerde con el tipo de la excepción que se lanzó es el que se ejecuta.

Por ejemplo, el siguiente es un tipo especial de bloque *catch* que atrapará un valor de cualquier tipo que se lance:

```
catch (...)
{
    <Coloque lo que desee aquí.>
}
```

CUADRO 17.3 *Definición de su propia clase de excepción*

Este es sólo un pequeño ejemplo para conocer la sintaxis de C++. No lo tome como ejemplo de un buen uso común del manejo de excepciones.

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

class SinLeche
{
public:
    SinLeche();
    SinLeche(int cuantos);
    int obtiene_donas();
private:
    int cuenta;
};
int main()
{
    int donas, leche;
    double dpv;

    try
    {
        cout << "Escriba el numero de donas:\n";
        cin >> donas;
        cout << "Escriba el numero de vasos de leche:\n";
        cin >> leche;

        if (leche <= 0)
                throw SinLeche(donas);

        dpv = donas/static_cast<double>(leche);
        cout << donas << " donas.\n"
             << leche << " vasos de leche.\n"
             << "Usted tiene " << dpv
             << " donas para cada vaso de leche.\n";
    }
    catch(SinLeche e)
    {
        cout << e.obtiene_donas() << " donas, y sin leche!\n"
             << "Vaya a comprar leche.\n";
    }
    cout << "Fin del programa.";
    return 0;
}
SinLeche::SinLeche()
{}
SinLeche::SinLeche(int cuantos) : cuenta(cuantos)
{}

int SinLeche::obtiene_donas()
{
    return cuenta;
}
```

Los diálogos de ejemplo son los mismos que los del cuadro 17.2.

CUADRO 17.4 *Cómo atrapar varias excepciones (parte 1 de 3)*

```
#include <iostream>          Aunque no se hace aquí, las clases de excepciones pueden
#include <string>            tener sus propios archivos de interfaz e implementación
using namespace std;         y pueden colocarse en un espacio de nombres.

class NumeroNegativo         Éste es otro ejemplo pequeño.
{
public:
    NumeroNegativo();
    NumeroNegativo(string llevame_a_tu_bloque_catch);
    string obtiene_mensaje();
private:
    string mensaje;
};

class DivisionEntreCero
{};

int main()
{
    int jem_hadar, klingons;
    double porcion;

    try
    {
        cout << "Escriba el numero de guerreros Jem Hadar:\n";
        cin >> jem_hadar;
        if (jem_hadar < 0)
             throw NumeroNegativo("Jem Hadar");

        cout << "Cuantos guerreros Klingon tiene?\n";
        cin >> klingons;
        if (klingons < 0)
            throw NumeroNegativo("Klingons");

        if (klingons != 0)
            porcion = jem_hadar/static_cast<double>(klingons);
        else
            throw DivisionEntreCero();
        cout << "Cada Klingon debe luchar contra "
             << porcion << " Jem Hadar.\n";
    }
```

CUADRO 17.4 *Cómo atrapar varias excepciones (parte 2 de 3)*

```
    catch(NumeroNegativo e)
    {
       cout << "No puede tener un numero negativo de "
            << e.obtiene_mensaje() << endl;
    }
    catch(DivisionEntreCero)
    {
       cout << "Pida ayuda.\n";
    }

    cout << "Fin del programa.\n";
    return 0;
}


NumeroNegativo::NumeroNegativo()
{}

NumeroNegativo::NumeroNegativo(string llevame_a_tu_bloque_catch)
    : mensaje(llevame_a_tu_bloque_catch)
 {}

string NumeroNegativo::obtiene_mensaje()
{
    return mensaje;
}
```

Diálogo de ejemplo 1

```
Escriba el numero de guerreros Jem Hadar:
1000
Cuantos guerreros Klingon tiene?
500
Cada Klingon debe luchar contra 2 Jem Hadar.
Fin del programa.
```

CUADRO 17.4 *Cómo atrapar varias excepciones (parte 3 de 3)*

Diálogo de ejemplo 2

```
Escriba el numero de guerreros Jem Hadar:
-10
No puede tener un numero negativo de Jem Hadar
Fin del programa.
```

Diálogo de ejemplo 3

```
Escriba el numero de guerreros Jem Hadar:
1000
Cuantos guerreros Klingon tiene?
0
Pida ayuda.
Fin del programa.
```

Los tres puntos no indican que se omitió algo. En realidad puede escribir esos tres puntos en su programa. Con estos tres puntos podemos colocar un buen bloque *catch* después de todos los demás bloques *catch*. Por ejemplo, podríamos agregarlo a los bloques *catch* del cuadro 17.4, de la siguiente manera:

```
catch(NumeroNegativo e)
{
   cout << "No puede tener un numero negativo de "
        << e.obtiene_mensaje() << endl;
}
catch(DivisionEntreCero)
{
   cout << "Pida ayuda.\n";
}
catch (...)
{
    cout << "Excepcion sin explicacion.\n";
}
```

Sin embargo, sólo tiene sentido colocar este bloque *catch* predeterminado al final de una lista de bloques *catch*. Por ejemplo, suponga que en vez de lo anterior utilizáramos:

```
catch(NumeroNegativo e)
{
    cout << "No puede tener un numero negativo de "
         << e.obtiene_mensaje() << endl;
}
catch (...)
{
    cout << "Excepcion sin explicacion.\n";
}
catch(DivisionEntreCero)
{
     cout << "Pida ayuda.\n";
}
```

Con este segundo ordenamiento, una excepción (un valor lanzado) de tipo `NumeroNegativo` se atrapará mediante el bloque *catch* de `NumeroNegativo` como debe ser. No obstante, si se lanza un valor de tipo `DivisionEntreCero`, sería atrapado por el bloque que empieza con *catch*`(...)`. Por lo tanto, nunca se llegaría al bloque *catch* de `DivisionEntreCero`. Por fortuna, la mayoría de los compiladores le avisan si comete este tipo de error.

TIP DE PROGRAMACIÓN

Las clases de excepciones pueden ser triviales

Aquí reproduciremos la definición de la clase `DivisionEntreCero` del cuadro 17.4:

```
class DivisionEntreCero
{};
```

Esta clase de excepción no tiene variables ni funciones miembro (aparte del constructor predeterminado). Lo único que tiene es su nombre, pero con eso basta. Si se lanza un objeto de la clase `DivisionEntreCero` se puede activar el bloque *catch* apropiado, como en el cuadro 17.4.

Al utilizar una clase de excepción trivial, por lo general no se tiene nada que hacer con la excepción (el valor lanzado) una vez que llega al bloque *catch*. La excepción sólo se utiliza para poder llegar al bloque *catch*. Por ende, podemos omitir el parámetro del bloque *catch*. (Puede omitir el parámetro del bloque *catch* cada vez que no lo necesite, sin importar que el tipo de excepción sea trivial o no.)

Lanzamiento de una excepción en una función

En ocasiones tiene sentido retrasar el manejo de una excepción. Por ejemplo, tal vez tenga una función con código que lance una excepción si hay un intento de dividir entre cero, pero quizás no desee atrapar la excepción en esa función. Tal vez se requiera que ciertos

programas que utilizan esa función sólo terminen si se lanza la excepción, y que otros programas que utilizan la función hagan otra cosa. De ser así, no sabría qué hacer con la excepción si la atrapara dentro de la función. En estos casos es conveniente no atrapar la excepción en la definición de la función, sino hacer que un programa (u otro código) que utilice esa función coloque la invocación a la misma en un bloque *try* y que atrape la excepción en un bloque *catch* que vaya después de ese bloque *try*.

Analice el programa del cuadro 17.5. Tiene un bloque *try*, pero no hay una instrucción *throw* visible en este bloque. La instrucción que se encarga de lanzar la excepción en el programa es

```
if (inferior == 0)
    throw DivisionEntreCero();
```

Esta instrucción no está visible en el bloque *try*. Sin embargo, se encuentra en el bloque *try* en términos de la ejecución del programa, ya que se encuentra en la definición de la función `division_segura` y hay una invocación de esta función en el bloque *try*.

Especificación de excepciones

Si una función no atrapa una excepción, por lo menos debería advertir a los programadores que cualquier invocación de esa función podría tal vez lanzar una excepción. Si existen excepciones que pudieran lanzarse, pero no atraparse en la definición de la función, entonces esos tipos de excepciones deberían listarse en la **especificación de excepciones**, lo cual se ilustra mediante la siguiente declaración de función del cuadro 17.5:

especificación de excepciones

```
double division_segura(int superior, int inferior)
throw (DivisionEntreCero);
```

Como se ilustra en el cuadro 17.5, la especificación de la función debería aparecer tanto en la declaración como en la definición de la función. Si una función tiene más de una declaración, entonces todas las declaraciones de la función deben tener especificaciones de excepciones idénticas. La especificación de excepciones para una función se conoce también algunas veces como la **lista throw**.

lista throw

Si hay más de una excepción que pueda lanzarse en la definición de la función, entonces los tipos de las excepciones se separan por comas, como se muestra a continuación:

```
void una_funcion( )  throw (DivisionEntreCero, OtraExcepcion);
```

Todos los tipos de excepciones que se listan en la especificación de excepciones se tratan de manera normal. Cuando decimos que la excepción se trata de manera normal, significa que se trata como lo hemos descrito antes de esta subsección. En especial, puede colocar la invocación de la función en un bloque *try* seguido de un bloque *catch* para atrapar ese tipo de excepción, y si la función lanza la excepción (y no la atrapa dentro de la función) entonces el bloque *catch* que vaya después del bloque *try* atrapará la excepción.

Si no hay especificación de excepciones (si no hay lista throw, ni siquiera una vacía), entonces es como si se listaran todos los tipos de excepciones posibles en la especificación de excepciones; es decir, cualquier excepción que se lance se tratará de manera normal.

¿Qué ocurre cuando se lanza una excepción en una función, pero no se lista en la especificación de la función (y no se atrapa dentro de la función)? En ese caso, el programa termina. En especial, observe que si se lanza una excepción en una función pero no se lista en la especificación de excepciones (y no se atrapa dentro de la función), entonces ningún

CUADRO 17.5 *Lanzamiento de una excepción dentro de una función (parte 1 de 2)*

```
#include <iostream>
#include <cstdlib>
using namespace std;

class DivisionEntreCero
{};

double division_segura(int superior, int inferior) throw
(Division EntreCero);

int main()
{
     int numerador;
     int denominador;
     double cociente;
     cout << "Escriba el numerador:\n";
     cin >> numerador;
     cout << "Escriba el denominador:\n";
     cin >> denominador;

     try
     {
        cociente = division_segura(numerador, denominador);
     }
     catch(DivisionEntreCero)
     {
          cout << "Error: Division ente cero!\n"
               << "Se abortara el programa.\n";
          exit(0);
     }

     cout << numerador << "/" << denominador
          << " = " << cociente << endl;

     cout << "Fin del programa.\n";
     return 0;
}
```

CUADRO 17.5 *Lanzamiento de una excepción dentro de una función (parte 2 de 2)*

```
double division_segura(int superior, int inferior) throw
(Division EntreCero)
{
    if (inferior == 0)
        throw DivisionEntreCero();

    return superior/static_cast<double>(inferior);
}
```

Diálogo de ejemplo 1

```
Escriba el numerador:
5
Escriba el denominador:
10
5/10 = 0.5
Fin del programa.
```

Diálogo de ejemplo 2

```
Escriba el numerador:
5
Escriba el denominador:
0
Error: Division entre cero!
Se abortara el programa.
```

bloque `catch` la atrapará y su programa terminará. Recuerde que si no hay una lista de especificación, ni siquiera una vacía, entonces es como si todas las excepciones se listaran en la lista de especificaciones, por lo que si se lanzara una excepción el programa no terminaría de la manera descrita en este párrafo.

Tenga en cuenta que la especificación de excepciones es para las excepciones que se "salen" de la función. Si no se salen de la función, no pertenecen en la especificación de excepciones. Si salen de la función, pertenecen en la especificación de excepciones sin

importar en dónde se originan. Si se lanza una excepción en un bloque *try* que se encuentre dentro de la definición de una función y se atrapa en un bloque *catch* dentro de la definición de la función, entonces su tipo no necesita listarse en la especificación de excepciones. Si la definición de una función incluye una invocación de otra función y esa otra función puede lanzar una excepción que no se atrape, entonces el tipo de excepción deberá colocarse en la especificación de excepciones.

Para indicar que una función no debe lanzar excepciones que no se atrapen dentro de ella, debemos utilizar una especificación de excepciones vacía, como se muestra a continuación:

```
void una_funcion( )  throw (  );
```

Como resumen, tenemos que:

```
void una_funcion(  )  throw (DivisionEntreCero, OtraExcepcion);
//Las excepciones de tipo DivisionEntreCero u OtraExcepcion se tratan
//de manera normal. Todas las demás excepciones hacen que el programa
//termine si no se atrapan en el cuerpo de la función.

void una_funcion(  )  throw (  );
//Lista vacía de excepciones: todas las excepciones hacen que el
//programa termine si se lanzan y no se atrapan en el cuerpo de
//la función.

void una_funcion( );
//Todas las excepciones de todos los tipos se tratan de manera
//normal.
```

Tenga en cuenta que un objeto de una clase derivada[1] es también un objeto de su clase base. Por lo tanto, si `D` es una clase derivada de la clase `B` y `B` se encuentra en la especificación de excepciones, entonces un objeto de la clase `D` que se lance se tratará de manera normal, ya que es un objeto de la clase `B` y esta clase está en la especificación de excepciones. Sin embargo, no se realizan conversiones automáticas de tipo. Si *double* está en la especificación de excepciones, no se toma en cuenta para lanzar un valor *int*. Tendría que incluir tanto *int* como *double* en la especificación de excepciones.

¡Advertencia!

Una advertencia final: no todos los compiladores tratan la especificación de excepciones como se espera. Algunos de ellos la tratan como un comentario y, en este caso, la especificación no tiene efecto sobre el código. Ésta es otra razón por la que debemos colocar todas las excepciones que nuestras funciones pudieran lanzar en la especificación de excepciones. De esta forma, todos los compiladores tratarían nuestras excepciones de igual manera. Desde luego que podríamos obtener la misma consistencia del compilador si no tuviéramos una especificación de excepciones, pero entonces nuestro programa no estaría tan bien documentado y no obtendríamos la comprobación adicional de errores que proporcionan los compiladores que sí utilizan la especificación de excepciones. Con un compilador que sí procesara la especificación de excepciones, nuestro programa terminaría tan pronto como lanzara una excepción que no anticipáramos. (Tenga en cuenta que éste es un comportamiento en tiempo de ejecución, pero el comportamiento en tiempo de ejecución que usted obtenga dependerá de su compilador.)

[1] Si no ha visto todavía las clases derivadas, puede ignorar sin problema todos los comentarios sobre ellas.

RIESGO *Especificación de una excepción en clases derivadas*

Cuando redefinimos o sobreescribimos la definición de una función en una clase derivada, debe tener la misma especificación de excepciones que tenía en la clase base, o debe tener una especificación de excepciones cuyas excepciones sean un subconjunto de las que se incluyen en la especificación de excepciones de la clase base. Visto de otra forma, cuando redefinimos o sobrescribimos la definición de una función, no podemos agregar excepciones a la especificación de excepciones (pero sí podemos eliminar ciertas excepciones, si así lo deseamos). Esto tiene sentido, ya que un objeto de la clase derivada puede utilizarse en cualquier lugar en donde pueda utilizarse un objeto de la clase base, por lo que una función redefinida o sobrescrita debe ajustarse a cualquier código que esté escrito para un objeto de la clase base.

Ejercicios de AUTOEVALUACIÓN

8. ¿Cuál es la salida que produce el siguiente programa?

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

void funcion_ejemplo(double prueba)  throw (int);

int main()
{
    try
    {
         cout << "Probando.\n";
         funcion_ejemplo(98.6);
         cout << "Probando despues de la llamada.\n";
    }
    catch(int)
    {
        cout << "Atrapando.\n";
    }

    cout << "Fin del programa.\n";
    return 0;
}
void funcion_ejemplo(double prueba) throw (int)
{
     cout << "Inicio de funcion_ejemplo.\n";
     if (test < 100)
         throw 42;
}
```

9. ¿Cuál es la salida que produce el programa del ejercicio de autoevaluación 8 si se realizara la siguiente modificación en el programa? Modifique

```
funcion_ejemplo(98.6);
```

en el bloque *try* por

```
funcion_ejemplo(212);
```

17.2 Técnicas de programación para el manejo de excepciones

Sólo utilice esto en circunstancias excepcionales.
Warren Peace, *The Lieutenant's Tools*

Hasta ahora le hemos mostrado mucho código que explica cómo funciona el manejo de excepciones en C++, pero aún no hemos visto un ejemplo de un programa que haga un buen uso realista del manejo de excepciones. Sin embargo, ahora que conoce la mecánica del manejo de excepciones, en esta sección podremos explicarle las técnicas del manejo de excepciones.

Cuándo lanzar una excepción

Hemos visto código bastante simple para ilustrar los conceptos básicos del manejo de excepciones. No obstante, nuestros ejemplos son tan simples como irreales. Un mejor lineamiento, aunque más complicado, sería utilizar una función separada para lanzar una excepción y otra función para atraparla. En la mayoría de los casos debemos incluir cualquier instrucción *throw* dentro de la definición de una función, listar la excepción en la especificación de excepciones para esa función y colocar la cláusula *catch* en una *función distinta*. Por ende, el uso preferido del trío *try-throw-catch* sería como se muestra a continuación:

```
void funcionA() throw (MiExcepcion)
{
        .
        .
        .
    throw MiExcepcion(<Tal vez un argumento>);
        .
        .
        .
}
```

Después, en *alguna otra función* (tal vez incluso en alguna otra función de otro archivo), tendríamos

```
void funcionB()
{
        .
        .
        .
```

```
    try
    {
        .
        .
        .
        funcionA();
        .
        .
        .
    }
    catch (MiExcepcion e)
    {
            <Manejo de la excepción.>

    }
        .
        .
        .
}
```

Es más, debería reservarse este tipo de uso de una instrucción *throw* para casos en los que no pueda evitarse. Si puede manejar un problema de alguna otra forma fácil, no lance una excepción. Reserve las instrucciones *throw* para situaciones en las que la manera en que se maneje la condición excepcional dependa de cómo y en dónde se utilizará la función. Si la manera en que se maneja la condición excepcional depende de cómo y en dónde se invoca la función, entonces lo mejor que se puede hacer es dejar que el programador que invoca a la función sea el que maneje la excepción. En todas las demás situaciones, casi siempre es preferible evitar el lanzamiento de excepciones.

Cuándo lanzar una excepción

En su mayoría, las instrucciones *throw* deben utilizarse dentro de funciones y listarse en una especificación de excepciones para la función. Es más, deberían reservarse para situaciones en las que la manera en que se maneje la condición excepcional dependa de cómo y en dónde se utilizará esa función. Si la manera en que se maneja la condición excepcional depende de cómo y en dónde se invoca la función, entonces lo mejor que se puede hacer es dejar que el programador que invoca a la función sea el que maneje la excepción. En todas las demás situaciones, casi siempre es preferible evitar el lanzamiento de excepciones.

RIESGO *Excepciones no atrapadas*

Toda excepción que lance su código deberá atraparse en alguna parte dentro del mismo. Si una excepción se lanza pero no se atrapa en ningún lado, su programa terminará.

RIESGO *Bloques* `try-catch` *anidados*

Puede colocar un bloque `try` y sus correspondientes bloques `catch` que le sigan dentro de un bloque `try` más grande, o dentro de un bloque `catch` más grande. Esto puede ser útil en situaciones poco usuales, pero si se ve tentado a hacer esto, debe tener en cuenta que hay una manera más adecuada de organizar su programa. Casi siempre es mejor colocar los bloques `try-catch` internos dentro de la definición de una función y colocar una invocación de la misma en el bloque `try` o `catch` externo (o tal vez sólo eliminar uno o más bloques `try` por completo).

Si coloca un bloque `try` y sus correspondientes bloques `catch` dentro de un bloque `try` más grande, y si se lanza una excepción en el bloque `try` interno pero no se atrapa en los bloques `try-catch` internos, entonces la excepción se lanzará hacia el bloque `try` exterior para su procesamiento y tal vez se atrape ahí.

RIESGO *Uso excesivo de excepciones*

Las excepciones le permiten escribir programas cuyo flujo de control es tan complicado que es casi imposible comprender el programa. Y por si fuera poco, esto no es difícil de lograr. El proceso de lanzar una excepción le permite transferir el control de flujo desde cualquier parte de su programa hacia casi cualquier otra parte del mismo. Durante los primeros días de la programación, este tipo de control de flujo sin restricción se permitía a través de una instrucción conocida como *goto*. Los expertos de programación ahora concuerdan en que dicho control de flujo sin restricción es un estilo de programación muy pobre. Las excepciones nos permiten regresar a esos viejos y malos días del control de flujo sin restricción. Las excepciones deben utilizarse con medida y sólo en ciertas formas. Una buena regla es la siguiente: si se ve tentado a incluir una instrucción `throw`, piense en cómo podría escribir su programa o la definición de su clase sin esta instrucción `throw`. Si piensa en una alternativa que produzca código razonable, entonces tal vez no sea conveniente incluir la instrucción `throw`.

Jerarquías de las clases de excepciones

Puede ser muy útil definir una jerarquía de clases de excepciones. Por ejemplo, podría tener una clase de excepción `ErrorAritmetico` y después definir una clase de excepción `ErrorDeDivisionEntreCero` que sea una clase derivada de `ErrorAritmetico`. Como una excepción `ErrorDeDivisionEntreCero` es una excepción `ErrorAritmetico`, todos los bloques `catch` para una excepción `ErrorAritmetico` atraparán a una excepción `ErrorDeDivisionEntreCero`. Si lista a `ErrorAritmetico` en una especificación de excepciones, entonces también habrá agregado a `ErrorDeDivisionEntreCero` a la especificación de excepciones, sin importar que incluya o no el nombre de `ErrorDeDivisionEntreCero` en la especificación de excepciones.

