COMO PROGRAMAR EN C/C++

SEGUNDA EDICION

H.M. DEITEL / P.J. DEITEL

PRENTICE HALL

COMO PROGRAMAR EN C/C++

SEGUNDA EDICIÓN

H. M. Deitel
Universidad Nova
Deitel y asociados

P. J. Deitel
Deitel y asociados

UNIVERSIDAD DE LA REPUBLICA
FACULTAD DE INGENIERIA
DPTO. DE DOCUMENTACION Y BIBLIOTECA
BIBLIOTECA CENTRAL
Ing. Edo. Garcia de Zuñiga
MONTEVIDEO - URUGUAY

No. de Entrada 052749

TRADUCCION:
Gabriel Sánchez García
Ingeniero Mecánico Electricista
UNAM

REVISION TECNICA:
Raymundo Hugo Rangel
Prof. Facultad de Ingeniería
UNAM

PRENTICE HALL

MÉXICO • NUEVA YORK • BOGOTÁ • LONDRES • MADRID
MUNICH • NUEVA DELHI • PARÍS • RÍO DE JANEIRO
SINGAPUR • SYDNEY • TOKIO • TORONTO • ZURICH

EDICION ESPAÑOL:

SUPERVISOR DE TRADUCCION: JOAQUIN RAMOS SANTALLA
SUPERVISION PRODUCCION: JULIAN ESCAMILLA LIQUIDANO

EDICION EN INGLES:

Acquisitions Editor: Marcia Horton
Production Editor: Joe Scordato
Copy Editor: Michael Schiaparelli
Chapter Opener and Cover Designer: Jeannette Jacobs
Buyer: Dave Dickey
Supplements Editor: Alice Dworkin
Editorial Assistant: Dolores Mars

DEITEL: COMO PROGRAMAR EN C/C++ 2/ED

Traducido el inglés de la obra: C How to Programm 2/Ed

Prohibida la reproducción total o parcial de esta obra, por cualquier medio o método sin autorización por escrito del editor.

Derechos reservados © 1995 respecto a la primera edición en español publicada por PRENTICE HALL HISPANOAMERICANA, S.A.

Calle 4 Nº 25-2º piso Fracc. Ind. Alce Blanco,
Naucalpan de Juárez, Edo. de México,
C.P. 53370

Miembro de la Cámara Nacional de la Industria Editorial, Reg. Núm. 1524

Original English Language Edition Publisher by Prentice Hall Inc.
Copyright © 1994
All Rights Reserved
ISBN 0-13-226119-7

IMPRESO EN MEXICO/PRINTED IN MEXICO

SEP

PROGRAMAS EDUCATIVOS, S. A. DE C.V.
CALZ. CHABACANO No. 65, LOCAL A
COL. ASTURIAS,DELEG. CUAUHTEMOC,
C.P. 06850, MÉXICO, D.F.

EMPRESA CERTIFICADA POR EL
INSTITUTO MEXICANO DE NORMALIZACIÓN
Y CERTIFICACIÓN A.C., BAJO LA NORMA
ISO-9002: 1994/NMX-CC-004: 1995
CON EL No. DE REGISTRO RSC-048

1996

5000

PARA

El Dr. Stephen Feldman, Presidente de la Universidad Nova y el Dr. Edward Lieblein, Rector del Centro para las Ciencias de la Computación y la Información, de la Universidad Nova.

Por su visión en relación a una institución científica para la educación avanzada, las ciencias de la computación y la investigación en el sur de Florida, y por sus incesantes esfuerzos por cristalizar esa visión en esta joven universidad.

H. M. D.

PARA

Mis profesores en Lawrenceville y el M.I.T.

Por infundir en mí el amor hacia el aprendizaje y por escribir.

P.J.D.