Prueba para la memoria disponible

En el capítulo 15 creamos nuevas variables dinámicas con código como el siguiente:

```
struct Nodo
{
     int datos;
     Nodo *enlace;
};
typedef Nodo* NodoPtr;
   ...
NodoPtr apuntador = new Nodo;
```

Esto funciona bien mientras haya suficiente memoria disponible para crear el nuevo nodo. Pero ¿qué ocurre si no hay memoria suficiente? Si no se puede crear el nodo debido a la falta de memoria, entonces se lanza una excepción del tipo `bad_alloc`. Este tipo es parte del lenguaje C++. No necesita definirlo.

Como *new* lanzará una excepción `bad_alloc` cuando no haya suficiente memoria para crear el nodo, puede comprobar si se agota la memoria de la siguiente manera:

```
try
{
     NodoPtr apuntador = new Nodo;
}
catch (bad_alloc)
{
     cout << "Se agoto la memoria!";
}
```

Desde luego que puede hacer otras cosas además de sólo dar un mensaje de advertencia, pero los detalles de lo que deba hacer dependerán de su tarea de programación específica.

Volver a lanzar una excepción

Es válido lanzar una excepción dentro de un bloque *catch*. En situaciones inusuales tal vez sea conveniente atrapar una excepción y luego, dependiendo de los detalles, optar por lanzar la misma excepción o una distinta, para que se maneje más arriba en la cadena de bloques de manejo de excepciones.

Ejercicios de AUTOEVALUACIÓN

10. ¿Qué ocurre si una excepción nunca se atrapa?
11. ¿Puede anidar un bloque *try* dentro de otro bloque *try*?

Resumen del capítulo

- El manejo de excepciones le permite diseñar y codificar el caso normal para su programa en forma separada del código que se encarga de las situaciones excepcionales.
- Una excepción puede lanzarse en un bloque *try*. Una alternativa sería lanzar una excepción en la definición de una función que no incluya un bloque *try* (o que no incluya un bloque *catch* para atrapar ese tipo de excepción). En este caso, puede colocarse una invocación de la función en un bloque *try*.
- Una excepción se atrapa en un bloque *catch*.
- Un bloque *try* puede ir seguido de más de un bloque *catch*. En este caso, siempre debe listar el bloque *catch* para una clase de excepción más específica antes del bloque *catch* para una clase de excepción más general.
- No use las excepciones en forma excesiva.

Respuestas a los ejercicios de autoevaluación

1.
```
Se entro al bloque try.
La excepcion se lanzo con
tiempo_espera igual a 46
Despues del bloque catch.
```

2.
```
Se entro al bloque try.
Se salio del bloque try.
Despues del bloque catch.
```

3.
```
throw tiempo_espera;
```

   Observe que la siguiente es una instrucción *if*, no una instrucción *throw*, aun cuando contiene una instrucción *throw*:

```
if (tiempo_espera > 30)
   throw tiempo_espera;
```

4. Cuando se ejecuta una instrucción *throw* es el final del bloque *try* circundante. Ya no se ejecutan las demás instrucciones del bloque *try*; el control pasa al (los) bloque(s) *catch* que le sigue(n). Cuando decimos que el control pasa al bloque *catch* que le sigue, significa que el valor lanzado se utiliza para el parámetro del bloque *catch* (si lo hay), y se ejecuta el código que está en el bloque *catch*.

5.
```
try
{
     cout << "Se entro al bloque try.";
     if (tiempo_espera > 30)
        throw tiempo_espera;
     cout << "Se salio del bloque try.";
}
```

6.
```
catch(int valor_lanzado)
{
cout << "La excepcion se lanzo con\n"
       << "tiempo_espera igual a " << valor_lanzado << endl;
}
```

7. `valor_lanzado` es el parámetro del bloque *catch*.

8.
```
Probando.
Inicio de funcion_ejemplo.
Atrapando.
Fin del programa.
```

9.
```
Probando.
Inicio de funcion_ejemplo.
Probando despues de la llamada.
Fin del programa.
```

10. Si una excepción no se atrapa en ningún lado, entonces su programa terminará.

11. Sí, puede tener un bloque *try* y sus correspondientes bloques *catch* dentro de otro bloque *try* grande. No obstante, tal vez sería mejor colocar los bloques *try* y *catch* interiores en la definición de una función y colocar una invocación de la función en el bloque *try* más grande.

Proyectos de programación

1. Escriba un programa que convierta la hora del formato de 24 horas al de 12 horas. A continuación se muestra un diálogo de ejemplo:


```
Escriba la hora en notacion de 24 horas:
13:07
Es lo mismo que
1:07 PM
De nuevo? (s/n)
s
Escriba la hora en notacion de 24 horas:
10:15
Es lo mismo que
10:15 AM
De nuevo? (s/n)
s
Escriba la hora en notacion de 24 horas:
10:65
La hora 10:65 no existe
Intente de nuevo:
Escriba la hora en notacion de 24 horas:
16:05
Es lo mismo que
4:05 PM
De nuevo? (s/n)
n
Fin del programa
```

Debe definir una clase de excepción llamada `ErrorEnFormatoDeTiempo`. Si el usuario introduce un tiempo inválido, como `10:65` o incluso como `&&*68`, su programa deberá lanzar y atrapar una excepción `ErrorEnFormatoDeTiempo`.

2. Escriba un programa que convierta fechas de formato numérico tipo mes/día a formato alfabético tipo mes/día (por ejemplo, 1/31 o 01/31 corresponde a Enero 31). El diálogo deberá ser similar al del proyecto de programación 1. Debe definir dos clases de excepciones, una llamada `ErrorMes` y otra llamada `ErrorDia`. Si el usuario introduce cualquier otra cosa que no sea un número de mes válido (enteros del `1` al 12), su programa deberá lanzar y atrapar una excepción `ErrorMes`. De manera similar, si el usuario introduce cualquier otra cosa que no sea un número de día válido (enteros del `1` al `29`, `30` o `31`, dependiendo del mes), su programa deberá lanzar y atrapar una excepción `ErrorDia`. Para mantener las cosas simples, permita que Febrero siempre tenga `29` días.

3. Defina una clase llamada `ArregloComprobado`. Los objetos de esta clase serán como arreglos comunes, pero se comprobará su rango. Si `a` es un objeto de la clase `ArregloComprobado` e `i` es un índice ilegal, entonces el uso de `a[i]` hará que su programa lance una excepción (un objeto) de la clase `ErrorArreglo-FueraDeRango`. La definición de la clase `ErrorArregloFueraDeRango` es parte de este proyecto. Observe que, entre otras cosas, su clase `ArregloComprobado` deberá contar con una sobrecarga apropiada de los operadores `[ ]`, como se describe en el apéndice 7.

4. Las pilas se introdujeron en el capítulo 13. Defina una clase tipo pila para almacenar una pila de elementos de tipo `char`. Un objeto pila deberá ser de tamaño fijo; el tamaño deberá ser un parámetro para el constructor que cree el objeto pila. Cuando se utilice en un programa, un objeto de la clase pila lanzará excepciones en las siguientes situaciones:

   - Lanzará una `ExcepcionDesbordamientoPila` si el programa de aplicación trata de meter datos en una pila que ya esté llena.
   - Lanzará una `ExcepcionPilaVacia` si el programa de aplicación trata de sacar datos de una pila vacía.

   La definición de las clases `ExcepcionDesbordamientoPila` y `ExcepcionPilaVacia` es parte de este proyecto. Escriba un programa de prueba adecuado.

5. (Basado en un problema en el libro de Stroustrup, *The C++ Programming Language*, 3era. Edición) Escriba un programa que consista de funciones que se llamen entre sí, con una profundidad de llamadas de 10. Dé a cada función un argumento que especifique el nivel en el cual deba lanzar una excepción. La función main pedirá y recibirá datos de entrada que especifiquen la profundidad de llamadas (nivel) al cual se debe lanzar una excepción. Después la función `main` llamará a la primera función. La función `main` atrapará la excepción y mostrará el nivel en el cual se lanzó esa excepción. No olvide el caso en el que la profundidad es 0, en donde `main` deberá lanzar y atrapar la excepción.

   Sugerencias: Podría utilizar 10 funciones distintas o 10 copias de la misma función que se llamen unas a otras, pero no lo haga. En vez de ello, para que su código sea compacto utilice una función `main` que llame a otra función que se llame a sí misma en forma recursiva. Suponga que hace esto; ¿es necesaria la restricción en cuanto a la profundidad de las llamadas? Esto puede hacerse sin necesidad de dar a la función argumentos adicionales, pero si no lo puede hacer de esa forma, intente agregar un argumento adicional a la función.

18 Biblioteca de plantillas estándar

18 Biblioteca de plantillas estándar

Las bibliotecas no se hacen; crecen.

AUGUSTINE BIRRELL

Introducción

Hay una extensa colección de estructuras de datos estándar para almacenar datos. Como son tan estándar, tiene sentido tener implementaciones portables estándar de estas estructuras de datos. La Biblioteca de plantillas estándar (STL) incluye bibliotecas para dichas estructuras de datos. En la STL se incluyen implementaciones de la pila, la cola y muchas otras estructuras de datos. Dentro del contexto de la STL, a estas estructuras de datos se les conoce como *clases contenedoras* debido a que se utilizan para almacenar colecciones de datos. En el capítulo 11 presentamos una visión general de la STL mediante la descripción de la clase de plantilla vector, que viene siendo una de las clases contenedoras en la STL. En este capítulo presentaremos una visión general de algunas de las clases incluidas en la STL. No tenemos espacio para ver detalladamente la STL, pero presentaremos suficiente material para ayudarle a que empiece a usar algunas de las clases contenedoras básicas.

La STL fue desarrollada por Alexander Stepanov y Meng Lee en Hewlett-Packard; está basada en la investigación realizada por Stepanov, Lee y David Musser. Es una colección de bibliotecas escritas en el lenguaje C++. Aunque la STL no forma parte del lenguaje básico de C++, sí forma parte del estándar de C++ por lo que cualquier implementación de C++ que se conforme al estándar debe incluir la STL. Como cuestión práctica, puede considerar que la STL es parte del lenguaje C++.

Como su nombre lo indica, las clases en la STL son clases de plantillas. Una clase contenedora típica tiene un parámetro de tipo para el tipo de datos que se va a almacenar en la clase contenedora.

Las clases contenedoras de la STL hacen un uso intensivo de los iteradores, que son objetos que facilitan la acción de recorrer los datos en un contenedor en forma cíclica. En la sección 15.1 vimos una introducción al concepto de un iterador, en donde hablamos sobre los apuntadores

como iteradores. Sería conveniente que leyera esa sección antes de comenzar a leer este capítulo. Si no lo ha hecho, debería leer también la sección 11.3, que trata acerca de la clase de plantilla vector de la STL.

La STL también incluye implementaciones de muchos algoritmos genéricos importantes, como los algoritmos de búsqueda y de ordenamiento. Los algoritmos se implementan como funciones de plantilla. Después de hablar sobre las clases contenedoras describiremos algunas de estas implementaciones de algoritmos.

genéricos

La STL difiere de otras bibliotecas de C++ (como `<iostream>` por ejemplo) en que las clases y los algoritmos son genéricos, que es otra manera de decir que son clases de plantilla y funciones de plantilla.

Prerrequisitos

En este capítulo utilizamos el material de los capítulos 2 al 12, así como de los capítulos 14 y 15.

18.1 Iteradores

El conejo blanco se puso sus anteojos, "¿En dónde debo empezar, su Majestad?" preguntó.

"Empieza en el principio", dijo el Rey, con mucha seriedad, "y continúa hasta que llegues al final; después te detienes".

Lewis Caroll, *Alicia en el país de las maravillas*

Los vectores, que presentamos en el capítulo 11, son una de las clases de plantilla contenedoras en la STL. Los iteradores son una generalización de los apuntadores (el capítulo15 incluye una introducción a los apuntadores que se utilizan como iteradores). En esta sección le mostraremos cómo usar los iteradores con los vectores. Otras clases de plantilla contenedoras que veremos en la sección 18.2 utilizan iteradores de la misma forma. Por lo tanto, todo lo que aprenda sobre los iteradores en esta sección se aplicará en un amplio rango de contenedores, por lo que no se aplica tan sólo en los vectores. Esto refleja uno de los dogmas básicos de la filosofía de la STL: la semántica, el nombramiento y la sintaxis para el uso de iteradores debe ser (y es) uniforme entre los distintos tipos de contenedores. Empezaremos con una revisión y discusión de las declaraciones `using`, que utilizaremos en forma intensiva cuando hablemos sobre los iteradores y la STL.

Declaraciones using

Puede ser conveniente que repase la subsección titulada "Calificación de los nombres" en el capítulo 9 antes de continuar con esta subsección y este capítulo.

Suponga que `mi_funcion` es una función que se define en el espacio de nombres `mi_espacio`. La siguiente declaración `using` le permite utilizar el identificador `mi_funcion` para hacer que indique las versiones de `mi_funcion` definidas en el espacio de nombres `mi_espacio`:

```
using mi_espacio::mi_funcion;
```

Dentro del alcance de esta declaración *using*, una expresión como `mi_funcion(1,2)` significa lo mismo que `mi_espacio::mi_funcion(1,2)`; es decir, dentro del alcance de la declaración *using* el identificador `mi_funcion` siempre indica la versión de `mi_funcion` definida en `mi_espacio`, en contraste a cualquier definición de `mi_funcion` definida en cualquier otro espacio de nombres.

Cuando hablamos sobre iteradores, con frecuencia aplicamos el operador `::` en otro nivel. A menudo verá expresiones como la siguiente:

```
using std::vector<int>::iterator;
```

En este caso, el identificador `iterator` nombra un tipo. Así que dentro del alcance de esta directiva *using*, se permitiría lo siguiente:

```
iterator p;
```

Esto declara a `p` como de tipo `iterator`. ¿Qué es el tipo `iterator`? Este tipo se define en la definición de la clase `vector<int>`. ¿Cuál clase `vector<int>`? La que está definida en el espacio de nombres `std`. (Más adelante explicaremos por completo la función del tipo `iterator`. En este punto sólo nos concentraremos en la explicación de las directivas *using*.)

Tal vez pueda objetar que todo esto es mucho trabajo para nada. No hay una clase `vector<int>` definida en cualquier otro espacio de nombres aparte de `std`. Eso podría ser o no cierto, pero podría haber una clase llamada `vector<int>` definida en algún otro espacio de nombres, ya sea hoy o en un futuro. Quizás podría también objetar que nunca ha escuchado sobre definir un tipo dentro de una clase. No hemos cubierto dichas definiciones, pero son posibles y son comunes en la STL. Por lo tanto, debe saber cómo usar esos tipos, inclusive aunque no los defina.

En resumen, considere la siguiente directiva *using*

```
using std::vector<int>::iterator;
```

Dentro del alcance de esta directiva *using*, el identificador `iterator` indica el tipo llamado `iterator` que se define en la clase `vector<int>`, que a su vez se define en el espacio de nombres `std`.

Fundamentos de los iteradores

iterador

Un **iterador** es la generalización de un apuntador, y de hecho por lo general se implementa mediante el uso de un apuntador, pero la abstracción de un iterador está diseñada para que usted evite todos los detalles relacionados con la implementación, ya que le proporciona una interfaz uniforme para los iteradores que es la misma en todas las distintas clases contenedoras. Cada clase contenedora tiene sus propios tipos de iteradores, de la misma forma que cada tipo de datos tiene su propio tipo de apuntador. Pero así como todos los tipos de apuntadores se comportan en esencia de la misma manera, también cada tipo de iterador se comporta igual, con la diferencia de que se utiliza sólo con su propia clase contenedora.

Un iterador no es un apuntador, pero no estará muy equivocado si piensa en él y lo utiliza como si fuera un apuntador. Al igual que una variable apuntador, una variable iterador se localiza en ("apunta a") una entrada de datos en el contenedor. Para manipular los iteradores se utilizan los siguientes operadores sobrecargados que se aplican a los objetos iteradores:

++ y −−

- Los operadores de incremento ++ prefijo y postfijo para desplazar el iterador hacia el siguiente elemento de datos.

- Los operadores de decremento −− prefijo y postfijo para desplazar el iterador hacia el elemento de datos anterior.
- Los operadores de igualdad y desigualdad, == y !=, para evaluar si dos iteradores apuntan a la misma ubicación de datos.

desreferencia

- Un operador de desreferencia *, de manera que si `p` es una variable iterador, entonces `*p` nos da el acceso a los datos ubicados en ("a los que apunta") `p`. Este acceso puede ser de sólo lectura, de sólo escritura o puede permitir tanto la lectura como la modificación de los datos, dependiendo de la clase contenedora específica.

No todos los iteradores tienen todos estos operadores. No obstante, la clase de plantilla `vector` es un ejemplo de un contenedor cuyos iteradores tienen todos estos operadores, y más.

Una clase contenedora tiene funciones miembro que inician el proceso del iterador. Después de todo, una nueva variable iterador no se ubica en ("apunta a") ningún dato en el contenedor. Muchas clases contenedoras, incluyendo la clase de plantilla `vector`, tienen las siguientes funciones miembro que devuelven objetos iteradores (valores de iteradores) que apuntan a elementos de datos especiales en la estructura de datos:

`c.begin()`

- `c.begin()` devuelve un iterador para el contenedor `c` que apunta al "primer" elemento de datos en el contenedor `c`.

`c.end()`

- `c.end()` devuelve algo que pueda usarse para evaluar si un iterador ha pasado más allá del último elemento de datos en un contenedor `c`. El iterador `c.end()` es completamente análogo al valor `NULL` que se usa para evaluar si un apuntador ha pasado el último nodo en una lista enlazada del tipo que vimos en el capítulo 15. El iterador `c.end()` es por lo tanto un iterador que no se encuentra en ningún elemento de datos, sino que es un tipo de marcador final o centinela.

Para muchas clases contenedoras, estas herramientas le permiten escribir ciclos *for* que iteren a través de todos los elementos en un objeto contenedor `c`, como se muestra a continuación:

```
//p es una variable iterador del tipo para el objeto contenedor c.

for (p = c.begin(); p!= c.end(); p++)
    procesa *p  //*p es el elemento de datos actual.
```

Éstas son las generalidades. Ahora veamos los detalles acerca de la configuración concreta de la clase contenedora de plantilla `vector`.

El cuadro 18.1 muestra el uso de los iteradores con la clase de plantilla `vector`. Tenga en cuenta que cada tipo de contenedor en la `STL` tiene sus propios tipos de iteradores, aunque todos se utilizan de las mismas formas básicas. Los iteradores que queremos para un vector de elementos *int* son del tipo

```
std::vector<int>::iterator
```

La clase de plantilla `list` es otra clase contenedora. Los iteradores para objetos `list` con elementos *int* son del tipo

```
std::list<int>::iterator
```

En el programa del cuadro 18.1 especializamos el nombre de tipo `iterator` para que se aplique a iteradores para vectores de elementos *int*. El nombre de tipo `iterator` que deseamos en el cuadro 18.1 se define en la clase de plantilla `vector`, por lo que si especializamos la clase de plantilla `vector` para elementos *int* y queremos el tipo de iterador para `vector<int>`, queremos el tipo

```
std::vector<int>::iterator;
```

CUADRO 18.1 *Los iteradores que se utilizan con un vector*

```
//Programa que demuestra el uso de los iteradores de la STL.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;
using std::vector<int>::iterator;

int main( )
{
    vector<int> contenedor;

    for (int i = 1; i <= 4; i++)
         contenedor.push_back(i);

    cout << "Esto es lo que hay en el contenedor:\n";
    iterator p;
    for (p = contenedor.begin( ); p != contenedor.end( ); p++)
         cout << *p << " ";
    cout << endl;

    cout << "Se asigno 0 a las entradas:\n";
    for (p = contenedor.begin( ); p != contenedor.end( ); p++)
         *p = 0;

    cout << "El contenedor ahora contiene:\n";
    for (p = contenedor.begin( ); p != contenedor.end( ); p++)
         cout << *p << " ";
    cout << endl;

    return 0;
}
```

Diálogo de ejemplo

```
Esto es lo que hay en el contenedor:
1 2 3 4
Se asigno 0 a las entradas:
El contenedor ahora contiene:
0 0 0 0
```

Como la definición del `vector` coloca el nombre `vector` en el espacio de nombres `std`, la declaración *`using`* completa es

```
using std::vector<int>::iterator;
```

El uso básico de los iteradores con el `vector` (o con cualquier clase contenedora) se ilustra mediante las siguientes líneas del cuadro 18.1:

```
iterator p;
for (p = contenedor.begin( ); p != contenedor.end( ); p++)
     cout << *p << " ";
```

Recuerde que `contenedor` es de tipo `vector<int>` y que el tipo `iterator` en realidad significa `vector<int>::iterator`.

Un vector `v` puede considerarse como un arreglo lineal de sus elementos de datos. Hay un primer elemento de datos `v[0]`, un segundo elemento de datos `v[1]` y así, sucesivamente. Un **iterador** `p` es un objeto que puede **ubicarse en** uno de estos elementos. (Considere que `p` apunta a uno de estos elementos.) Un iterador puede desplazar su ubicación de un elemento hacia otro. Si `p` se ubica, por decir, en `v[7]`, entonces `p++` desplaza `p` de manera que se ubique en `v[8]`. Esto permite que un iterador se desplace a través del vector del primer elemento hasta el último, pero necesita encontrar el primer elemento y saber cuando ha visto el último elemento.

ubicado en

Para saber si un iterador está en la misma ubicación que otro, puede utilizar el operador ==. Por ejemplo, si tiene un iterador que apunta al primero, al último o a cualquier otro elemento, puede probar si otro iterador está ubicado en el primero, último o en otro elemento.

Si `p1` y `p2` son dos iteradores, entonces la comparación

```
p1 == p2
```

es verdadera cuando, y sólo cuando, `p1` y `p2` se ubican en el mismo elemento. (Esto es análogo a los apuntadores. Si `p1` y `p2` fueran apuntadores, esto sería cierto si apuntaran al mismo objeto.) Como siempre, `!=` es sólo la negación de == y por ende

```
p1 != p2
```

es verdadera cuando `p1` y `p2` no se ubican en el mismo elemento.

`begin()`

La función miembro `begin()` se utiliza para colocar un iterador en el primer elemento en un contenedor. Para los vectores y muchas otras clases contenedoras, la función miembro `begin()` devuelve un iterador que se ubica en el primer elemento. (Para un vector `v` el primer elemento es `v[0]`.)
Por lo tanto,

```
iterator p = v.begin();
```

inicializa la variable iterador `p` con un iterador que se ubica en el primer elemento. Así, el ciclo *`for`* básico para visitar todos los elementos del vector `v` sería

```
iterator p;
for (p = v.begin(); Expresion_booleana; p++)
     Accion_en_ubicacion p;
```

La *Expresion_booleana* deseada para una condición de paro es

```
p = v.end()
```

La función miembro `end()` devuelve un valor centinela que puede comprobarse para ver si un iterador ha pasado el último elemento. Si `p` se ubica en este último elemento, entonces después de `p++` la prueba `p = v.end()` cambia de *`false`* a *`true`*. El ciclo *`for`* con la *`Expresión_booleana`* correcta sería entonces

end()

```
iterator p;
for (p = v.begin(); p != v.end(); p++)
    Acción_en_ubicación p;
```

Observe que `p != v.end()` no cambiará de *`true`* a *`false`* sino hasta que la ubicación de `p` haya avanzado más allá del último elemento. Por lo tanto, `v.end()` no se ubica en ningún elemento. El valor `v.end()` es un valor especial que sirve como centinela. No es un iterador ordinario, pero puede comparar `v.end()` con un iterador mediante el uso de `==` y `!=`. El valor `v.end()` es análogo al valor `NULL` que se utiliza para marcar el final de una lista enlazada del tipo que vimos en el capítulo 15.

El siguiente ciclo *`for`* del cuadro 18.1 utiliza esta misma técnica con el vector llamado `contenedor`:

```
iterator p;
for (p = contenedor.begin(); p != contenedor.end(); p++)
    cout << *p << " ";
```

La acción que se realiza en la ubicación del iterador `p` es

```
cout << *p << " ";
```

El operador de desreferencia * está sobrecargado para los iteradores de los contenedores de la STL, de forma que `*p` produzca el elemento en la ubicación `p`. En especial, para un contenedor tipo vector, `*p` produce el elemento que se ubica en el iterador `p`. Por ende, la instrucción `cout` anterior muestra el elemento ubicado en el iterador `p` y, de igual forma, el ciclo *`for`* completo muestra todos los elementos en el contenedor tipo vector.

El **operador de desreferencia** `*p` siempre produce el elemento ubicado en el iterador `p`. En ciertas situaciones `*p` produce un acceso de sólo lectura, lo cual no nos permite modificar el elemento. En otras situaciones nos proporciona acceso al elemento y nos permite modificarlo. Para los vectores, `*p` nos permitirá modificar el elemento ubicado en `p`, como se muestra mediante el siguiente ciclo *`for`* del cuadro 18.1:

operador de desreferencia

```
for (p = contenedor.begin(); p != contenedor.end(); p++)
    *p = 0;
```

Iterador

Un iterador es un objeto que puede utilizarse con un contenedor para obtener acceso a los elementos del mismo. Un iterador es una generalización de la noción de un apuntador, y los operadores ==, !=, ++ y −− se comportan de la misma forma para los iteradores que para los apuntadores. El esquema básico de cómo un iterador puede recorrer en forma cíclica todos los elementos en un contenedor es:

```
iterator p;
for (p = contenedor.begin(); p != contenedor.end(); p++)
    Procesa_elemento_en_ubicacion p;
```

La función miembro `begin()` devuelve un iterador que se encuentra en el primer elemento. La función miembro `end()` devuelve un valor que sirve como valor centinela, una ubicación más allá del último elemento en el contenedor.

Este ciclo *for* recorre todos los elementos en el contenedor tipo vector y asigna un 0 como valor para todos los elementos.

Desreferencia

operador de desreferencia

Cuando se aplica el operador de desreferencia *p a un iterador p se produce el elemento ubicado en el iterador p. Para algunas clases contenedoras de la STL, *p produce un acceso de sólo lectura, lo cual no nos permite modificar el elemento. Para otras clases contenedoras de la STL nos proporciona acceso al elemento y nos permite modificarlo.

Ejercicios de AUTOEVALUACIÓN

1. Si v es un vector, ¿qué es lo que devuelve v.begin()? ¿Qué es lo que devuelve v.end()?
2. Si p es un iterador para un objeto vector v, ¿qué es *p?
3. Suponga que v es un vector de elementos *int*. Escriba un ciclo *for* que muestre todos los elementos de v, excepto el primero.

Tipos de iteradores

Distintos contenedores tienen distintos tipos de iteradores. Los iteradores se clasifican de acuerdo con los tipos de operaciones que se realizan en ellos. Los iteradores tipo vector son de la forma más general; es decir, todas las operaciones funcionan con estos iteradores. Por lo tanto utilizaremos de nuevo el contenedor tipo vector para ilustrar el uso de los iteradores. En este caso utilizaremos un vector para mostrar el uso de los operadores *decremento* y *acceso aleatorio* de los iteradores. El cuadro 18.2 muestra otro programa que utiliza un objeto vector llamado contenedor y un iterador p.

operador decremento

En el cuadro 18.2 se utiliza el **operador decremento**. La línea con este operador se muestra en negritas. Como podría esperar, p−− desplaza el iterador p hacia la ubicación anterior. El operador decremento −− es el mismo que el operador incremento ++, sólo que desplaza al iterador en dirección opuesta.

Los operadores incremento y decremento pueden usarse en notación prefijo (++p) o postfijo (p++). Además de modificar el valor de p, también devuelven un valor. Los detalles del valor devuelto son por completo análogos a lo que ocurre con los operadores incremento y decremento en las variables *int*. En notación prefijo, primero se modifica la variable y se devuelve el valor modificado. En notación postfijo se devuelve el valor antes de modificar la variable. Como es preferible no utilizar los operadores de incremento y decremento como expresiones que devuelven un valor, los utilizaremos sólo para modificar el valor de la variable.

Las siguientes líneas del cuadro 18.2 muestran que con los iteradores de vectores se tiene *acceso aleatorio* a los elementos de un vector como contenedor:

```
iterator p = contenedor.begin();
cout << "La tercera entrada es " << contenedor[2] << endl;
cout << "La tercera entrada es " << p[2] << endl;
cout << "La tercera entrada es " << *(p + 2) << endl;
```

CUADRO 18.2 *Uso de iteradores bidireccionales y de acceso aleatorio (parte 1 de 2)*

```
//Programa para demostrar el uso de iteradores bidireccionales y de acceso aleatorio.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;
using std::vector<char>::iterator;

int main( )
{
     vector<char> contenedor;

     contenedor.push_back('A');
     contenedor.push_back('B');
     contenedor.push_back('C');
     contenedor.push_back('D');

     for (int i = 0; i < 4; i++)
          cout << "contenedor[" << i << "] == "
               << contenedor[i] << endl;

     iterator p = contenedor.begin( );
     cout << "La tercera entrada es " << contenedor[2] << endl;
     cout << "La tercera entrada es " << p[2] << endl;
     cout << "La tercera entrada es " << *(p + 2) << endl;

     cout << "De regreso a contenedor[0].\n";
     p = contenedor.begin( );
     cout << "que tiene el valor " << *p << endl;

     cout << "Dos pasos hacia delante y un paso hacia atras:\n";
     p++;
     cout << *p << endl;
     p++;
     cout << *p << endl;
     p--;
     cout << *p << endl;

     return 0;
}
```

Tres notaciones distintas para lo mismo.

Esta notación es especializada para vectores y arreglos.

Éstas dos funcionan para cualquier iterador de acceso aleatorio.

Éste es el operador de decremento. Funciona para cualquier iterador bidireccional.

CUADRO 18.2 *Uso de iteradores bidireccionales y de acceso aleatorio (parte 2 de 2)*

Diálogo de ejemplo

```
contenedor[0] == A
contenedor[1] == B
contenedor[2] == C
contenedor[3] == D
La tercera entrada es C
La tercera entrada es C
La tercera entrada es C
De regreso a contenedor[0].
que tiene el valor A
Dos pasos hacia delante y un paso hacia atras:
B
C
B
```

acceso aleatorio

Acceso aleatorio significa que podemos avanzar en un paso directo hacia cualquier elemento específico. Ya hemos usado `contenedor[2]` como una forma de acceso aleatorio para un vector. Es tan sólo el operador de corchetes, que es estándar en los arreglos y vectores. Lo nuevo es que podemos usar esta misma notación de corchetes con un iterador. La expresión `p[2]` es una forma de obtener acceso al elemento indizado por el número 2.

Las expresiones `p[2]` y `*(p + 2)` son equivalentes por completo. De una forma análoga a la aritmética de apuntadores (vea el capítulo 12), `(p + 2)` nombra la ubicación que está dos posiciones más allá de `p`. Como `p` se encuentra en la primera ubicación (índice `0`) en el código anterior, `(p + 2)` está en la tercera ubicación (índice `2`). La expresión `(p + 2)` devuelve un iterador. La expresión `*(p + 2)` desreferencia a ese iterador. Desde luego que puede sustituir el `2` con un entero no negativo distinto para obtener un apuntador que apunte hacia un elemento distinto.

Asegúrese de tomar en cuenta que ni `p[2]` ni `(p + 2)` modifican el valor del iterador en la variable iterador `p`. La expresión `(p + 2)` devuelve otro iterador en otra ubicación, pero deja a `p` en donde estaba. Lo mismo ocurre con `p[2]`. Observe también que el significado de `p[2]` y `(p + 2)` depende de la ubicación del iterador en `p`. Por ejemplo, `(p + 2)` indica dos ubicaciones más allá de la ubicación de `p`, en dondequiera que sea.

Por ejemplo, suponga que el código del cuadro 18.2 que vimos antes se sustituyera con lo siguiente (observe que se agrega `p++`):

```
iterator p = contenedor.begin();
p++;
cout << "La tercera entrada es " << contenedor[2] << endl;
cout << "La tercera entrada es " << p[2] << endl;
cout << "La tercera entrada es " << *(p + 2) << endl;
```

La salida de estas tres instrucciones `cout` ya no sería

```
La tercera entrada es C
La tercera entrada es C
La tercera entrada es C
```

sino

```
La tercera entrada es C
La tercera entrada es D
La tercera entrada es D
```

La instrucción `p++` desplaza a `p` de la ubicación `0` a la ubicación `1`, por lo que `(p + 2)` es ahora un iterador en la ubicación `3`, no en la ubicación `2`. Por lo tanto, `*(p + 2)` es equivalente a `contenedor[3]`, no a `contenedor[2]`.

Ahora sabemos lo suficiente acerca de los iteradores como para comprender cómo se clasifican. Los principales tipos de iteradores son

iterador de avance

Iteradores de avance: el operador ++ trabaja sobre el iterador.

iterador bidireccional

Iteradores bidireccionales: tanto ++ como −− trabajan sobre el iterador.

iterador de acceso aleatorio

Iteradores de acceso aleatorio: ++, −− y el acceso aleatorio trabajan sobre el iterador.

Observe que estas categorías son cada vez más fuertes: todo iterador de acceso aleatorio es también un iterador bidireccional, y todo iterador bidireccional es también un iterador de avance.

Como veremos, distintas clases contenedoras de plantilla tienen distintos tipos de iteradores. Los iteradores para la clase de plantilla `vector` son de acceso aleatorio.

Observe que los nombres *iterador de avance*, *iterador bidireccional* e *iterador de acceso aleatorio* se refieren a los tipos de iteradores, no a los nombres de los tipos. Los nombres reales de los tipos serán algo así como `std::vector<int>::iterator`, que en este caso es un iterador de acceso aleatorio.

Tipos de iteradores

Los distintos contenedores tienen distintos tipos de iteradores. Los principales tipos de operadores son:

Iteradores de avance: ++ trabaja sobre el iterador.

Iteradores bidireccionales: tanto ++ como −− trabajan sobre el iterador.

Iteradores de acceso aleatorio: ++, −− y el acceso aleatorio trabajan sobre el iterador.

Ejercicios de AUTOEVALUACIÓN

4. Suponga que el vector `v` contiene las letras `'A'`, `'B'`, `'C'` y `'D'` en ese orden. ¿Cuál es la salida del siguiente código?

```
using std::vector<char>::iterator;
   . . .
iterator i = v.begin();
```

```
i++;
cout << *(i + 2) << " ";
i--;
cout << i[2] << " ";
cout << *(i + 2) << " ";
```

Iteradores constantes y mutables

Cada una de las categorías iterador de avance, iterador bidireccional e iterador de acceso aleatorio se subdivide en dos categorías: *constante* y *mutable*, dependiendo de cómo se comporte el operador de desreferencia con el iterador. Con un **iterador constante** el operador de desreferencia produce una versión de sólo lectura del elemento. Con un iterador `p` constante puede usar `*p` por ejemplo para asignarlo a una variable o mostrarlo en pantalla, pero no puede modificar el elemento en el contenedor si, por ejemplo, le asigna un valor a `*p`. Con un **iterador mutable** p, se puede asignar un valor a `*p` y se modificará el elemento correspondiente en el contenedor. Los iteradores de los vectores son mutables, como se muestra mediante las siguientes líneas del cuadro 18.1:

iterador constante

iterador mutable

```
cout << Se asigno un 0 a las entradas:\n";
for (p = contenedor.begin(); p != contenedor.end(); p++)
    *p = 0;
```

Si un contenedor sólo tiene iteradores constantes, no se puede obtener un iterador mutable para ese contenedor. Pero si un contenedor tiene iteradores mutables y desea un iterador constante para el contenedor, puede tenerlo. Podría ser conveniente tener un iterador constante como un tipo de comprobación de error si pretende que su código no modifique los elementos en el contenedor. Por ejemplo, lo siguiente producirá un iterador constante para un contenedor tipo vector llamado `contenedor`:

```
std::vector<char>::const_iterator p = contenedor.begin();
```

o de forma equivalente

```
using std::vector<char>::const_iterator;
const_iterator p = contenedor.begin();
```

Si se declara `p` de esta manera, lo siguiente produciría un mensaje de error:

```
*p = 'Z';
```

Por ejemplo, el cuadro 18.2 se comportaría de la misma forma si modificara

```
using std::vector<int>::iterator;
```

por

```
using std::vector<int>::const_iterator;
```

y si sustituyera

```
iterator p;
```

por

```
const_iterator p;
```

Sin embargo, un cambio similar no funcionaría en el cuadro 18.1 debido a la siguiente línea del programa en este cuadro:

```
*p = 0;
```

Observe que `const_iterator` es un nombre de tipo, mientras que *iterador constante* es el nombre de un tipo de iterador. No obstante, cada iterador de un tipo llamado `const_iterator` será un iterador constante.

Iterador constante

Un iterador constante es uno que no le permite modificar el elemento en su ubicación.

Iteradores inversos

Algunas veces es conveniente recorrer en forma cíclica los elementos de un contenedor, en orden inverso. Si tiene un contenedor con iteradores bidireccionales, podría estar tentado a probar lo siguiente:

```
iterator p;
for (p = contenedor.end(); p != contenedor.begin(); p--)
    cout << *p << " ";
```

El código compilará y podría hacer que algo así funcionara en algunos sistemas, pero en esencia hay algo mal con esto: `contenedor.end()` no es un iterador regular sino sólo un centinela, y `contenedor.begin()` no es un centinela.

Por fortuna hay una manera sencilla de hacer lo que desea. Para un contenedor con iteradores bidireccionales, hay una forma de invertir todo mediante el uso de un tipo de iterador conocido como **iterador inverso**. Lo siguiente funcionará sin problemas:

iterador inverso

```
reverse_iterador rp;
for (rp = contenedor.rbegin(); rp != contenedor.rend(); rp++)
    cout << *rp << " ";
```

Iteradores inversos

Un iterador inverso puede usarse para recorrer en forma cíclica todos los elementos de un contenedor, siempre y cuando éste tenga iteradores bidireccionales. El esquema general es el siguiente:

```
reverse_iterator rp;
for (rp = c.rbegin(); rp != c.rend(); rp++)
    Procesar_en_ubicacion p;
```

El objeto c es una clase contenedora con iteradores bidireccionales.

Cuando utilice `reverse_iterator`, necesitará tener cierto tipo de declaración *`using`* o algo equivalente. Por ejemplo, si `c` es un `vector<int>`, lo siguiente será suficiente:

```
using std::vector<int>::reverse_iterator;
```

rbegin()
rend()

La función miembro `rbegin()` devuelve un iterador ubicado en el último elemento. La función miembro `rend()` devuelve un centinela que marca el "fin" de los elementos en el orden inverso. Observe que para un iterador de tipo `reverse_iterator`, el operador de incremento ++ desplaza al iterador por los elementos hacia atrás. En otras palabras, se invierte el significado de `--` y ++. El programa en el cuadro 18.3 demuestra el uso de un iterador inverso.

El tipo `reverse_iterator` también tiene una versión constante, llamada `const_reverse_iterator`.

RIESGO *Problemas del compilador*

Algunos compiladores siguen teniendo problemas con las declaraciones de los iteradores. Hay distintas formas en las que puede declarar un iterador. Por ejemplo, hemos estado utilizando lo siguiente:

```
using std::vector<char>::iterator;
 . . .
iterator p;
```

De manera alternativa podría utilizar

```
std::vector<char>::iterator p;
```

No se ve tan agradable, pero también podría usar

```
using namespace std;
 . . .
vector<char>::iterator p;
```

y existen otras variaciones similares.

Su compilador debe aceptar cualquiera de estas alternativas. Sin embargo, hemos descubierto que algunos compiladores sólo aceptan algunas de ellas. Si una no funciona con su compilador, pruebe otra.

Otros tipos de iteradores

iterador de entrada
iterador de salida

Existen otros tipos de iteradores que no cubriremos en este libro. Haremos una breve mención de dos tipos de iteradores cuyos nombres podría encontrarse. Un **iterador de entrada** es en esencia un iterador de avance que puede utilizarse con flujos de entrada. Un **iterador de salida** es en esencia un iterador de avance que puede usarse con flujos de salida. Para obtener más detalles tendrá que consultar una referencia más avanzada.

CUADRO 18.3 *El iterador inverso*

```
//Programa que demuestra el uso de un iterador inverso.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;
using std::vector<char>::iterator;
using std::vector<char>::reverse_iterator;

int main( )
{
    vector<char> contenedor;

    contenedor.push_back('A');
    contenedor.push_back('B');
    contenedor.push_back('C');

    cout << "Hacia delante:\n";
    iterator p;
    for (p = contenedor.begin( ); p != contenedor.end( ); p++)
        cout << *p << " ";
    cout << endl;

    cout << "Hacia atras:\n";
    reverse_iterator rp;
    for (rp = contenedor.rbegin( ); rp != contenedor.rend( ); rp++)
        cout << *rp << " ";
    cout << endl;

    return 0;
}
```

Diálogo de ejemplo

```
Hacia delante:
A B C
Hacia atras:
C B A
```

Ejercicios de AUTOEVALUACIÓN

5. Suponga que el vector `v` contiene las letras `'A'`, `'B'`, `'C'` y `'D'` en ese orden. ¿Cuál es la salida del siguiente código?

```
using std::vector<char>::reverse_iterator;
   . . .
reverse_iterator i = v.rbegin();
i++; i++;
cout << *i << " ";
i–;
cout << *i << " ";
```

6. Suponga que desea ejecutar el siguiente código, en donde `v` es un vector de elementos `int`:

```
for (p = v.begin(); p != v.end(); p++)
    cout << *p << " ";
```

¿Cuál de las siguientes es una manera posible de declarar `p`?

```
std::vector<int>::iterator p;
std::vector<int>::const_iterator p;
```

18.2 Contenedores

Coloca todos tus huevos en una canasta y
—VIGILA ESA CANASTA.
Mark Twain, *Pudd'nhead Wilson*

clase contenedora

Las **clases contenedoras** de la STL son distintos tipos de estructuras de datos para almacenar información como listas, colas y pilas. Cada una de ellas es una clase de plantilla con un parámetro para el tipo específico de datos a almacenar. Por ejemplo, puede especificar una lista como lista con elementos `int`, `double` o `string`, o de cualquier clase o tipo `struct` que desee. Cada clase contenedora de plantilla puede tener sus propias funciones especializadas de acceso y de mutación para agregar y extraer datos del contenedor. Las distintas clases contenedoras pueden tener distintos tipos de iteradores. Por ejemplo, una clase contenedora podría tener iteradores bidireccionales, mientras que otra clase contenedora podría tener sólo iteradores de avance. No obstante, cada vez que se definen, los operadores de iteradores y las funciones miembro `begin()` y `end()` tienen el mismo significado para todas las clases contenedoras de la STL.

Contenedores secuenciales

Un contenedor secuencial ordena sus elementos de datos en una lista de manera que haya un primer elemento, un segundo elemento y así sucesivamente hasta un último elemento. Las listas enlazadas que vimos en el capítulo 15 son ejemplos de un tipo de lista. Algunas veces a las listas que vimos en el capítulo 15 se les conoce como **listas simplemente enlazadas**, ya que sólo hay un enlace de una ubicación a otra. La STL no tiene un contenedor

lista simplemente enlazada

que corresponda con dicha lista simplemente enlazada, aunque algunas implementaciones ofrecen una implementación de una lista simplemente enlazada, por lo general bajo el nombre `slist`. La lista más simple que forma parte de la STL es la **lista doblemente enlazada**, que es la clase de plantilla llamada `list`. En el cuadro 18.4 se ilustra la diferencia entre estos dos tipos de listas.

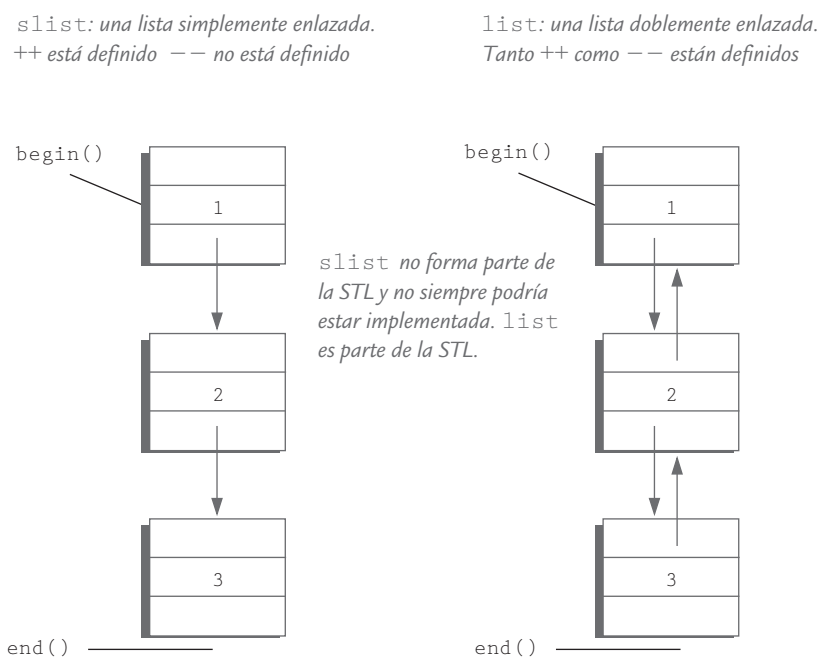
lista doblemente enlazada

Las listas en el cuadro 18.4 contienen los tres valores enteros `1`, `2` y `3` en ese orden. Los tipos para las dos listas son `slist<int>` y `list<int>`. Ese cuadro también indica la ubicación de los iteradores `begin( )` y `end( )`. Todavía no le hemos dicho cómo puede introducir los enteros en las listas.

`slist` y `list`

En el cuadro 18.4 hemos dibujado nuestras listas simplemente enlazada y doblemente enlazada como nodos y apuntadores de la forma que vimos en el capítulo 15. La clase `list` de la STL y la clase `slist` no estándar podrían (o no) implementarse de esta forma. No obstante, al utilizar las clases de plantilla de la STL queda protegido de estos detalles de implementación. Así, sólo necesita pensar en términos de ubicaciones para los datos (que podrían o no ser nodos) e iteradores (no apuntadores). Puede considerar que las flechas en el cuadro 18.4 indican las direcciones para ++ (hacia abajo) y −− (hacia arriba en el cuadro 18.4).

CUADRO 18.4 *Dos tipos de listas*



Presentaremos la clase de plantilla `slist` para ayudar a proporcionar un contexto para los contenedores secuenciales. Corresponde a lo que vimos en gran parte en el capítulo 15 y es lo primero que viene a la mente de la mayoría de los programadores cuando se mencionan las *listas enlazadas*. No obstante, como la clase de plantilla `slist` no es estándar, no hablaremos más sobre ella. Si su implementación de C++ ofrece la clase de plantilla `slist` y desea utilizarla, los detalles son similares a los que describiremos para `list`, con la excepción de que los operadores decremento `--` (prefijo y postfijo) no están definidos para `slist`.

`push_back`

En el cuadro 18.5 se muestra un programa simple que utiliza la clase de plantilla `list` de la STL. La función `push_back` agrega un elemento al final de la lista. Observe que para la clase de plantilla `list`, el operador de desreferencia le proporciona acceso a los datos para su lectura y modificación. Observe además que con la clase de plantilla `list` y con todas las clases de plantilla e iteradores de la STL, todas las definiciones se colocan en el espacio de nombres `std`.

espacio de nombres

El cuadro 18.5 podría compilarse y ejecutarse de la misma forma exacta si sustituimos `list` y `list<int>` con `vector` y `vector<int>`, respectivamente. Esta uniformidad en el uso es una parte clave de la sintaxis de la STL.

Sin embargo, existen diferencias entre un contenedor tipo vector y una lista. Una de las principales diferencias es que un contenedor tipo vector tiene iteradores de acceso aleatorio, mientras que una lista sólo tiene iteradores bidireccionales. Por ejemplo, si empieza con el cuadro 18.2, que utiliza acceso aleatorio, y sustituye todas las ocurrencias de `vector` y `vector<char>` con `list` y `list<char>` respectivamente y después compila el programa, recibirá un error de compilación. (Recibirá un mensaje de error aun cuando elimine las instrucciones que contienen `contenedor[i]` o `contenedor[2]`.)

En el cuadro 18.6 se muestran las clases contenedoras secuenciales básicas de plantilla de la STL. En el cuadro 18.7 aparece un ejemplo de algunas funciones miembro. Otros contenedores, como las pilas y las colas, pueden obtenerse a partir del uso de estas técnicas que describimos en la sección titulada "Los adaptadores de contenedores `stack` y `queue`".

destructores

Todas estas clases de plantilla secuenciales tienen un destructor que devuelve el almacenamiento para su reciclaje.

`deque`

Deque significa "cola con doble terminación". Un deque es un tipo de super cola. Con una cola puede agregar datos en un extremo de la secuencia de datos y extraer datos del otro extremo. Con una deque puede agregar datos en cualquier extremo y extraerlos de cualquier extremo. La clase de plantilla `deque` es una clase de plantilla para un deque con un parámetro para el tipo de datos almacenados.

Contenedores secuenciales

Un contenedor secuencial ordena sus elementos de datos en una lista, de manera que haya un primer elemento, un siguiente elemento, y así sucesivamente hasta un último elemento. Las clases contenedoras de plantilla secuenciales que hemos visto son `slist`, `list`, `vector` y `deque`.

CUADRO 18.5 *Uso de la clase de plantilla* `list`

```
//Programa que demuestra el uso de la clase de plantilla list de la STL.
#include <iostream>
#include <list>
using std::cout;
using std::endl;
using std::list;
using std::list<int>::iterator;

int main( )
{
    list<int> objetoLista;

    for (int i = 1; i <= 3; i++)
        objetoLista.push_back(i);

    cout << "La lista contiene:\n";
    iterator iter;
    for (iter = objetoLista.begin( ); iter != objetoLista.end( ); iter++)
        cout << *iter << " ";
    cout << endl;

    cout << "Se asigno un 0 a todas las entradas:\n";
    for (iter = objetoLista.begin( ); iter != objetoLista.end( ); iter++)
        *iter = 0;

    cout << "La lista ahora contiene:\n";
    for (iter = objetoLista.begin( ); iter != objetoLista.end( ); iter++)
        cout << *iter << " ";
    cout << endl;

    return 0;
}
```

Diálogo de ejemplo

```
La lista contiene:
1 2 3
Se asigno un 0 a todas las entradas:
La lista ahora contiene:
0 0 0
```

CUADRO 18.6 *Contenedores secuenciales básicos de la STL*

Nombre de la clase de plantilla	Nombres de los tipos de iteradores	Tipos de iteradores	Archivo de encabezado de biblioteca
`slist` Advertencia: `slist` no forma parte de la STL.	`slist<T>::iterator` `slist<T>::const_iterator`	mutable hacia delante constante hacia delante	`<slist>` Depende de la implementación y puede no estar disponible.

`list`	`list<T>::iterator`
	`list<T>::const_iterator`
	`list<T>::reverse_iterator`
	`list<T>::const_reverse_iterator`
		mutable bidireccional
		constante bidireccional
		mutable bidireccional
		constante bidireccional
			`<list>`

`vector`	`vector<T>::iterator`
	`vector<T>::const_iterator`
	`vector<T>::reverse_iterator`
	`vector<T>::const_reverse_iterator`
		mutable de acceso aleatorio
		constante de acceso aleatorio
		mutable de acceso aleatorio
		constante de acceso aleatorio
			`<vector>`

`deque`	`deque<T>::iterator`
	`deque<T>::const_iterator`
	`deque<T>::reverse_iterator`
	`deque<T>::const_reverse_iterator`
		mutable de acceso aleatorio
		constante de acceso aleatorio
		mutable de acceso aleatorio
		constante de acceso aleatorio
			`<deque>`

CUADRO 18.7 *Algunas funciones miembro de los contenedores secuenciales (parte 1 de 2)*

Función miembro (`c` es un objeto contenedor)	Significado
`c.size()`	Devuelve el número de elementos en el contenedor.
`c.begin()`	Devuelve un iterador ubicado en el primer elemento del contenedor.
`c.end()`	Devuelve un iterador ubicado una posición más allá del último elemento en el contenedor.
`c.rbegin()`	Devuelve un iterador ubicado en el último elemento del contenedor. Se utiliza con `reverse_iterator`. No es miembro de `slist`.

CUADRO 18.7 *Algunas funciones miembro de los contenedores secuenciales (parte 2 de 2)*

Función miembro (`c` es un objeto contenedor)	Significado
`c.rend()`	Devuelve un iterador ubicado una posición más allá del primer elemento en el contenedor. Se utiliza con `reverse_iterator`. No es miembro de `slist`.
`c.push_back(`*Elemento*`)`	Inserta el *Elemento* al final de la secuencia. No es miembro de `slist`.
`c.insert(`*Iterador, Elemento*`)`	Inserta el *Elemento* al frente de la secuencia. No es miembro de `vector`.
`c.insert(`*Iterador, Elemento*`)`	Inserta una copia de *Elemento* antes de la ubicación de *Iterador*.
`c.erase(`*Iterador*`)`	Elimina el elemento en la posición Iterador. Devuelve un iterador en la posición inmediata siguiente. `Devuelve c.end()` si se elimina el último elemento.
`c.clear()`	Una función `void` que elimina todos los elementos del contenedor.
`c.front()`	Devuelve una referencia al elemento que está al frente de la secuencia. Equivalente a `*(c.begin())`.
`c1 == c2`	Verdadero si `c1.size() == c2.size()` y si cada elemento de `c1` es igual a su correspondiente elemento de `c2`.
`c1 != c2`	`!(c1 == c2)`

Todos los contenedores secuenciales que vimos en esta sección también tienen un constructor, un constructor de copia y diversos constructores predeterminados para inicializar el contenedor con elementos predeterminados o especificados. Cada uno tiene también un destructor que devuelve todo el almacenamiento para su reciclaje, además de un operador de asignación con el comportamiento correcto.

RIESGO *Los iteradores y la eliminación de elementos*

Cuando agrega o elimina un elemento en o de un contenedor, otros iteradores podrían verse afectados. En general, no hay garantía de que los iteradores se ubiquen en el mismo elemento después de una operación de agregar o eliminar. No obstante, algunos contenedores garantizan que los iteradores no se desplazarán debido a las operaciones de agregar o eliminar elementos, excepto por supuesto si el iterador se ubica en un elemento que se vaya a eliminar.

De las clases de plantilla que hemos visto hasta ahora, `list` y `slist` garantizan que sus iteradores no se desplazarán debido a operaciones de agregar o eliminar elementos, excepto desde luego si el iterador se ubica en un elemento que se elimine. Las clases de plantilla `vector` y `deque` no tienen esa garantía.

TIP DE PROGRAMACIÓN

Definiciones de tipos en los contenedores

Las clases contenedoras de la STL contienen definiciones de tipos que pueden ser útiles cuando se programa con estas clases. Ya hemos visto que las clases contenedoras de la STL pueden contener los siguientes nombres de tipos: `iterator`, `const_iterator`, `reverse_iterator` y `const_reverse_iterator` (y por lo tanto, deben contener sus definiciones de tipos ocultas). Por lo general también hay otras definiciones de tipos.

Todas las clases de plantilla que hemos visto hasta ahora tienen los tipos definidos `value_type` y `size_type`. El tipo `value_type` es el tipo de los elementos que se almacenan en el contenedor. Por ejemplo, `list<int>::value_type` es otro nombre para *`int`*. Otro tipo definido es `size_type`, el cual es un tipo entero sin signo que representa el tipo de retorno para la función miembro `size`. Como vimos en el capítulo 11, el tipo `size_type` para la clase de plantilla vector es *`unsigned int`*, aunque la mayoría de los compiladores estarían felices si usted considerara el tipo sólo como *`int`*.

Ejercicios de AUTOEVALUACIÓN

7. ¿Cuál es una de las principales diferencias entre un contenedor `vector` y un contenedor `list`?

8. ¿Cuáles de las siguientes clases de plantilla `slist`, `list`, `vector` y `deque` tienen la función miembro `push_back`?

9. ¿Cuáles de las clases de plantilla `slist`, `list`, `vector` y `deque` tienen iteradores de acceso aleatorio?

10. ¿Cuáles de las clases de plantilla `slist`, `list`, `vector` y `deque` pueden tener iteradores mutables?

Los adaptadores de contenedor `stack` y `queue`

Los adaptadores de contenedor son clases de plantilla que se implementan encima de otras clases. Por ejemplo, la clase de plantilla `stack` se implementa de manera predeterminada encima de la clase de plantilla `deque`, lo cual significa que muy en el fondo de la implementación del adaptador de contenedor `stack` se encuentra un contenedor `deque`, que es en donde residen todos los datos. No obstante, usted queda protegido de estos detalles de implementación y ve a una pila (`stack`) como una simple estructura de datos del tipo "último en entrar/primero en salir".

cola de prioridades

Las clases de plantilla `queue` y `priority_queue` son otras de las clases de adaptadores de contenedor. En el capítulo 15 hablamos sobre las pilas y las colas. Una **cola de prioridades** es como una cola con la propiedad adicional de que cada entrada recibe una prioridad cuando se agrega a la cola. Si todas las entradas tienen la misma prioridad, en una cola de prioridades se extraen de la misma forma en que se extraen de una cola. Si los elementos tienen distintas prioridades, los elementos de mayor prioridad se extraen antes de los elementos de menor prioridad. No hablaremos con detalle sobre las colas de prioridades; sólo las mencionamos para aquellos que estén familiarizados con el concepto.

Aunque una clase de adaptador de plantilla tiene una clase contenedora en base a la cual se crea, puede optar por especificar un contenedor subyacente distinto, por cuestión de eficiencia u otras razones que dependan de su aplicación. Por ejemplo, cualquier contenedor secuencial puede servir como contenedor subyacente para una clase `stack` y cualquier

otro contenedor secuencial para el que pueda servir `vector` como contenedor subyacente para una clase `queue`. La estructura de datos subyacente predeterminada es `deque`, tanto para `stack` como para `queue`. Para una cola de prioridades (`priority_deque`), el contenedor predeterminado subyacente es `vector`. Si no tiene problemas con el tipo de contenedor predeterminado subyacente, entonces un adaptador de contenedor se verá como cualquier otra clase contenedora para usted. Por ejemplo, el nombre de tipo para la clase de plantilla `stack` que utiliza el contenedor predeterminado subyacente es `stack<int>` para una pila de elementos `int`. Si desea especificar que el contenedor subyacente sea mejor la clase de plantilla `vector`, podría usar `stack<int, vector<int> >` como el nombre del tipo. Nosotros siempre utilizaremos el contenedor predeterminado subyacente.

Advertencia

Si especifica un contenedor subyacente, debe estar consciente de que no debe colocar dos símbolos `>` en la expresión de tipos sin un espacio entre ellos; de lo contrario el compilador podría confundirse. Use `stack<int, vector<int> >` con un espacio entre los últimos dos `>`. No utilice `stack<int, vector<int>>`.

Plantilla stack
Plantilla queue

En el cuadro 18.8 se muestran las funciones miembro y otros detalles relacionados con la clase de plantilla `stack`. En el cuadro 18.9 se muestran los mismos detalles para la clase de plantilla `queue`. En el cuadro 18.10 se muestra un ejemplo simple sobre el uso de la clase de plantilla `stack`.

Ejercicios de AUTOEVALUACIÓN

11. ¿Qué tipo de iteradores (de avance, bidireccional o de acceso aleatorio) tiene la clase de adaptador de plantilla `stack`?

12. ¿Qué tipo de iteradores (de avance, bidireccional o de acceso aleatorio) tiene la clase de adaptador de plantilla `queue`?

13. Si `s` es un objeto `stack<char>`, ¿cuál es el tipo de valor de retorno de `s.pop()`?

Los contenedores asociativos `set` y `map`

contenedores asociativos
clave

En esencia, los **contenedores asociativos** son bases de datos muy simples. Almacenan datos como objetos `struct` o cualquier otro tipo de datos. Cada elemento de datos tiene un valor asociado, el cual se conoce como su **clave**. Por ejemplo, si los datos son un objeto `struct` con un registro de empleados, la clave podría ser el número de seguro social de los empleados. Los elementos se recuperan en base a la clave. El tipo de la clave y el tipo de los datos a almacenar no necesitan tener relación entre sí, aunque a menudo están relacionados. Un caso muy simple es cuando cada elemento de datos es su propia clave. Por ejemplo, en un objeto de la clase `set` cada elemento es su propia clave.

`set`

En cierto sentido, la clase de plantilla `set` (conjunto) es el contenedor más simple que pueda imaginar. Almacena elementos sin repetición. La primera inserción coloca un elemento en el conjunto. Las inserciones adicionales después de la primera no tienen efecto, por lo que ningún elemento aparece más de una vez. Cada elemento es su propia clave; en esencia, sólo se agregan o eliminan elementos y se pregunta si el elemento está dentro del objeto `set` o no. Al igual que todas las clases de la STL, la clase de plantilla `set` se escribió con un enfoque en la eficiencia. Para que pueda funcionar con eficiencia, un objeto `set` almacena sus valores en orden. Puede especificar el orden que se utilizará para almacenar los elementos de la siguiente manera:

```
set<T, Ordenamiento> s;
```

CUADRO 18.8 *La clase de plantilla* `Stack`

Detalles relacionados con la clase de adaptador de plantilla `Stack`

Escriba el nombre `stack<T>` o `stack<T, Contenedor_subyacente>` para una pila de elementos de tipo `T`.

Encabezado de biblioteca: `<stack>`, el cual coloca la definición en el nombre de espacio `std`.

Tipos definidos: `value_type`, `size_type`.

No hay iteradores.

Funciones miembro de ejemplo

Función miembro (`s` es un objeto stack)	**Significado**
`s.size()`	Devuelve el número de elementos en la pila.
`s.emtpy()`	Devuelve *`true`* si la pila está vacía; en caso contrario devuelve *`false`*.
`s.top()`	Devuelve una referencia mutable para el miembro superior de la pila.
`s.push(`*Elemento*`)`	Inserta una copia de *Elemento* en la parte superior de la pila.
`s.pop()`	Extrae el elemento superior de la pila. Observe que `pop` es una función `void`. No devuelve el elemento extraído.
`s1 == s2`	Devuelve *`true`* si `s1.size() == s2.size()` y cada elemento de `s1` es igual al elemento correspondiente de `s2`; en caso contrario devuelve *`false`*.

La clase de plantilla `stack` también tiene un constructor predeterminado, un constructor de copia y un constructor que toma un objeto de cualquier clase contenedora secuencial e inicializa la pila con los elementos en la secuencia. También tiene un destructor que devuelve todo el almacenamiento para su reciclaje y un operador de asignación con un comportamiento correcto.

CUADRO 18.9 *La clase de plantilla* `Queue`

Detalles relacionados con la clase de adaptador de plantilla `Queue`

Escriba el nombre `queue<T>` o `queue<T, Contenedor_subyacente>` para una cola de elementos de tipo `T`. Por cuestión de eficiencia, el `Contenedor_subyacente` no puede ser un tipo `vector`.

Encabezado de biblioteca: `<queue>`, el cual coloca la definición en el espacio de nombres `std`.

Tipos definidos: `value_type`, `size_type`.

No hay iteradores.

Funciones miembro de ejemplo

Función miembro (`q` es un objeto queue).	**Significado**
`q.size()`	Devuelve el número de elementos en la cola.
`q.emtpy()`	Devuelve `true` si la cola está vacía; en caso contrario devuelve `false`.
`q.front()`	Devuelve una referencia mutable para el primer miembro de la cola.
`q.back()`	Devuelve una referencia mutable para el último miembro de la cola.
`q.push(`*Elemento*`)`	Agrega *Elemento* a la parte posterior de la cola.
`q.pop()`	Extrae el primer elemento de la cola. Observe que `pop` es una función *`void`*. No devuelve el elemento extraído.
`q1 == q2`	Devuelve true si `q1.size() == q2.size()` y cada elemento de `q1` es igual al elemento correspondiente de `q2`; en caso contrario devuelve *`false`*.

La clase de plantilla `queue` también tiene un constructor predeterminado, un constructor de copia y un constructor que toma un objeto de cualquier clase contenedora secuencial e inicializa la pila con los elementos en la secuencia. También tiene un destructor que devuelve todo el almacenamiento para su reciclaje y un operador de asignación con un comportamiento correcto.

CUADRO 18.10 *Programa que utiliza la clase de plantilla* `Pila`

```
//Programa que demuestra el uso de la clase de plantilla stack de la STL.
#include <iostream>
#include <stack>
using std::cin;
using std::cout;
using std::endl;
using std::stack;

int main( )
{
    stack<char> s;

    cout << "Escriba una linea de texto:\n";
    char siguiente;
    cin.get(siguiente);
    while (siguiente != '\n')
    {
        s.push(siguiente);
        cin.get(siguiente);
    }

    cout << "Escrita al reves seria:\n";
    while ( ! s.empty( ) )
    {
        cout << s.top( );
        s.pop( );
    }
    cout << endl;

    return 0;
}
```

La función miembro `pop` *extrae un elemento, pero no lo devuelve.* `pop` *es una función* `void`*. Por lo tanto, necesitamos usar* `top` *para leer el elemento que extraemos.*

Diálogo de ejemplo

```
Escriba una linea de texto:
popote
Escrita al reves seria:
etopop
```

Ordenamiento debe ser una relación de ordenamiento que se comporte en forma correcta y que reciba dos argumentos de tipo `T`; además debe devolver un valor `bool`.[1] `T` es el tipo de elementos almacenados. Si no se especifica un orden, entonces se asume que el ordenamiento será el operador relacional `<`. En el cuadro 18.11 se muestran algunos detalles básicos relacionados con la clase de plantilla `set`. En el cuadro 18.12 aparece un ejemplo simple que muestra cómo utilizar algunas de las funciones miembro de la clase de plantilla `set`.

mapa

En esencia, un **mapa** es una función que se proporciona como un conjunto de pares ordenados. Para cada valor `primero` que aparece en un par, hay a lo mucho un valor `segundo`, de tal forma que `(primero, segundo)` se encuentre en el mapa. La clase de plantilla `map` implementa objetos mapa en la STL. Por ejemplo, si desea asignar un número único a cada nombre de cadena, podría declarar un objeto `map` de la siguiente manera:

```
map<string, int> numero_mapa;
```

Para los valores `string` que se conocen como *claves*, el objeto `numero_mapa` puede asociar un valor `int` único.

Al igual que un objeto `set`, un objeto `map` almacena sus elementos en orden, en base a los valores de sus claves. Puede especificar el ordenamiento en base a las claves como una tercera entrada en los corchetes angulares, `< >`. Si no especifica un orden, se utiliza el predeterminado. Las restricciones en cuanto a los ordenamientos que se pueden utilizar son las mismas que para los ordenamientos que se permiten para la clase de plantilla `set`. Observe que el ordenamiento es en base a los valores de las claves solamente. El segundo tipo puede ser cualquier tipo y no necesita tener nada que ver con el ordenamiento. Al igual que con el objeto `set`, el ordenamiento de las entradas ordenadas en un objeto `map` se realiza por cuestión de eficiencia.

En el cuadro 18.13 se muestran algunos detalles básicos acerca de la clase de plantilla `map`. Para poder comprender estos detalles, primero necesita saber algo acerca de la clase de plantilla `pair`.

`pair`

La clase de plantilla `pair<T1, T2>` tiene objetos que son pares de valores, de tal forma que el primer elemento sea del tipo `T1` y el segundo de tipo `T2`. Si `unPar` es un objeto de tipo `pair<T1, T2>`, entonces `unPar.primero` es el primer elemento, que es de tipo `T1` y `unPar.segundo` es el segundo elemento, que es de tipo `T2`. Las variables miembro `primero` y `segundo` son variables miembro públicas, por lo que no se necesitan funciones de acceso o de mutación.

El archivo de encabezado para la plantilla `pair` es `<utility>`. Por lo tanto, para usar la clase de plantilla `pair` necesita las siguientes líneas (o algo parecido) en su archivo:

```
#include<utility>
using std::pair;
```

Vamos a mencionar otros dos contenedores asociativos, aunque no proporcionaremos ningún detalle acerca de ellos. Las clases de plantilla `multiset` y `multimap` son en esencia las mismas que `set` y `map`, respectivamente, sólo que `multiset` permite la repetición de elementos y `multimap` permite asociar varios valores con cada valor clave.

[1] El ordenamiento debe ser un *ordenamiento débil estricto*. La mayoría de los ordenamientos que se utilizan para implementar el operador `<` son del tipo *ordenamiento débil estricto*. Para aquellos que desean saber los detalles: Un **ordenamiento débil estricto** debe ser: (irreflexivo) `Ordenamiento(x, x)` es siempre falso; (antisimétrico) `Ordenamiento(x, y)` implica `!Ordenamiento(y, x)`; (transitivo) `Ordenamiento(x, y)` y `Ordenamiento(y, z)` implica `Ordenamiento(x, z)`; y (transitividad de equivalencia) si `x` es equivalente a `y`, e `y` es equivalente a `z`, entonces `x` es equivalente a `z`. Dos elementos `x` y `y` son equivalentes si `Ordenamiento(x, y)` y `Ordenamiento(y, x)` son falsos.

CUADRO 18.11 *La clase de plantilla* `Set`

Detalles relacionados con la clase de plantilla `Set`

Escriba el nombre `set<T>` o `set<T, Ordenamiento>` para un conjunto de elementos de tipo `T`.

El `Ordenamiento` se utiliza para ordenar los elementos para su almacenamiento. Si no se proporciona ningún `Ordenamiento`, el orden que se utiliza es el operador binario `<`.

Encabezado de biblioteca: `<set>`, el cual coloca la definición en el espacio de nombres `std`.

Los tipos definidos incluyen: `value_type`, `size_type`.

Iteradores: `iterator`, `const_iterator`, `reverse_iterator` y `const_reverse_iterator`.
Todos los iteradores son bidireccionales y los que no incluyen `const_` son mutables.
`begin()`, `end()`, `rbegin()` y `rend()` tienen el comportamiento esperado.
Las acciones de agregar o eliminar elementos no afectan a los iteradores, excepto un iterador que se ubica en el elemento extraído.

Funciones miembro de ejemplo

Función miembro (`s` es un objeto set)	**Significado**
`s.insert(`*Elemento*`)`	Inserta una copia de *Elemento* en el conjunto. Si *Elemento* ya se encuentra en el conjunto, esta instrucción no tiene efecto.
`s.erase(`*Elemento*`)`	Extrae *Elemento* del conjunto. Si *Elemento* no se encuentra en el conjunto, esta instrucción no tiene efecto.
`s.find(`*Elemento*`)`	Devuelve un iterador mutable que se ubica en la copia de *Elemento* en el conjunto. Si *Elemento* no se encuentra en el conjunto, se devuelve `s.end()`.
`s.erase(`*Iterador*`)`	Elimina el elemento que se encuentra en la ubicación del *Iterador*.
`s.size()`	Devuelve el número de elementos en el conjunto.
`s.empty()`	Devuelve *`true`* si el conjunto está vacío; en caso contrario devuelve *`false`*.
`s1 == s2`	Devuelve *`true`* si el conjunto contiene los mismos elementos; en caso contrario devuelve *`false`*.

La clase de plantilla `set` también tiene un constructor predeterminado, un constructor de copia y otros constructores especializados que no se mencionan aquí. También tiene un destructor que devuelve todo el almacenamiento para su reciclaje y un operador de asignación con un comportamiento correcto.

CUADRO 18.12 *Programa que utiliza la clase de plantilla* `set`

```
//Programa que demuestra el uso de la clase de plantilla set.
#include <iostream>
#include <set>
using std::cout;
using std::endl;
using std::set;
using std::set<char>::const_iterator;

int main( )
{
    set<char> s;

    s.insert('A');
    s.insert('D');
    s.insert('D');
    s.insert('C');
    s.insert('C');
    s.insert('B');

    cout << "El conjunto contiene:\n";
    const_iterator p;
    for (p = s.begin( ); p != s.end( ); p++)
    cout << *p << " ";
    cout << endl;

    cout << "Se elimino C.\n";
    s.erase('C');
    for (p = s.begin( ); p != s.end( ); p++)
    cout << *p << " ";
    cout << endl;

    return 0;
}
```

Sin importar cuántas veces agregue un elemento a un conjunto, éste sólo contiene una copia de ese elemento.

Diálogo de ejemplo

```
El conjunto contiene:
A B C D
Se elimino C.
A B D
```

CUADRO 18.13 *La clase de plantilla* `Map`

Detalles relacionados con la clase de plantilla `map`

Escriba el nombre `map<TipoClave, T>` o `map<TipoClave, T, Ordenamiento>` para un objeto `map` que asocie elementos de tipo `TipoClave` con elementos de tipo `T`.

El `Ordenamiento` se utiliza para ordenar elementos en base al valor clave, para un almacenamiento eficiente. Si no se proporciona un `Ordenamiento`, se utiliza el operador binario `<`.

Encabezado de biblioteca: `<map>`, el cual coloca la definición en el espacio de nombres `std`.

Los tipos definidos incluyen: `key_type` para el tipo de los valores clave; `mapped_type` para el tipo de los valores asignados y `size_type`. (Por lo tanto, el tipo definido `key_type` es en sí lo que antes llamamos `TipoClave`.)

Iteradores: `iterator`, `const_iterator`, `reverse_iterator` y `const_reverse_iterator`. Todos los iteradores son bidireccionales. Los iteradores que no incluyen `cons_` no son constantes ni mutables, sino cierta combinación de ambos. Por ejemplo, si `p` es de tipo `iterator`, puede modificar el valor clave pero no el valor de tipo `T`. Tal vez sea mejor, por lo menos al principio, tratar a todos los iteradores como si fueran constantes. `begin()`, `end()`, `rbegin()` y `rend()` tienen el comportamiento esperado.

La acción de agregar o eliminar elementos no afecta a los iteradores, excepto un iterador ubicado en el elemento extraído.

Funciones miembro de ejemplo

Función miembro (`m` es un objeto map)	**Significado**
`m.insert(`*Elemento*`)`	Inserta *Elemento* en el mapa. *Elemento* es de tipo `pair<TipoClave, T>`. Devuelve un valor de tipo `pair<iterator, bool>`. Si la inserción tiene éxito, la segunda parte del par devuelto es *`true`* y el iterador se ubica en el elemento insertado.
`m.erase(`*Clave_destino*`)`	Extrae el elemento con la clave *`Clave_destino`*.
`m.find(Clave_destino)`	Devuelve un iterador que se ubica en el elemento con el valor clave *`Clave_destino`*. Devuelve `m.end()` si no hay dicho elemento.
`m.size()`	Devuelve el número de pares en el mapa.
`m.empty()`	Devuelve *`true`* si el mapa está vacío; en caso contrario devuelve *`false`*.
`m1 == m2`	Devuelve *`true`* si los mapas contienen los mismos pares; en caso contrario devuelve *`false`*.

La clase de plantilla `map` también tiene un constructor predeterminado, un constructor de copia y otros constructores especializados que no se mencionan aquí. También tiene un destructor que devuelve todo el almacenamiento para su reciclaje y un operador de asignación con un comportamiento correcto.

Eficiencia

La STL se diseñó con una orientación a la eficiencia. De hecho, las implementaciones de la STL se esfuerzan por ser eficientes en forma óptima. Por ejemplo, los elementos `set` y `map` se almacenan en orden, de manera que los algoritmos que busquen los elementos puedan ser más eficientes.

Cada una de las funciones miembro para cada una de las clases de plantilla tienen un tiempo de ejecución máximo garantizado. Estos tiempos máximos de ejecución se expresan mediante el uso de lo que se conoce como notación Big O, la cual veremos en la sección 18.3 (en esta sección también daremos algunos tiempos de ejecución garantizados para algunas de las funciones miembro de los contenedores que ya hemos visto. Estos tiempos aparecen en la subsección titulada "Tiempos de ejecución para el acceso a los contenedores"). Cuando utilice referencias más avanzadas o incluso más adelante en este capítulo, conocerá los tiempos de ejecución máximos garantizados para ciertas funciones.

Ejercicios de AUTOEVALUACIÓN

14. ¿Por qué los elementos en la clase de plantilla `set` se almacenan en orden?

15. ¿Puede un objeto `set` tener elementos de un tipo de clase?

16. Suponga que `s` es del tipo `set<char>`, ¿qué valor devuelve `s.find('A')`, si `'A'` está en `s`? ¿Qué valor se devuelve si `'A'` no está en `s`?

18.3 Algoritmos genéricos

"Cura el consumo, la anemia, la disfunción sexual y todas las demás enfermedades."

Afirmación típica de un vendedor ambulante en relación con el "aceite de víbora"

En esta sección veremos algunas plantillas de funciones básicas en la STL. Aquí no podemos darle una descripción extensiva sobre todas, pero presentaremos una muestra lo bastante extensa como para que se dé una buena idea de lo que contiene la STL y le daremos los detalles suficientes como para que pueda empezar a utilizar estas funciones de plantilla.

algoritmo genérico

A las funciones de plantilla se les conoce algunas veces como **algoritmos genéricos**. El término *algoritmo* se utiliza por una razón. Recuerde que un algoritmo es sólo un conjunto de instrucciones para realizar una tarea. Un algoritmo puede presentarse en cualquier lenguaje, incluyendo un lenguaje de programación como C++. Sólo que, cuando se utiliza la palabra *algoritmo*, por lo general los programadores tienen en mente una presentación menos formal en inglés o en pseudocódigo. Como tal, a menudo se considera como una abstracción del código que define una función. Proporciona los detalles importantes pero no los detalles finos de la codificación. La STL especifica ciertos detalles sobre los algoritmos detrás de las funciones de plantilla de la STL, razón por la cual algunas veces se les llama *algoritmos* genéricos. Estas plantillas de funciones de la STL hacen más que sólo producir un valor en cualquier forma que deseen los implementadores. Las plantillas de funciones en la STL vienen con requerimientos mínimos que deben satisfacer sus implementaciones para poder cumplir con el estándar. En la mayoría de los casos deben implementarse con un tiempo de ejecución garantizado. Esto agrega una dimensión completamente nueva a la

idea de una interfaz de función. En la STL la interfaz no sólo indica a un programador lo que hace la función y cómo utilizarla. La interfaz también indica qué tan rápido se va a realizar la tarea. En algunos casos, el estándar también especifica el algoritmo específico que se utilizará, aunque no el detalle exacto de la codificación. Lo que es más, cuando especifica el algoritmo específico, lo hace debido a la eficiencia conocida del mismo. El nuevo punto clave es una especificación de una garantía de eficiencia para el código. En este capítulo utilizaremos los términos *algoritmo genérico*, *función genérica* y *plantilla de función de la STL* para indicar lo mismo.

Para poder tener cierta terminología para hablar sobre la eficiencia de estas funciones de plantilla o algoritmos genéricos, primero presentaremos ciertos antecedentes en relación con la forma en que se mide por lo general la eficiencia de los algoritmos.

Tiempos de ejecución y notación Big O

Si pregunta a un programador qué tan rápido es su programa, podría esperar una respuesta como "dos segundos". No obstante, la velocidad de un programa no puede darse mediante un solo número. Por lo general un programa requiere una cantidad mayor de tiempo en las entradas extensas que en las entradas más pequeñas. Es de esperarse que un programa que ordena números se tarde menos tiempo en ordenar diez números que lo que se tardaría en ordenar mil números. Tal vez tarde dos segundos en ordenar diez números, pero diez segundos en ordenar mil números. Entonces ¿cómo debería responder el programador a las preguntas sobre qué tan rápido es su programa?

El programador tendría que proporcionar una tabla de valores en la que se mostrara cuánto tiempo tardaría el programa para distintos tamaños de entrada. Por ejemplo, la tabla podría ser como la que se muestra en el cuadro 18.14. Esta tabla no proporciona un tiempo individual, sino distintos tiempos para una variedad de tamaños de entrada diferentes. La tabla es una descripción de lo que en matemáticas se conoce como una **función**. Así como una función de C++ (que no sea `void`) toma un argumento y devuelve un valor, también esta función toma un argumento en forma de un tamaño de entrada y devuelve un número que es el tiempo que tarda el programa en procesar una entrada de ese tamaño. Si llamamos a esta función *T*, entonces *T*(10) es 2 segundos, *T*(100) es 2.1 segundos, *T*(1,000) es 10 segundos y *T*(10,000) es 2.5 minutos. La tabla es sólo un ejemplo de algunos de los valores de esta función *T*. El programa tardará cierta cantidad de tiempo con entradas de todos los tamaños. Por ende y aunque no se muestran en la tabla, también hay valores para *T*(1), *T*(2), ..., *T*(101), *T*(102) y así sucesivamente. Para cualquier entero positivo *N*, *T*(*N*) es la cantidad de tiempo que tarda el programa en ordenar *N* números. La función *T* se conoce como el **tiempo de ejecución** del programa.

función matemática

tiempo de ejecución

CUADRO 18.14 *Algunos valores de una función de tiempo de ejecución*

Tamaño de la entrada	Tiempo de ejecución
10 números	2 segundos
100 números	2.1 segundos
1,000 números	10 segundos
10,000 números	2.5 minutos

Hasta ahora hemos asumido que este programa de ordenamiento tardará la misma cantidad de tiempo con cualquier lista de *N* números. Esto no tiene que ser cierto. Tal vez el programa tarde menos tiempo si la lista ya está ordenada, o casi ordenada. En este caso, *T*(*N*) se define como el tiempo que tarda la lista "más difícil"; es decir, el tiempo que tarda esa lista de *N* números que hace que el programa se ejecute durante más tiempo. A esto se le conoce como el **tiempo de ejecución del peor caso**. En este capítulo *siempre utilizaremos el término tiempo de ejecución del peor caso* cuando proporcionemos un tiempo de ejecución para un algoritmo o para cierto código.

tiempo de ejecución del peor caso

A menudo, el tiempo que tarda un programa o algoritmo se proporciona mediante una fórmula como $4N + 3$, $5N + 4$ o N^2. Si el tiempo de ejecución $T(N)$ es $5N + 5$, entonces con las entradas de tamaño *N* el programa se ejecutará durante $5N + 5$ unidades de tiempo.

A continuación se muestra cierto código para la búsqueda en un arreglo `a` con *N* elementos, de manera que se pueda determinar si un valor específico `destino` se encuentra en el arreglo:

```
int i = 0;
bool encontrado = false;
while (( i < N) && !(encontrado))
     if (a[i] == destino)
         encontrado = true;
     else
        i++;
```

Queremos calcular cierta estimación de cuánto tiempo tardará una computadora en ejecutar este código. Nos gustaría una estimación que no dependa del tipo de computadora que utilicemos, ya sea porque no sabemos cuál computadora utilizaremos o porque podríamos utilizar distintas computadoras para ejecutar el programa en distintos momentos. Una posibilidad es contar el número de "pasos", pero no es fácil decidir qué es un paso. En esta situación lo más común sería contar el número de **operaciones**. El término *operaciones* es casi tan impreciso como el término *paso*, pero por lo menos hay cierto acuerdo en la práctica acerca de lo que califica a una operación. Digamos que, para este código de C++, cada aplicación de cualquiera de los siguientes operadores contará como una operación: =, <, &&, !, [], == y ++. La computadora debe hacer otras cosas además de llevar a cabo estas operaciones, pero parecen ser las cosas principales que está haciendo y asumiremos que se toman en cuenta para el bloque de tiempo requerido para ejecutar este código. De hecho, nuestro análisis de tiempo supondrá que todo lo demás no requiere de tiempo y que el tiempo total para que nuestro programa se ejecute es igual al tiempo necesario para llevar a cabo estas operaciones. Aunque es evidente que esto es una idealización que no es verdadera por completo, resulta ser que esta suposición de simplificación funciona bien en la práctica y se utiliza a menudo cuando se analiza un programa o algoritmo.

operaciones

Incluso hasta con nuestra suposición de simplificación, debemos considerar dos casos: que el valor `destino` se encuentre en el arreglo o no. Consideremos primero el caso en el que `destino` no se encuentra en el arreglo. El número de operaciones realizadas dependerá del número de elementos del arreglo que se analicen en la búsqueda. La operación = se realiza dos veces antes de que se ejecute el ciclo. Como estamos suponiendo que `destino` no se encuentra en el arreglo, el ciclo se ejecutará *N* veces, una para cada elemento del arreglo. Cada vez que se ejecute el ciclo se realizarán las siguientes operaciones: <, &&, !, [], == y ++. Esto agrega 6 operaciones para cada una de las *N* iteraciones del ciclo. Por último, después de *N* iteraciones la expresión Booleana se comprueba de nuevo y resulta ser falsa. Esto agrega tres operaciones finales (<, &&, !).[2] Si llevamos la cuenta de todas estas

[2] Debido a la evaluación de corto circuito, `!(encontrado)` no se evalúa, por lo que en realidad obtenemos 2 operaciones y no 3. No obstante, lo importante es obtener un buen límite superior. Si agregamos una operación adicional, no es algo considerable.

operaciones, obtenemos un total de $6N + 5$ operaciones cuando el `destino` no se encuentra en el arreglo. Dejaremos esto como un ejercicio para que el lector confirme que si el destino se encuentra en el arreglo, entonces el número de operaciones será de $6N + 5$ *o menos*. Por ende, el tiempo de ejecución del peor caso es $T(N) = 6N + 5$ operaciones para cualquier arreglo de N elementos y cualquier valor de `destino`.

Acabamos de determinar que el tiempo de ejecución del peor caso para nuestro código de búsqueda es $6N + 5$ operaciones. Sin embargo, las operaciones no son una unidad tradicional de tiempo, como nanosegundos, segundos o minutos. Si queremos saber cuánto tiempo tardará el algoritmo en cierta computadora específica, debemos saber cuánto tiempo le lleva a esa computadora realizar una operación. Si puede realizarse en un nanosegundo, entonces el tiempo será $6N + 5$ nanosegundos. Si puede realizarse en un segundo, el tiempo será $6N + 5$ segundos. Si utilizamos una computadora lenta que tarde diez segundos en realizar una operación, el tiempo será de $60N + 50$ segundos. En general, si la computadora tarda c nanosegundos en realizar una operación, entonces el tiempo de ejecución actual aproximado será de $c(6N + 5)$ nanosegundos. (Decimos *aproximado* ya que estamos haciendo algunas suposiciones de simplificación y por lo tanto el resultado puede no ser el tiempo de ejecución exacto absoluto.) Esto significa que nuestro tiempo de ejecución de $6N + 5$ es una estimación muy burda. Para expresar el tiempo en nanosegundos debemos multiplicar por cierta constante que depende de la computadora específica que esté utilizando. Nuestra estimación de $6N + 5$ es sólo precisa "dentro de un múltiplo constante". Existe una notación estándar para estos tipos de estimaciones, la cual veremos a continuación.

notación Big O

Por lo general, las estimaciones sobre el tiempo de ejecución, como la que acabamos de ver, se expresan en algo que se conoce como **notación Big O** (es la letra O, no el dígito cero). Suponga que estimamos que el tiempo de ejecución es, por decir, de $6N + 5$ operaciones y suponga que sabemos que sin importar cuál pueda ser el tiempo de ejecución exacto de cada operación distinta, siempre habrá algún factor constante c tal que el tiempo de ejecución real sea menor o igual que

$$c(6N + 5)$$

Bajo estas circunstancias, decimos que el código (programa o algoritmo) se ejecuta en el tiempo $O(6N + 5)$. Por lo general esto se interpreta como "Big O de $6N + 5$". No necesitamos saber cuál será la constante c. De hecho, sin duda será distinta para las distintas computadoras, pero debemos saber que hay una c para cada sistema computacional razonable. Si la computadora es muy veloz, entonces la c podría ser menor que uno, por decir 0.001. Si la computadora es muy lenta, la c podría ser grande, por decir 1000. Además, como para cambiar las unidades, por decir de nanosegundo a segundo, sólo se requiere un múltiplo constante, no hay necesidad de proporcionar unidades de tiempo.

Asegúrese de tener en cuenta que una estimación Big O es una estimación de límite superior. Siempre aproximamos en base a los números en el lado superior, en vez del lado inferior del conteo real. Tenga en cuenta también que, cuando realizamos una estimación Big O, no necesitamos determinar un conteo muy exacto del número de operaciones realizadas. Sólo necesitamos una estimación que sea correcta "hasta un múltiplo constante". Si nuestra estimación es el doble de extensa que el número real, con eso basta.

tamaño de la tarea

Un orden de estimación de magnitud, como el valor anterior $6N + 5$, contiene un parámetro para el tamaño de la tarea resuelta por el algoritmo (programa, o pieza de código). En nuestro caso de ejemplo, este parámetro N era el número de elementos del arreglo en los que se iba a realizar la búsqueda. No es sorpresa que se requiera más tiempo para buscar en un número mayor de elementos de un arreglo que para buscar en un número menor. Las estimaciones de tiempo de ejecución Big O siempre se expresan como una función del tamaño del problema. En este capítulo, todos nuestros algoritmos involucrarán un rango de valores en algún contenedor. En todos los casos N será el número de elementos en ese rango.

La siguiente es una forma alterna y pragmática de pensar acerca de las estimaciones Big O:

Busque sólo en el término con el exponente más alto y no ponga atención a los múltiplos constantes.

Por ejemplo, todas las siguientes son $O(N^2)$:

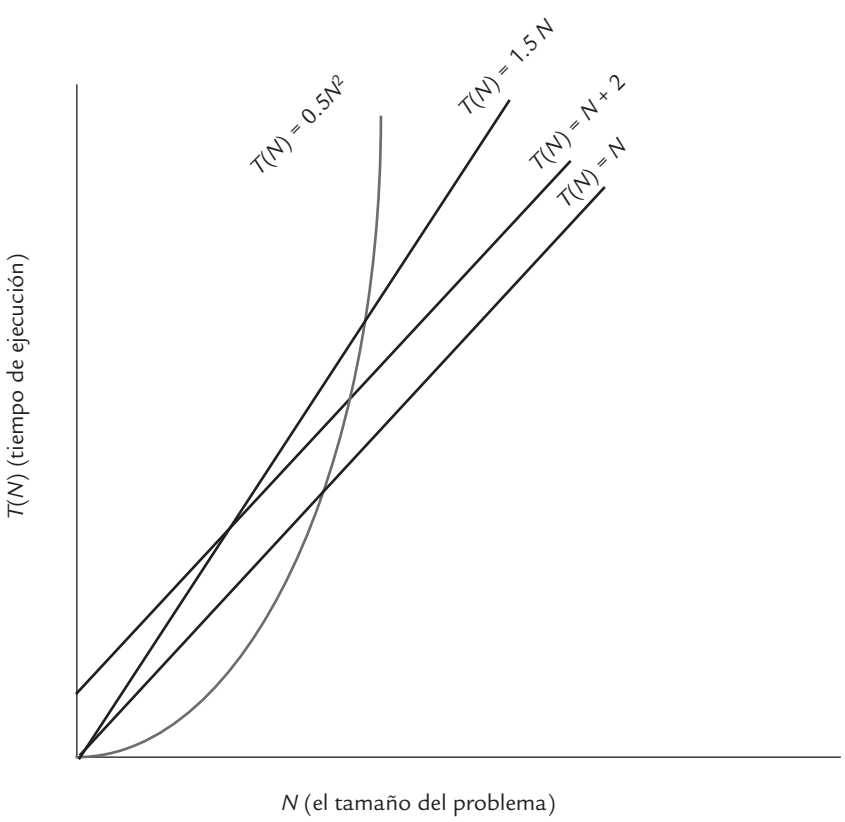
$N^2 + 2N + 1$, $3N^2 + 7$, $100N^2 + N$

Todas las siguientes son $O(N^3)$:

$N^3 + 5N^2 + N + 1$, $8N^3 + 7$, $100N^3 + 4N + 1$

Sin duda, las estimaciones de tiempo de ejecución Big O son burdas, pero contienen cierta información. No diferencian entre un tiempo de ejecución de $5N + 5$ y un tiempo de ejecución de $100N$, pero nos permiten distinguir entre algunos tiempos de ejecución y por consecuencia determinan que algunos algoritmos son más rápidos que otros. Si analiza los gráficos en el cuadro 18.15 podrá ver que todos los gráficos para funciones que sean $O(N)$

CUADRO 18.15 *Comparación de los tiempos de ejecución*



en algún momento quedan debajo del gráfico para la función $0.5N^2$. El resultado es inevitable: un algoritmo $O(N)$ siempre se ejecutará más rápido que cualquier algoritmo $O(N^2)$, siempre y cuando utilicemos valores de N que sean lo bastante grandes. Aunque un algoritmo $O(N^2)$ podría ser más rápido que un algoritmo $O(N)$ para el tamaño del problema que usted maneje, los programadores han encontrado que en la práctica los algoritmos $O(N)$ tienen un mejor rendimiento que los algoritmos $O(N^2)$ para la mayoría de las aplicaciones prácticas que son intuitivamente "extensas". Para otros dos tiempos de ejecución Big O cualesquiera se aplican observaciones similares.

lineal

cuadrático

Cierta terminología ayudará con nuestras descripciones de los tiempos de ejecución de los algoritmos genéricos. **Tiempo de ejecución lineal** significa un tiempo de ejecución de $T(N) = aN + b$. Un tiempo de ejecución lineal siempre es un tiempo de ejecución $O(N)$. **Tiempo de ejecución cuadrático** significa un tiempo de ejecución con el término N^2 más alto. Un tiempo de ejecución cuadrático siempre es un tiempo de ejecución $O(N^2)$. En ocasiones también encontramos logaritmos en las fórmulas de tiempo de ejecución. Estos por lo general se proporcionan sin ninguna base, ya que para cambiar la base sólo se requiere un múltiplo constante. Si ve log N, piense en el logaritmo base 2 de N, pero no estaría mal pensar en el logaritmo base 10 de N. Los logaritmos son funciones con un crecimiento muy lento. Por lo tanto, un tiempo de ejecución nO(log N) es muy rápido.

Tiempos de ejecución del acceso a los contenedores

Ahora que sabemos acerca de la notación Big O, podemos expresar la eficiencia de algunas de las funciones de acceso para las clases contenedoras que vimos en la sección 18.2. Las inserciones en la parte posterior de un objeto `vector` (`push_back`), en la parte frontal o posterior de un objeto `deque` (`push_back` y `push_front`) y en cualquier parte de un objeto `list` (`insert`) son todas $O(1)$ (es decir, un límite superior constante en el tiempo de ejecución, que es independiente del tamaño del contenedor.) La inserción o eliminación de un elemento arbitrario para un objeto `vector` o `deque` es $O(N)$, en donde N es el número de elementos en el contenedor. La búsqueda (`find`) en un objeto `set` o `map` es $O(\log N)$, en donde N es el número de elementos en el contenedor.

Ejercicios de AUTOEVALUACIÓN

17. Demuestre que un tiempo de ejecución $T(N) = aN + b$ es un tiempo de ejecución $O(N)$. *Sugerencia:* la única cuestión es + b. Suponga que N siempre es por lo menos 1.

18. Demuestre que para dos bases a y b cualesquiera para logaritmos, si a y b son mayores que 1, entonces hay una constante c tal que $\log_a N \leq c(\log_b N)$. Por lo tanto, no hay necesidad de especificar una base en $O(\log N)$. Es decir, $O(\log_a N)$ y $O(\log_b N)$ significan lo mismo.

Algoritmos de secuencia no modificadores

En esta sección describiremos las funciones de plantilla que operan sobre contenedores pero no modifican el contenido del contenedor de ninguna forma. Un buen ejemplo simple y común es la función `find` genérica.

La función `find` genérica es similar a la función miembro `find` de la clase de plantilla `set`, pero es una función `find` distinta; en especial, la función `find` genérica recibe más argumentos que la función `find` que vimos cuando presentamos la clase de plantilla `set`. La función `find` genérica busca en un contenedor para ubicar un elemento específico, pero la función `find` genérica puede usarse con cualquiera de las clases contenedoras secuenciales de la STL. El cuadro 18.16 muestra un ejemplo del uso de la función `find` genérica

CUADRO 18.16 *La función* `find` *genérica (parte 1 de 2)*

```
//Programa que demuestra el uso de la función find genérica.
#include <iostream>
#include <vector>
#include <algorithm>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
using std::vector<char>::const_iterator;
using std::find;

int main( )
{
    vector<char> linea;

    cout << "Escriba una linea de texto:\n";
    char siguiente;
    cin.get(siguiente);
    while (siguiente != '\n')
    {
        linea.push_back(siguiente);
        cin.get(siguiente);
    }

    const_iterator donde;
    donde = find(linea.begin( ), linea.end( ), 'e');
    //donde se ubica en la primera ocurrencia de 'e' en v.

    const_iterator p;
    cout << "Escribio lo siguiente antes de escribir su primera e:\n";
    for (p = linea.begin( ); p != donde; p++)
        cout << *p;
    cout << endl;

    cout << "Usted escribio lo siguiente despues de eso:\n";
    for (p = donde; p != linea.end( ); p++)
        cout << *p;
    cout << endl;

    cout << "Fin de la demostracion.\n";
    return 0;
}
```

Si `find` *no encuentra lo que está buscando, devuelve su segundo argumento.*

CUADRO 18.16 *La función* `find` *genérica (parte 2 de 2)*

Diálogo de ejemplo 1

```
Escriba una linea de texto:
Una linea de texto.
Escribio lo siguiente antes de escribir su primera e:
Una lin
Usted escribio lo siguiente despues de eso:
ea de texto.
Fin de la demostracion.
```

Diálogo de ejemplo 2

```
Escriba una linea de texto:
No, jamas!
Escribio lo siguiente antes de escribir su primera e:
No, jamas!
Usted escribio lo siguiente despues de eso:

Fin de la demostracion.
```

Si `find` *no encuentra lo que está buscando, devuelve* `linea.end()`.

que se utiliza con la clase `vector<char>`. La función en el cuadro 18.16 se comportaría de la misma forma si sustituyéramos `vector<char>` con `list<char>` en todo el ejemplo, o si sustituyéramos `vector<char>` con cualquier otra clase contenedora secuencial. Ésta es una de las razones por las que las funciones se llaman *genéricas*. Una definición de la función `find` trabaja para una amplia selección de contenedores.

Si la función `find` no encuentra el elemento que está buscando, devuelve su segundo argumento iterador, el cual no necesita ser igual a `end()` como en el cuadro 18.16. El diálogo de ejemplo 2 muestra la situación cuando `find` no encuentra lo que está buscando.

¿Funciona `find` con todas las clases contenedoras? No, no con todas. Para empezar, toma iteradores como argumentos y ciertos contenedores (como `stack`) no tienen iteradores. Para utilizar la función `find` el contenedor debe tener iteradores, los elementos deben almacenarse en una secuencia lineal de manera que el operador ++ desplace a los iteradores a través del contenedor, y los elementos deben poder compararse mediante el uso de ==. En otras palabras, el contenedor debe tener iteradores de avance (o algún tipo más fuerte de iteradores, como los iteradores bidireccionales).

Cuando presentemos las plantillas de funciones genéricas, describiremos el parámetro del tipo de iterador mediante el uso del nombre del tipo requerido de iterador como el nombre para el parámetro de tipo. Por lo tanto, `ForwardIterator` deberá sustituirse por un tipo que sea un tipo para alguna clase de iterador de avance, como el tipo `iterator` en una clase `list`, `vector` o en cualquier otra clase contenedora de plantilla. Recuerde que un iterador bidireccional es también un iterador de avance, y un iterador de acceso aleatorio

es también un iterador bidireccional. Por ende, el nombre del tipo `ForwardIterator` puede utilizarse con cualquier tipo de iterador que sea del tipo de iterador bidireccional o de acceso aleatorio, así como también un simple tipo de iterador de avance. En algunos casos, cuando especificamos `ForwardIterator` podemos utilizar un tipo de iterador aún más simple; a saber, un iterador de entrada o de salida, pero como no hemos visto estos tipos de iteradores, no los mencionaremos en nuestras declaraciones de funciones de plantilla.

Recuerde que los nombres *iterador de avance*, *iterador bidireccional* e *iterador de acceso aleatorio* se refieren a los tipos de iteradores, no a los nombres de los tipos. Los verdaderos nombres de los tipos serán algo así como `std::vector<int>::iterator`, que en este caso es un iterador de acceso aleatorio.

El cuadro 18.17 muestra un ejemplo de ciertas funciones genéricas no modificadores de la STL; además, utiliza una notación que es común cuando se habla sobre los iteradores de los contenedores. Las ubicaciones de los iteradores que se encuentran cuando se realiza un desplazamiento de un iterador `primero` hacia, pero no igual a, un iterador `ultimo` se conoce como el **rango[primero, ultimo)**. Por ejemplo, el siguiente ciclo *for* muestra todos los elementos en el rango `[primero, ultimo)`:

```
for (iterator p = primero; p != ultimo; p++)
     cout << *p << endl;
```

Tenga en cuenta que cuando se proporcionan dos rangos, no necesitan estar en el mismo contenedor, ni siquiera en el mismo tipo de contenedor. Por ejemplo, para la función `search`, los rangos `[primero1, ultimo1)` y `[primero2, ultimo2)` pueden estar en el mismo contenedor, o en contenedores distintos.

Rango [primero, ultimo)

El movimiento desde un iterador `primero` (que a menudo es `contenedor.begin()`) hasta, pero sin incluir, una ubicación `ultimo` (que a menudo es `contenedor.end()`), es tan común que se le ha asignado un nombre especial, `rango[primero, ultimo)`. Por ejemplo, el siguiente código muestra todos los elementos en el rango `[c.begin(), c.end())`, en donde `c` es algún objeto contenedor, como un vector:

```
for (iterator p = c.begin(); p != c.end(); p++)
    cout << *p << endl;
```

Observe que en el cuadro 18.17 hay tres funciones de búsqueda: `find`, `search` y `binary_search`. La función `search` busca una subsecuencia, mientras las funciones `find` y `binary_search` buscan un valor individual. ¿Cómo decide si va a utilizar `find` o `binary_search` cuando busca un elemento individual? Una devuelve un iterador mientras que la otra sólo devuelve un valor Booleano, pero esa no es la mayor diferencia. La función `binary_ search` requiere que el rango que se busca esté ordenado (en orden ascendente mediante el uso de <) y se ejecuta en el tiempo $O(\log N)$, mientras que la función `find` no requiere que el rango esté ordenado pero garantiza sólo el tiempo lineal. Si tiene o puede tener los elementos en orden, puede buscar en ellos con mucho más rapidez si utiliza `binary_search`.

Tenga en cuenta que con la función `binary_search` se garantiza que la implementación utilizará el algoritmo de búsqueda binaria, que vimos en el capítulo 13. La importancia de utilizar el algoritmo de búsqueda binaria es que garantiza un tiempo de ejecución muy rápido, $O(\log N)$. Si no ha leído el capítulo 13 y no ha escuchado acerca de la búsqueda binaria, sólo considérela como un algoritmo de búsqueda muy eficiente que requiere que los elementos estén ordenados. Esos son los únicos dos puntos sobre la búsqueda binaria que son relevantes para el material en este capítulo.

CUADRO 18.17 *Algunas funciones genéricas no modificadoras*

Todas funcionan para los iteradores de avance, lo cual significa que también funcionan para los iteradores bidireccionales y de acceso aleatorio. (En algunos casos funcionan también para otros tipos de iteradores, que no cubriremos aquí.)

```
template <class ForwardIterator, class T>
ForwardIterator find(ForwardIterator primero,
                                    ForwardIterator ultimo, const T& destino);
//Recorre el rango [primero, ultimo) y devuelve un iterador ubicado en
//la primera ocurrencia de destino. Devuelve segundo si no se encuentra destino.
//Complejidad de tiempo: lineal en el tamaño del rango [primero, segundo).

template<class ForwardIterator, class T>
int[3] count(ForwardIterator primero, ForwardIterator ultimo, const T& destino);
//Recorre el rango [primero, ultimo) y devuelve el número
//de elementos iguales a destino.
//Complejidad de tiempo: lineal en el tamaño del rango [primero, ultimo).

template <class ForwardIterator1, class ForwardIterator2>
bool equal(ForwardIterator1 primero1, ForwardIterator2 ultimo 1,
                                   ForwardIterator2 primero2);
//Devuelve true si [primero1, ultimo 1) contiene los mismos elementos en el mismo
//orden que los primeros elementos ultimo1-primero1 que empiezan en primero2.
//En caso contrario devuelve false.
//Complejidad de tiempo: lineal en el tamaño del rango [primero, ultimo).

template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(ForwardIterator1 primero1, ForwardIterator1 ultimo1,
                     ForwardIterator2 primero2, ForwardIterator2 ultimo2);
//Comprueba si [primero2, ultimo2) es un subrango de [primero1, ultimo1).
//De ser así, devuelve un iterador ubicado en [primero1, ultimo1) al principio de
//la primera coincidencia. Devuelve ultimo1 si no se encuentra una coincidencia.
//Complejidad de tiempo: cuadrática en el tamaño del rango [primero1, ultimo1).

template<class ForwardIterator,  class T>
bool binary_search(ForwardIterator primero, ForwardIterator ultimo, const T& destino);
//Precondición: el rango [primero, ultimo) se ordena en forma ascendente mediante <.
//Utiliza el algoritmo de búsqueda binaria para determinar si el destino se encuentra
//en el rango [primero, ultimo).
//Complejidad de tiempo: para los iteradores de acceso aleatorios O(log N). Para los
//iteradores que no son de acceso aleatorio lineal es N, en donde N es el tamaño
//del rango [primero, ultimo).
```

[3] El tipo de retorno actual es un tipo entero que no hemos visto, pero el valor de retorno debe poder asignarse a una variable de tipo `int`.

Ejercicios de AUTOEVALUACIÓN

19. Sustituya todas las ocurrencias del identificador `vector` con el identificador `list` en el cuadro 18.16. Compile y ejecute el programa.

20. Suponga que `v` es un objeto de la clase `vector<int>`. Utilice la función genérica `search` (cuadro 18.17) para escribir cierto código para determinar si `v` contiene o no el número `42` que está en la posición inmediata anterior a `43`. No necesita escribir un programa completo; sólo proporcione todas las directivas `include` y `using` necesarias. *Sugerencia:* puede ser conveniente utilizar un segundo vector.

Algoritmos modificadores de contenedores

El cuadro 18.18 contiene descripciones de algunas de las funciones genéricas en la STL, las cuales modifican el contenido de un contenedor de alguna forma.

Recuerde que cuando agrega o extrae un elemento a (o de) un contenedor, esa operación puede afectar a cualquiera de los otros iteradores. No hay garantía de que los iteradores se encontrarán en el mismo elemento después de una acción de agregar o eliminar, a menos que la clase contenedora de plantilla haga dicha garantía. De las clases de plantilla que hemos visto, `list` y `slist` garantizan que sus iteradores no se desplazarán mediante operaciones de agregar o eliminar, excepto desde luego si el iterador se ubica en un elemento que sea extraído. Las clases de plantilla `vector` y `deque` no hacen dicha garantía. Algunas de las plantillas de funciones en el cuadro 18.18 garantizan los valores de ciertos iteradores específicos y podemos, desde luego, contar con esas garantías sin importar cuál sea el contenedor.

Ejercicios de AUTOEVALUACIÓN

21. ¿Puede utilizar la función de plantilla `random_shuffle` con un contenedor `list`?

22. ¿Puede utilizar la función de plantilla `copy` con contenedores `vector`, aun cuando `copy` requiere iteradores de avance y `vector` tiene iteradores de acceso aleatorio?

Algoritmos de conjuntos

El cuadro 18.19 muestra un ejemplo de las funciones genéricas de operación de conjuntos definidas en la STL. Observe que estos algoritmos genéricos asumen que los contenedores almacenan sus elementos en orden. Los contenedores `set`, `map`, `multiset` y `multimap` almacenan sus elementos en orden; por lo tanto, todas las funciones en el cuadro 18.19 se aplican a estos cuatro contenedores de clase de plantilla. Otros contenedores como `vector` no almacenan sus elementos en orden, por lo que estas funciones no deben utilizarse con dichos contenedores. La razón de requerir que los elementos estén ordenados es para que los algoritmos puedan ser más eficientes.

CUADRO 18.18 *Algunas funciones genéricas modificadoras*

```
template <class T>
void swap(T& variable1, T& variable2);
//Intercambia los valores de variable1 y variable2
```

El nombre del parámetro del tipo de iterador indica el tipo de iterador para el que trabaja la función. Recuerde que estos son requerimientos mínimos del iterador. Por ejemplo, `ForwardIterator` *funciona para los iteradores de avance, los iteradores bidireccionales y los iteradores de acceso aleatorio.*

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 copy(ForwardIterator1 primero1, ForwardIterator1 ultimo1,
                      ForwardIterator2 primero2, ForwardIterator2 ultimo2);
//Precondición: Los rangos [primero1, ultimo1) y [primero2, ultimo2)
//son del mismo tamaño.
//Acción: copia los elementos de las ubicaciones [primero1, ultimo1) a las
//ubicaciones [primero2, ultimo2)).
//Devuelve ultimo2.
//Complejidad de tiempo: lineal en el tamaño del rango [primero1, ultimo1).

template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator primero, ForwardIterator ultimo,
                                                const T& destino);
//Extrae del rango [primero, ultimo) los elementos iguales a destino. El tamaño del
//contenedor no se modifica. Los valores extraídos que sean iguales a destino se
//mueven hacia el final del rango [primero, ultimo). Hay entonces un iterador i
//en este rango, de tal forma que los valores que no sean iguales a destino estén
//en [primero, i). Esta i se devuelve.
//Complejidad de tiempo: lineal en el tamaño del rango [primero, ultimo).

template<class BidirectionalIterator>
void reverse(BidirectionalIterator primero, BidirectionalIterator ultimo);
//Invierte el orden de los elementos en el rango [primero, ultimo).
//Complejidad de tiempo: lineal en el tamaño del rango [primero, ultimo).

template<class RandomAccessIterator>
void random_shuffle(RandomAccessIterator primero, RandomAccessIterator ultimo);
//Utiliza un generador de números pseudo-aleatorio para reordenar en forma
//aleatoria los elementos en el rango [primero, ultimo).
//Complejidad de tiempo: lineal en el tamaño del rango [primero, ultimo).
```

CUADRO 18.19 *Operaciones de conjuntos*

Estas operaciones funcionan para objetos `set`, `map`, `multiset`, `multimap` *(y demás contenedores) pero no funcionan para todos los contenedores. Por ejemplo, no funcionan para objetos* `vector`, `list` *o* `deque` *a menos que su contenido esté ordenado. Para que estos contenedores funcionen, los elementos en el contenedor deben almacenarse en orden. Todos estos operadores funcionan para los iteradores de avance, lo cual significa que también trabajan para los iteradores bidireccionales y de acceso aleatorio. (En algunos casos funcionan también para otros tipos de iteradores que no hemos visto en este libro.)*

```
template<class ForwardIterator1, class ForwardIterator2>
bool includes(ForwardIterator1 primero1, ForwardIterator1 ultimo1,
              ForwardIterator2 primero2, ForwardIterator2 ultimo2);
//Devuelve true si cada elemento en el rango [primero2, ultimo2) también se
//encuentra en el rango [primero1, ultimo1). En caso contrario devuelve false.
//Complejidad de tiempo: lineal en el tamaño de [primero1, utlimo1)
//más [primero2, ultimo2).

template<class ForwardIterator1, class ForwardIterator2, class ForwardIterator3>
void set_union(ForwardIterator1 primero1, ForwardIterator ultimo1,
               ForwardIterator2 primero2, ForwardIterator2 ultimo2,
                                          ForwardIterator3 resultado);
//Crea una unión ordenada de los dos rangos [primero1, ultimo1) y [primero2, ultimo2).
//La unión se ordena empezando en resultado.
//Complejidad de tiempo: lineal en el tamaño de [primero1, ultimo1)
//más [primero2, ultimo2).

template<class ForwardIterator1, class ForwardIterator2, class ForwardIterator3>
void set_intersection(ForwardIterator1 primero1, ForwardIterator1 ultimo1,
                      ForwardIterator2 primero2, ForwardIterator2 ultimo2,
                                                 ForwardIterator3 resultado);
//Crea una intersección ordenada de los dos rangos [primero1, ultimo1)
//y [primero2, ultimo2).
//La intersección se ordena empezando en resultado.
//Complejidad de tiempo: lineal en el tamaño de [primero1, ultimo1)
//más [primero2, ultimo2).

template<class ForwardIterator1, class ForwardIterator2, class ForwardIterator3>
void set_difference(ForwardIterator1 primero1, ForwardIterator ultimo1,
            ForwardIterator2 primero2, ForwardIterator2 ultimo2,
                                       ForwardIterator3 resultado);
//Crea una diferencia de conjuntos ordenados de los dos rangos [primero1, ultimo1)
//y [primero2, ultimo2).
//La diferencia consiste de los elementos en el primer rango que no están en el segundo.
//El resultado se almacena empezando en resultado.
//Complejidad de tiempo: lineal en el tamaño de [primero1, ultimo1)
//más [primero2, ultimo2).
```

Ejercicios de AUTOEVALUACIÓN

23. La versión de un curso de matemáticas de un conjunto no mantiene sus elementos en orden y tiene un operador de unión. ¿Por qué la función de plantilla `set_union` requiere que los contenedores mantengan sus elementos en orden?

Algoritmos de ordenamiento

El cuadro 18.20 proporciona las declaraciones y la documentación para dos funciones de plantilla, una para ordenar un rango de elementos y una para mezclar dos rangos ordenados de elementos. Observe que la función de ordenamiento `sort` garantiza un tiempo de ejecución de $O(N \log N)$. Aunque está más allá del alcance de este libro, puede demostrarse que no se puede escribir un algoritmo de ordenamiento que sea más rápido que $O(N \log N)$. Por lo tanto, esto garantiza que el algoritmo de ordenamiento sea lo más rápido posible, hasta un múltiplo constante.

CUADRO 18.20 *Algunos algoritmos de ordenamiento genérico*

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator primero, RandomAccessIterator ultimo);
//Ordena los elementos en el rango [primero, ultimo) en orden ascendente.
//Complejidad de tiempo: O(N log N) en donde N es el tamaño del rango [primero, ultimo).

template<class ForwardIterator1, class ForwardIterator2, class ForwardIterator3>
void merge(ForwardIterator1 primero1, ForwardIterator1 ultimo1,
                   ForwardIterator2 primero2, ForwardIterator2 ultimo2,
                   ForwardIterator3 resultado);
//Precondición: Los rangos [primero1, ultimo1) y [primero2, ultimo2)
//están ordenados.
//Acción: mezcla los dos rangos en un rango ordenado [resultado, ultimo3) en donde
//ultimo3 = resultado + (ultimo1 – primero1) + (ultimo2 – primero2).
//Complejidad de tiempo: lineal en el tamaño del rango [primero1, ultimo1)
//más el tamaño de [primero2, ultimo2).
```

El ordenamiento utiliza el operador <, por lo cual este operador debe estar definido. Hay otras versiones (que no se muestran aquí) que le permiten proporcionar la relación de ordenamiento. Ordenado significa en orden ascendente.

Resumen del capítulo

- Un iterador es la generalización de un apuntador. Los iteradores se utilizan para desplazarse a través de los elementos en cierto rango de un contenedor. Por lo general se definen las operaciones ++, −− y el operador de desreferencia * para un iterador.

- Las clases contenedoras con iteradores tienen las funciones miembro `end()` y `begin()`, las cuales devuelven valores de iteradores de tal forma que se puedan procesar todos los datos en el contenedor de la siguiente manera:

```
for (p = c.begin(); p != c.end(); p++)
     procesa p  //*p es el elemento de datos actual.
```

- Los tipos principales de iteradores son

  Iteradores de avance: ++ trabaja con el iterador.

  Iteradores bidireccionales: tanto ++ como −− trabajan con el iterador.

  Iteradores de acceso aleatorio: ++, −− y el acceso aleatorio trabajan con el iterador.

- Con un iterador `p` constante, el operador de desreferencia `*p` produce una versión de sólo lectura del elemento. Con un iterador `*p` mutable, se puede asignar un valor a `*p`.

- Un contenedor bidireccional tiene iteradores inversos que permiten que el código itere a través de los elementos en el contenedor, en orden inverso.

- Las principales clases contenedoras de plantilla en la STL son `list`, que tiene iteradores bidireccionales mutables, y las clases de plantilla `vector` y `deque`, las cuales tienen iteradores de acceso aleatorio mutables.

- `stack` y `queue` son clases adaptadoras de contenedores, lo cual significa que se crean encima de otras clases contenedoras. Un objeto `stack` es un contenedor del tipo "último en entrar/primero en salir". Un objeto `queue` es un contenedor del tipo "primero en entrar/primero en salir".

- Las clases contenedoras de plantilla `set`, `map`, `multiset` y `multimap` almacenan sus elementos en orden para la eficiencia de los algoritmos de búsqueda. Un objeto `set` es una colección simple de elementos. Un objeto `map` permite almacenar y recuperar elementos en base a valores clave. La clase `multiset` permite la repetición de entradas. La clase `multimap` permite asociar una sola clave con varios elementos de datos.

- La STL incluye funciones de plantilla para implementar algoritmos genéricos con garantías sobre su tiempo de ejecución máximo.

Respuestas a los ejercicios de autoevaluación

1. `v.begin()` devuelve un iterador que se ubica en el primer elemento de `v`. `v.end()` devuelve un valor que sirve como valor centinela al final de todos los elementos de `v`.

2. `*p` es el operador de desreferencia aplicado a `p`. `*p` es una referencia al elemento que está en la ubicación `p`.

3.
```
using std::vector<int>::iterator;
   . . .
iterator p;
```

```
for (p = v.begin(), p++; p != v.end(); p++)
     cout << *p << " ";
```

4. D C C

5. B C

6. Ambas funcionarían.

7. Una diferencia principal es que un contenedor `vector` tiene iteradores de acceso aleatorio, mientras que un contenedor `list` sólo tiene iteradores bidireccionales.

8. Todas excepto `slist`.

9. `vector` y `deque`.

10. Todas pueden tener iteradores mutables.

11. La clase de adaptador de plantilla `stack` no tiene iteradores.

12. La clase de adaptador de plantilla `queue` no tiene iteradores.

13. No se devuelve un valor; `pop` es una función `void`.

14. Para facilitar una búsqueda de elementos eficiente.

15. Sí, pueden ser de cualquier tipo, aunque sólo hay un tipo para cada objeto `set`. El parámetro de tipo en la clase de plantilla es el tipo de los elementos almacenados.

16. Si `'A'` está en `s`, entonces `s.find('A')` devuelve un iterador ubicado en el elemento `'A'`. Si `'A'` no está en `s`, entonces `s.find('A')` devuelve `s.end()`.

17. Sólo tenga en cuenta que $aN + b \leq (a + b)N$, siempre y cuando $1 \leq N$.

18. Esto es matemáticas, no C++. Por lo tanto, = significa *igual que*, no una asignación.

    Primero observe que $\log_a N = (\log_a b)(\log_b N)$.

    Para ver esta primera identidad sólo tome en cuenta que si eleva a a la potencia $\log_a N$ obtendrá N, y si eleva a a la potencia $(\log_a b)(\log_b N)$ también obtendrá N.

    Si hace que $c = (\log_a b)$ obtendrá $\log_a N = c(\log_b N)$.

19. Los programas deben ejecutarse de la misma forma exacta.

20.
```
#include <iostream>
#include <vector>
#include <algorithm>
using std::cout;
using std::vector;
using std::vector<int>::const_iterator;
using std::search;
 . . .
vector<int> destino;
destino.push_back(42);
destino.push_back(43);
```

```
const_iterator resultado = search(v.begin(), v.end(),
                                  destino.begin(), destino.end());
if (resultado != v.end())
    cout << "Se encontro 42, 43.\n";
else
    cout << "42, 43 no estan ahí.\n";
```

21. No, debe tener iteradores de acceso aleatorio, y la clase de plantilla `list` sólo tiene iteradores bidireccionales.

22. Sí, un iterador de acceso aleatorio es también un iterador de avance.

23. La función de plantilla `set_union` requiere que los contenedores mantengan sus elementos en orden para permitir que la plantilla de función se implemente de una manera más eficiente.

Proyectos de programación

1. Escriba un programa en el que declare un objeto `deque` para almacenar valores de tipo *`double`*, que lea 10 números *`double`* y los almacene en el objeto `deque`. Después llame a la función genérica `sort` para ordenar los números en el objeto `deque` y muestre los resultados.

2. Escriba un programa que permita que el usuario escriba cualquier cantidad de nombres de estudiantes y sus calificaciones. El programa deberá entonces mostrar los nombres de los estudiantes y las calificaciones de acuerdo con el orden ascendiente de éstas. Use la clase de plantilla `vector` y la función genérica `sort` de la STL. Tome en cuenta que tendrá que definir un tipo de estructura o de clase para los datos que consistan de más de un nombre de estudiante y una calificación. También necesitará sobrecargar el operador `<` para esta estructura o clase.

3. 

Un número **primo** es un entero mayor que 1 y puede dividirse sólo entre sí mismo y 1. Un entero `x` es **divisible** entre un entero `y` si existe otro entero `z` tal que `x = y * z`. El matemático griego Eratóstenes escribió un algoritmo para encontrar todos los números primos menores que cierto entero *N*. A este algoritmo se le conoce como la *Criba de Eratóstenes* y funciona de la siguiente manera: se comienza con una lista de enteros del 2 al *N*. El número 2 es el primer número primo (es instructivo considerar por qué es verdad esto). Los *múltiplos* de 2 (es decir, 4, 6, 8, etc.) *no son primos*. Quitaremos estos números de la lista. Ahora, el primer número después de 2 que no se haya eliminado de la lista será el siguiente primo. Este número es 3. Los *múltiplos de 3 no son números primos*. Elimine los múltiplos de 3 de la lista. Observe que el 6 ya se ha eliminado; elimine el 9, el 12 ya está eliminado, elimine el 15, etc. El primer número que no se elimine será el siguiente número primo. El algoritmo continuará de esta forma hasta llegar a *N*. Todos los números que no se eliminen de la lista serán números primos.

   a) Escriba un programa mediante el uso de este algoritmo para encontrar todos los números primos menores que el número *N* suministrado por el usuario. Use un contenedor tipo vector para los enteros. Utilice un arreglo de valores *`bool`* en el que al principio se asigne *`true`* a todos sus elementos, para llevar la cuenta de los enteros eliminados. Cambie la entrada a *`false`* para los enteros que se eliminen de la lista.

   b) Pruebe para *N* = 10, 30, 100 y 300.

   Mejoras:

   c) En realidad, no necesitamos llegar hasta *N*. Podemos detenernos en *N/2*. Pruebe esto y evalúe su programa. *N/2* funciona y es mejor, pero no es el número más pequeño que podríamos utilizar. Argumente que para obtener todos los números primos entre 1 y *N*, el límite mínimo es la raíz cuadrada de *N*.

   d) Modifique su código de la parte (a) para utilizar la raíz cuadrada de *N* como límite superior.

4. Suponga que tiene una colección de registros de estudiantes. Los registros son estructuras del siguiente tipo:

```
struct InfoEstudiante
{
     string nombre;
     int calif;
};
```

Los registros se mantienen en un objeto `vector<InfoEstudiante>`. Escriba un programa que pida y obtenga datos, y cree un vector de registros de estudiantes, después que ordene el vector por nombre, calcule las calificaciones mínima y máxima, el promedio de la clase, y que después imprima estos datos de resumen junto con un cuadro de honor con las calificaciones. (No nos interesa quién tuvo la calificación menor y quién la mayor, sólo nos interesan las estadísticas de valor máximo, mínimo y promedio.) Pruebe su programa.

5. Continúe el proyecto de programación 4; escriba una función que separe los estudiantes del vector `InfoEstudiante` en dos vectores, uno que contenga registros de los estudiantes aprobados y otro para los reprobados. (Utilice una calificación de 60 o mejor para pasar.)

Le pediremos que haga esto de dos formas y que proporcione algunas estimaciones del tiempo de ejecución.

a) Considere seguir usando un vector. Podría generar un segundo vector de estudiantes aprobados y un tercer vector de estudiantes reprobados. Esto mantiene registros duplicados durante al menos cierta parte del tiempo, por lo que no es conveniente hacerlo así. Podría crear un vector de estudiantes reprobados y una función de prueba por fallar. Entonces podría usar la función `push_back` para meter de vuelta los registros de los estudiantes fracasados y después eliminar del vector original mediante `erase` (que es una función miembro) los registros del estudiante reprobado. Escriba el programa de esta forma.

b) Considere la eficiencia de esta solución. Existe la posibilidad de borrar miembros $O(N)$ de la parte media de un vector. Tiene que desplazar muchos miembros en este caso. La operación de borrar desde la parte media de un vector es una operación $O(N)$. Proporcione una estimación Big O del tiempo de ejecución para este programa.

c) Si utilizó un objeto `list<InfoEstudiante>`, ¿cuáles son los tiempos de ejecución para las funciones `erase` e `insert`? Considere cómo la eficiencia en tiempo de `erase` para un objeto `list` afecta al tiempo de ejecución para el programa. Vuelva a escribir este programa, pero esta vez utilice un objeto `list` en vez de un objeto `vector`. Recuerde que un objeto `list` no cuenta con indización ni con acceso aleatorio; además sus iteradores sólo son bidireccionales, no de acceso aleatorio.

Apéndice 1
Palabras clave de C++

Las siguientes palabras clave deberán utilizarse sólo para propósitos predefinidos en el lenguaje C++; en especial, no deberán utilizarse para nombres de variables o funciones definidas por el programador. Además de las palabras que se listan a continuación, los identificadores que contienen un guión bajo doble (_ _) están reservados para que los utilicen las implementaciones de C++ y las bibliotecas estándar, por lo tanto usted no deberá utilizarlos en sus programas.

```
asm          do           inline              return       typedef
auto         double       int                 short        typeid
bool         dynamic_cast log                 signed       typename
break        else         long                sizeof       union
case         enum         mutable             static       unsigned
catch        explicit     namespace           static_cast  using
char         extern       new                 struct       virtual
class        false        operator            switch       void
const        float        private             template     volatile
const_cast   for          protected           this         wchar_t
continue     friend       public              throw        while
default      goto         register            true
delete       if           reinterpret_cast    try
```

Estas representaciones alternativas para los operadores y signos de puntuación están reservadas y tampoco deben utilizarse.

```
and &&       and_eq &=   bitand &    bitor |    compl ~     not  !
not_eq !=    or ||       or_eq |=    xor ^      xor_eq ^=
```

Apéndice 2
Precedencia de operadores

Todos los operadores de un mismo cuadro tienen la misma precedencia. Los operadores en los cuadros superiores tienen mayor precedencia que los operadores en los cuadros inferiores. Los operadores unarios y el operador de asignación se ejecutan de derecha a izquierda cuando tienen la misma precedencia. Por ejemplo, `x = y = z` significa `x = (y = z)`. Otros operadores que tengan las mismas precedencias se ejecutarán de izquierda a derecha. Por ejemplo, `x + y + z` significa `(x + y) + z`.

::	operador de resolución de alcance
.	operador punto
−>	selección de miembro
[]	indización de arreglos
( )	llamada a función
++	operador de incremento posfijo (se coloca después de la variable)
−−	operador de decremento posfijo (se coloca después de la variable)
++	operador de incremento prefijo (se coloca antes de la variable)
−−	operador de decremento prefijo (se coloca antes de la variable)
!	negación
−	unario negativo
+	unario positivo
*	desreferencia
&	dirección de
new	
delete	
delete[]	
sizeof	
*	multiplicación
/	división
%	residuo (módulo)
+	suma
−	resta
<<	operador de inserción (salida)
>>	operador de extracción (entrada)
<	menor que <= menor o igual que
>	mayor que >= mayor o igual que
==	igual a
!=	no igual a
&&	y
||	o
=	asignación
+=	suma y asignación
−=	resta y asignación
*=	multiplicación y asignación
/=	división y asignación
%=	módulo y asignación

Mayor precedencia (se ejecuta primero)

Menor precedencia (se ejecuta al último)

Apéndice 3
El conjunto de caracteres ASCII

Sólo se muestran los caracteres imprimibles. El carácter número 32 es el espacio en blanco.

32		56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	O	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	—	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	—	119	w
48	0	72	H	96	‘	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g		

Apéndice 4
Algunas funciones de biblioteca

Las siguientes listas están organizadas de acuerdo con el propósito para el que se utiliza la función, en lugar de ordenarlas de acuerdo a la biblioteca en la que se encuentra. La declaración de la función proporciona el número y los tipos de los argumentos, así como el tipo de valor devuelto. En la mayoría de los casos, las declaraciones de las funciones sólo proporcionan el tipo de parámetro y no dan un nombre para el mismo. (En la sección "Forma alterna para declarar funciones" del capítulo 3 podrá consultar una explicación sobre este tipo de declaración de funciones.)

Funciones aritméticas

Declaración de la función	Descripción	Archivo de encabezado
`int abs (int);`	Valor absoluto	`cstdlib`
`long labs (long);`	Valor absoluto	`cstdlib`
`double fabs (double);`	Valor absoluto	`cmath`
`double sqrt (double);`	Raíz cuadrada	`cmath`
`double pow (double, double);`	Devuelve el primer argumento elevado a la potencia de su segundo argumento	`cmath`
`double exp (double);`	Devuelve e (base del logaritmo natural) elevado a la potencia de su argumento	`cmath`
`double log (double);`	Logaritmo natural (ln)	`cmath`
`double log10 (double);`	Logaritmo base 10	`cmath`
`double ceil (double);`	Devuelve el entero más pequeño que sea mayor o igual que su argumento	`cmath`
`double floor (double);`	Devuelve el entero más grande que sea menor o igual que su argumento	`cmath`

Funciones miembro de entrada y salida

Formato de la llamada a una función	Descripción	Archivo de encabezado
Var_flujo.open (*Nombre_archivo_externo*);	Conecta el archivo con el *Nombre_archivo_externo* al flujo nombrado por *Var_flujo*. El *Nombre_archivo_externo* es un valor tipo cadena de texto.	fstream
Var_flujo.fail( );	Devuelve *true* si falla la operación anterior (como una apertura) sobre el flujo *Var_flujo*.	fstream o iostream
Var_flujo.close( );	Desconecta el flujo *Var_flujo* del archivo al que está conectado.	fstream
Var_flujo.bad( );	Devuelve *true* si el flujo *Var_flujo* está corrupto.	fstream o iostream
Var_flujo.eof( );	Devuelve *true* si el programa trata de leer más allá del último carácter en el archivo conectado al flujo de entrada *Var_flujo*. En caso contrario, devuelve *false*.	fstream o iostream
Var_flujo.get(*Variable_char*);	Lee un carácter del flujo de entrada *Var_flujo* y hace que *Variable_char* sea igual a este carácter. *No* omite el espacio en blanco.	fstream o iostream
Var_flujo.getline (*Var_cadena*, *Caracteres_Max* +1);	Se lee una línea de entrada del flujo *Var_flujo* y la cadena resultante se coloca en *Var_cadena*. Si la línea tiene más de *Caracteres_Max*, sólo se leen los primeros *Caracteres_Max*. El tamaño declarado de la *Var_cadena* debe ser *Caracteres_Max* +1 o mayor.	fstream o iostream
Var_flujo.peek( );	Lee un carácter del flujo de entrada *Var_flujo* y devuelve ese carácter. Pero la lectura del carácter *no* se elimina del flujo de entrada; la siguiente lectura leerá el mismo carácter.	fstream o iostream
Var_flujo.put(*Exp_char*);	Escribe el valor de la *Exp_char* en el flujo de salida *Var_flujo*.	fstream o iostream

Funciones miembro de entrada y salida (*continuación*)

Formato de la llamada a una función	Descripción	Archivo de encabezado
Var_flujo.putback(*Exp_char*);	Coloca el valor de *Exp_char* en el flujo de entrada *Var_flujo*, para que ese valor sea el siguiente valor de entrada que se lea del flujo. El archivo conectado al flujo no se modifica.	fstream o iostream
Var_flujo.precision(*Exp_int*);	Especifica el número de dígitos a enviar después del punto decimal, para los valores de punto flotante que se envían al flujo de salida *Var_flujo*.	fstream o iostream
Var_flujo.width(*Exp_int*);	Establece la anchura del campo para el siguiente valor a enviar al flujo *Var_flujo*.	fstream o iostream
Var_flujo.setf(*Bandera*);	Activa las banderas para dar formato a la salida que va al flujo *Var_flujo*. En el cuadro 5.5 podrá consultar la lista de posibles banderas.	fstream o iostream
Var_flujo.unsetf(*Bandera*);	Borra las banderas para dar formato a la salida que va al flujo *Var_flujo*. En el cuadro 5.5 podrá consultar la lista de posibles banderas.	fstream o iostream

Funciones de caracteres

Para todas estas funciones el tipo del argumento es *int*, pero para la mayoría de los fines podemos considerar el tipo del argumento como si fuera *char*. Si el valor devuelto es de tipo *int*, debemos realizar una conversión de tipos explícita o implícita para obtener un *char*.

Declaración de la función	Descripción	Archivo de encabezado
`bool isalnum(char);`	Devuelve *true* si su argumento satisface a `isalpha` o a `isdigit`. En caso contrario, devuelve *false*.	`cctype`
`bool isalpha(char);`	Devuelve *true* si su argumento es una letra mayúscula o minúscula. También puede devolver *true* para otros argumentos. Los detalles son dependientes de la implementación. En caso contrario, devuelve *false*.	`cctype`
`bool isdigit(char);`	Devuelve *true* si su argumento es un dígito. En caso contrario, devuelve *false*.	`cctype`
`bool ispunct(char);`	Devuelve *true* si su argumento es un carácter imprimible que no satisface a `isalnum` y no es espacio en blanco. (Estos caracteres se consideran como caracteres de puntuación.) En caso contrario, devuelve *false*.	`cctype`
`bool isspace(char);`	Devuelve *true* si su argumento es un carácter de espacio en blanco (como un espacio, una tabulación o una nueva línea). En caso contrario, devuelve *false*.	`cctype`
`bool iscntrl(char);`	Devuelve *true* si su argumento es un carácter de control. En caso contrario, devuelve *false*.	`cctype`
`bool islower(char);`	Devuelve *true* si su argumento es una letra minúscula. En caso contrario, devuelve *false*.	`cctype`
`bool isupper(char);`	Devuelve *true* si su argumento es una letra mayúscula. En caso contrario, devuelve *false*.	`cctype`
`int tolower(char);`	Devuelve la versión en minúsculas de su argumento. Si no hay una versión en minúsculas, devuelve su argumento sin modificarlo.	`cctype`
`int toupper(char);`	Devuelve la versión en mayúsculas de su argumento. Si no hay una versión en mayúsculas, devuelve su argumento sin modificarlo.	`cctype`

Funciones de cadenas

Declaración de la función	Descripción	Archivo de encabezado
`int atoi(const char a[]);`	Convierte una cadena de caracteres en un entero.	`cstdlib`
`long atol(const char a[]);`	Convierte una cadena de caracteres en un entero `long`.	`cstdlib*`
`double atof(const char a[]);`	Convierte una cadena de caracteres en un `double`.	`cstdlib`
`strcat(`*Variable_cadena*, *Expresion_cadena*`);`	Adjunta el valor de la *Expresion_cadena* al final de la cadena en la *Variable_cadena*	`cstring`
`strcmp(`*Exp_cadena1*, *Exp_cadena2*`);`	Devuelve *true* si los valores de las dos expresiones de cadena son distintas; en caso contrario devuelve `false`.†	`cstring`
`strcpy(`*Variable_cadena*, *Expresion_cadena*`);`	Cambia el valor de la *Variable_cadena* por el valor de la *Expresion_cadena*.	`cstring`
`strlen(`*Expresion_cadena*`)`	Devuelve la longitud de la *Expresion_cadena*.	`cstring`
strncat(*Variable_cadena, Expresion_cadena*, *Límite*);	Igual que `strcat`, sólo que se adjunta un máximo de *Límite* caracteres.	`cstring`
`strncmp(`*Exp_cadena1*, *Exp_cadena2, Límite*`);`	Igual que `strcmp`, sólo que se compara un máximo de *Límite* caracteres.	`cstring`
`strncpy(`*Variable_cadena*, *Expresion_cadena, Límite*`);`	Igual que `strcpy`, sólo que se copia un máximo de *Límite* caracteres.	`cstring`
`strstr(`*Expresion_cadena, Patrón*`)`	Devuelve un apuntador a la primera ocurrencia de la cadena *Patrón* en *Expresion_cadena*. Devuelve el apuntador `NULL` si no se encuentra el *Patrón*.	`cstring`
`strchr(`*Expresion_cadena, Carácter*`)`	Devuelve un apuntador a la primera ocurrencia del *Carácter* en *Expresion_cadena*. Devuelve el apuntador `NULL` si no se encuentra el *Carácter*.	`cstring`
`strrchr(`*Expresión_cadena, Carácter*`)`	Devuelve un apuntador a la última ocurrencia del *Carácter* en *Expresion_cadena*. Devuelve el apuntador `NULL` si no se encuentra el *Carácter*.	`cstring`

†Devuelve un entero que es menor que cero, cero o mayor que cero, dependiendo de si *Exp_cadena1* es menor, igual o mayor que *Exp_cadena2*, respectivamente. El orden es lexicográfico.
*Algunas implementaciones la colocan en `cmath`.

Generador de números aleatorios

Declaración de la función	Descripción	Archivo de encabezado
`int random(int);`	La llamada `random(n)` devuelve un entero pseudo-aleatorio mayor o igual que `0` y menor o igual que `n-1`. (No está disponible en todas las implementaciones. Si no está disponible, debe usar `rand`.)	`cstdlib`
`int rand( );`	La llamada `rand( )` devuelve un entero pseudoaleatorio mayor o igual que `0` y menor o igual que `RAND_MAX`. `RAND_MAX` es una constante entera predefinida que se define en `cstdlib`. El valor de `RAND_MAX` es dependiente de la implementación, pero será por lo menos `32767`.	`cstdlib`
`void srand(unsigned int);` (El tipo *`unsigned int`* es un tipo entero que sólo permite valores no negativos. Puede pensar en el tipo de argumento como *`int`* con la restricción de que no debe ser negativo.)	Reinicializa el generador de números aleatorios. El argumento es la semilla. Si se llama a `srand` varias veces con el mismo argumento, `rand` o `random` (la que usted utilice) producirá la misma secuencia de números pseudoaleatorios. Si se llama a `rand` o a `random` sin llamar antes a `srand`, la secuencia de números producidos será la misma como si se hubiera llamado a `srand` con un argumento de `1`.	`cstdlib`

Funciones trigonométricas

Estas funciones utilizan radianes, no grados.

Declaración de la función	Descripción	Archivo de encabezado
`double acos(double);`	Arco coseno	`cmath`
`double asin(double);`	Arco seno	`cmath`
`double atan(double);`	Arco tangente	`cmath`
`double cos(double);`	Coseno	`cmath`
`double cosh(double);`	Coseno hiperbólico	`cmath`
`double sin(double);`	Seno	`cmath`
`double sinh(double);`	Seno hiperbólico	`cmath`
`double tan(double);`	Tangente	`cmath`
`double tanh(double);`	Tangente hiperbólica	`cmath`

Apéndice 5
La instrucción `assert`

En el capítulo 5 utilizamos lo siguiente para verificar si se abría con éxito un archivo llamado `flujo_in`:

```
if (flujo_in.fail())
{
    cout << "No pudo abrirse el archivo de entrada.\n";
    exit(1);
}
```

La instrucción `assert` puede utilizarse para escribir la misma prueba, de la siguiente manera:

```
assert(!flujo_in.fail());
```

Observe que en este caso tuvimos que insertar un *not* para poder obtener el significado equivalente, ya que estamos afirmando que la apertura del archivo *no falló*.

La instrucción `assert` consta del identificador `assert` seguido por una expresión lógica entre paréntesis y termina con un punto y coma. Puede utilizarse cualquier expresión lógica. Si la expresión lógica es *`false`*, entonces el programa termina y se emite un mensaje de error. Si la expresión lógica es *`true`*, nada ocurre y el programa continúa con la siguiente instrucción después de la instrucción `assert`. Por lo tanto, las instrucciones `assert` son una manera compacta de incluir comprobaciones de error dentro de su programa.

La instrucción `assert` se define en la biblioteca `cassert`, por lo que cualquier programa que utilice una instrucción `assert` deberá contener la siguiente directiva `include`:

```
#include <cassert>
```

`assert` es una macro, la cual es una construcción similar a una función y por lo tanto tiene sentido definirla en una biblioteca.

Una ventaja de usar instrucciones `assert` es que puede desactivarlas. Puede usar las instrucciones `assert` en su programa para depurarlo y después desactivarlas para que los usuarios no reciban mensajes de error que tal vez no entenderían. Al desactivarlas también se reduce el trabajo en exceso realizado por su programa. Para desactivar todas las instrucciones `assert` en su programa, agregue `#define NDEBUG` antes de la directiva `include`, como se muestra a continuación:

```
#define NDEBUG
#include <cassert>
```

Por ende, si inserta `#define NDEBUG` en su programa una vez que está depurado por completo, entonces se desactivan todas las instrucciones `assert` en su programa. Si después modifica su programa y necesita depurarlo de nuevo, puede activar de vuelta las instrucciones `assert` si elimina la instrucción `#define NDEBUG` (o si la convierte en comentario).

Apéndice 6
Funciones en línea

Cuando la definición de una función miembro es corta, puede incluirla dentro de la definición de la clase. Sólo necesita sustituir la declaración de la función miembro con su definición; no obstante y como la definición de la función se encuentra dentro de la definición de la clase, no debe incluir el nombre de la clase ni el operador de resolución de alcance. Por ejemplo, la clase `Par` que se define a continuación tiene definiciones de funciones en línea para sus dos constructores y para la función miembro `obtiene_primero`:

```
class Par
{
public:
    Par( )
    {}
    Par(char primer_valor, char segundo_valor)
    : primero(primer_valor), segundo(segundo_valor) {}
    char obtiene_primero()
    { return primero;}
        ...
private:
    char primero;
    char segundo;
};
```

Observe que en la definición de una función en línea no se requiere un punto y coma después de la llave de cierre, aunque no es incorrecto incluirlo.

El compilador trata las definiciones de funciones en línea de manera distinta, por lo que en general se ejecutan con más eficiencia, aunque consumen más almacenamiento. Con una función en línea, cada llamada a la función en su programa se sustituye por una versión compilada de la definición de la función, por lo que las llamadas a las funciones en línea no producen el exceso de carga de trabajo de la llamada normal a una función.

Apéndice 7
Sobrecarga de los corchetes de índice de arreglo

Puede sobrecargar los corchetes [] para una clase, de manera que puedan usarse con objetos de la clase. Si desea usar [] en una expresión del lado izquierdo de un operador de asignación, entonces el operador debe definirse de manera que devuelva una referencia, lo cual se indica al agregar & al tipo devuelto. (Esto tiene cierta similitud con lo que discutimos acerca de cómo sobrecargar los operadores de E/S << y >>.) Cuando se sobrecarga el operador [], *debe ser* una función miembro; el operador [] sobrecargado *no puede ser* un operador friend. (En esta cuestión, [] se sobrecarga de una forma similar a la manera en que se sobrecarga el operador de asignación =; en la sección del capítulo 12 titulada "Sobrecarga del operador de asignación" vimos cómo sobrecargar el operador =.)

Por ejemplo, el siguiente código define una clase llamada Par, cuyos objetos se comportan como arreglos de caracteres con los dos índices 1 y 2 (*no* 0 y 1):

```
class Par
{
public:
    Par();
    Par(char primer_valor, char segundo_valor);
    char& operator[](int indice);
private:
    char primero;
    char segundo;
};
```

La definición de la función miembro [] puede ser de la siguiente manera:

```
char& Par::operator[](int indice)
{
     if (indice == 1)
         return primero;
     else if (indice == 2)
         return segundo;
     else
     {
         cout << "Valor de indice ilegal.\n";
         exit(1);
     }
}
```

Los objetos se declaran y se utilizan de la siguiente manera:

```
Par a;
a[1] = 'A';
a[2] = 'B';
cout << a[1] << a[2] << endl;
```

Observe que en `a[1]`, `a` es el objeto que hace la llamada y `1` es el argumento para la función miembro `[]`.

Apéndice 8
El apuntador this

Cuando se definen funciones miembro para una clase, algunas veces puede ser conveniente hacer referencia al objeto que hace la llamada. El apuntador *this* es un apuntador predefinido que apunta al objeto que hace la llamada. Por ejemplo, considere una clase como la siguiente:

```
class Ejemplo
{
public:
      ...
     void muestra_cosas( );
      ...
private:
    int cosas;
      ...
};
```

Las siguientes dos formas de definir la función miembro `muestra_cosas` son equivalentes:

```
void Ejemplo::muestra_cosas( )
{
   cout << cosas;
}
//No es un buen estilo, pero se ilustra el uso del apuntador this:
void Ejemplo::muestra_cosas( )
{
     cout << (this->cosas);
}
```

Tenga en cuenta que *this* no es el nombre del objeto que hace la llamada, sino el nombre de un apuntador que apunta al objeto que hace la llamada. El apuntador *this* no puede modificar su valor; siempre apunta al objeto que hace la llamada.

Como indica el comentario antes del ejemplo anterior del uso de *this*, muy pocas veces se requiere usar este apuntador. No obstante, es útil en unas cuantas situaciones.

Uno de los usos comunes del apuntador *this* es para sobrecargar el operador de asignación =. Por ejemplo, considere la siguiente clase:

```
class ClaseCadena
{
public:
    ...
    ClaseCadena& operator=(const ClaseCadena& lado_derecho);
    ...
private:
     char *a;//Arreglo dinámico para un valor de cadena con
             terminación '\0'.
};
```

La siguiente definición del operador de asignación sobrecargado puede usarse en cadenas de asignaciones como

```
s1 = s2 = s3;
```

Esta cadena de asignaciones indica que

```
s1 = (s2 = s3);
```

La definición del operador de asignación sobrecargado utiliza el apuntador *this* para devolver el objeto en el lado izquierdo del signo = (que es el objeto que hace la llamada):

```
//Esta versión no funciona en todos los casos. Vea también la
siguiente versión.
ClaseCadena& ClaseCadena::operator=(const ClaseCadena& lado_derecho)
{
    delete [] a;
    a = new char[strlen(lado_derecho.a) + 1];
    strcpy(a, lado_derecho.a);
    return *this;
}
```

La definición anterior tiene un problema en un caso: si se coloca el mismo objeto en ambos lados del operador de asignación (por ejemplo, `s = s;`), entonces se eliminará el miembro del arreglo. Para evitar este problema puede utilizar el apuntador *this* para probar este caso especial, de la siguiente manera:

```
//Versión final con error corregido:
ClaseCadena& ClaseCadena::operator=(
                                const ClaseCadena& lado_derecho)
{
    if (this == &lado_derecho)
    {
        return *this;
    }
    else
    {
        delete [] a;
        a = new char[strlen(lado_derecho.a) + 1];
        strcpy(a, lado_derecho.a);
        return *this;
    }
}
```

En la sección del capítulo 12 titulada "Sobrecarga del operador de asignación" sobrecargamos el operador de asignación para una clase tipo cadena llamada `VarCadena`. En esa sección no necesitamos el apuntador *this* porque teníamos una variable miembro llamada `longitud_max` que podíamos usar para probar si se utilizaba o no el mismo objeto en ambos lados del operador de asignación =. Con la clase `ClaseCadena` que describimos antes no tenemos alternativa, ya que sólo hay una variable miembro. En este caso no tenemos otra alternativa más que utilizar el apuntador *this*.

Apéndice 9
Sobrecarga de operadores como operadores miembro

Por lo general, en este libro hemos sobrecargado operadores al tratarlos como amigos (friends) de la clase. Por ejemplo, en el cuadro 8.5 del capítulo 8 sobrecargamos el operador + como friend. Para ello etiquetamos el operador como friend dentro de la definición de la clase, como se muestra a continuación:

```
//Clase para montos de dinero en pesos.
class Dinero
{
public:
    friend Dinero operador +(const Dinero& monto1,
                                          const Dinero& monto2);
         . . .
```

Después definimos el operador sobrecargado + fuera de la definición de la clase (como se muestra en el cuadro 8.5).

También es posible sobrecargar el operador + (y otros operadores) como **operadores miembro**. Para sobrecargar el operador + como operador miembro, la definición de la clase debe comenzar así:

```
//Clase para montos de dinero en pesos.
class Dinero
{
public:
    Dinero operator +(const Dinero& monto2);
```

Observe que cuando se sobrecarga un operador binario como operador miembro, sólo hay un parámetro (y no dos). El objeto que hace la llamada actúa como el primer parámetro. Por ejemplo, considere el siguiente código:

```
Dinero costo(1, 50), impuesto(0, 15), total;
total = costo + impuesto;
```

Cuando + se sobrecarga como operador miembro, entonces en la expresión `costo + impuesto` el `costo` variable es el objeto que hace la llamada e `impuesto` es el único argumento para +.

La definición del operador miembro + sería la siguiente:

```
Dinero Dinero::operator +(const Dinero& monto2)
{
     Dinero temp;
     temp.todo_centavos = todo_centavos + monto2.todo_centavos;
     return temp;
}
```

Observe la siguiente línea de la definición de este operador miembro:

```
temp.todo_centavos = todo_centavos + monto2.todo_centavos;
```

El primer argumento para + es `todo_centavos` sin calificativo, por lo que se considera como la variable miembro `todo_centavos` del *objeto que hace la llamada*.

Probablemente le parezca extraño sobrecargar un operador como variable miembro, pero es fácil acostumbrarse a los nuevos detalles. Muchos expertos aconsejan siempre sobrecargar los operadores como operadores miembro, en vez de operadores friend. Esto se relaciona más con la programación orientada a objetos. No obstante, existe una gran desventaja en cuanto a sobrecargar un operador binario como operador miembro. Cuando se sobrecarga un operador binario como operador miembro, los dos argumentos dejan de ser simétricos. Uno es el objeto que hace la llamada y sólo el segundo "argumento" es verdadero. Esto es antiestético, pero también tiene una deficiencia bastante práctica. Cualquier conversión automática de tipos sólo se aplicará al segundo argumento; así, por ejemplo, lo siguiente sería válido:

```
Dinero monto_base(100, 60), monto_completo;
monto_completo = monto_base + 25;
```

Esto se debe a que `Dinero` tiene un constructor con un argumento de tipo *`long`*, y por lo tanto el valor `25` se considerará como un valor *`long`* que se convierte automáticamente en un valor de tipo `Dinero`.

No obstante, si sobrecarga + como operador miembro, entonces no podrá invertir los dos argumentos para +. Lo siguiente es inválido:

```
monto_completo = 25 + monto_base;
```

Esto se debe a que `25` no puede ser un objeto invocador. La conversión de valores *`long`* al tipo `Dinero` funciona para argumentos, pero no para objetos invocadores.

Por otro lado, si sobrecarga + como friend, entonces lo siguiente sería completamente válido:

```
monto_completo = 25 + monto_base;
```

Índice

Símbolos

A

D

E

F

G

H

T

¡Felicidades! Gracias por adquirir el nuevo libro *Resolución de problemas con C++*, quinta edición. Este texto incluye 6 meses de acceso gratuito al sitio Web y apoyos tales como:

- CodeMate de Addison Wesley. Es un código para visualizar, compilar, ejecutar, editar y programar problemas directamente a través de los vínculos que aparecen en su libro de texto, sin tener que instalar algún dispositivo complementario.
- Un libro electrónico, con vínculos a CodeMate de Addison Wesley.
- Autoevaluaciones para reforzar lo aprendido.

Usuarios que utilizan CodeMate por primera vez:
Es necesario registrarse en línea por medio de una computadora con conexión a Internet. El proceso toma sólo unos minutos y el registro será solicitado una sola vez.

1. Vaya a `www.pearsoneducacion.net/savitch`, y complete los pasos solicitados.
2. Siga las instrucciones en pantalla. Si necesita ayuda durante el registro en línea, haga clic en **Cómo Registrarse.**

Usuarios ya registrados:
Solamente necesita registrarse una vez; si ya lo ha hecho, acceda al sitio `www.pearsoneducacion.net/savitch`, seleccione el vínculo **CodeMate** e introduzca su nombre de usuario y su contraseña para comenzar a utilizar el código.

Importante: El código de acceso solamente puede ser utilizado una vez durante seis meses después de la activación. Este código no es transferible. Si desea un acceso adicional, puede adquirirlo en línea a través de la página `www.aw-bc.com/codemate`.