Contenido

05749

UNIVERSIDAD DE LA REPUBLICA
FACULTAD DE INGENIERIA
DEPARTAMENTO DE
DOCUMENTACION Y BIBLIOTECA
MONTEVIDEO - URUGUAY

Ilustraciones

Prefacio

¡Bienvenido a C! Este libro ha sido escrito por un viejo y por un joven. El viejo (HMD; Massachusetts Institute of Technology 1967) ha estado programando y/o enseñando programación por más de 30 años. El joven (PJD; MIT 1991) ha estado programando desde hace una docena de años y se ha infectado del "virus" de la enseñanza y de la escritura. El viejo programa enseña a partir de la experiencia. El joven programa parte de una reserva inagotable de energía. Al viejo le gusta la claridad. Al joven le encanta la espectacularidad. El viejo aprecia la elegancia y la belleza. El joven desea resultados. Nos reunimos los dos para producir un libro que esperamos resulte entre informativo, interesante y entretenido.

En la mayor parte de los entornos educativos, C se enseña a personas que ya saben cómo programar. Muchos educadores piensan que la complejidad de C y cierto número de otras dificultades, incapacitan a C para un primer curso sobre programación —precisamente el curso meta de este libro. por tanto, ¿por qué es que escribimos este texto?

C de hecho se ha convertido en el lenguaje de elección para la implantación de sistemas en la industria, existen buenas razones para creer que su variante orientada a objetos, C++, resultará el lenguaje dominante de los años finales de los 90. Durante trece años Harvey Deitel ha estado enseñando Pascal en entornos universitarios, con énfasis en el desarrollo de programas claramente escritos y bien estructurados. Mucho de lo que se enseña en una secuencia de cursos introductorios de Pascal, forman los principios básicos de la programación estructurada. Hemos presentado este material exactamente de la misma forma en que HMD ha llevado a cabo sus cursos en la universidad. Se presentan algunos escollos, pero cuando éstos ocurren, los hacemos notar y explicamos los procedimientos para manejarlos con eficacia. Nuestra experiencia ha sido en que los alumnos manejan el curso de forma aproximada igual a como manejan Pascal. Existe una diferencia notable: sin embargo, los alumnos resultan muy motivados por el hecho de que están aprendiendo un lenguaje que les será de inmediata utilidad en cuanto dejen el entorno universitario. Esto incrementa su entusiasmo para este material —una gran ayuda cuando usted considera que C es en realidad más difícil de aprender.

Nuestra meta estaba clara: producir un libro de texto de programación en C para cursos a nivel universitario de introducción a la programación de computadoras, para estudiantes con poca o ninguna experiencia en programación,

pero también producir un libro que ofreciera el tratamiento riguroso de la teoría y de la práctica que se requiere en los cursos tradicionales de C. Para alcanzar estas metas, tuvimos que producir un libro más extenso que otros textos de C esto se debe al hecho de que nuestro texto enseña también de forma paciente los principios de programación estructurada. Aproximadamente mil alumnos han estudiado este material en nuestros cursos. Y en todo el mundo, decenas de millares de estudiantes han aprendido a programar en C, partiendo de la primera edición de este libro.

El libro contiene una gran recopilación de ejemplos, ejercicios y proyectos que se tomaron de muchos campos, a fin de proporcionar al estudiante la oportunidad de resolver problemas interesantes del mundo real.

El libro se concentra en principios de buena ingeniería de software y hace hincapié en la claridad de la programación, mediante el uso de la metodología de la programación estructurada. Evitamos el uso de terminología y de especificaciones de sintaxis antiguas, favoreciendo el enseñar mediante el ejemplo.

Entre los dispositivos pedagógicos de este texto, se encuentran programas y resultados ilustrativos completos, para demostrar los conceptos; un conjunto de objetivos y una sinopsis al principio de cada uno de los capítulos; los errores comunes de programación y las prácticas sanas de programación que se enumeran a todo lo largo de cada uno de los capítulos y se resumen al final de los mismos; secciones de resumen y de terminología en cada capítulo; ejercicios de autoevaluación y sus respuestas en cada capítulo; y la recopilación más completa de ejercicios existente en ningún libro de C. Los ejercicios van desde simples preguntas recordatorio a problemas extensos de programación y a proyectos de importancia. Aquellos instructores que requieran sustanciales proyectos de fin de cursos para sus alumnos, encontrarán muchos problemas apropiados enlistados en los ejercicios correspondientes a los capítulos 5 hasta el 21. Hemos puesto un gran esfuerzo en la recopilación de los ejercicios para aumentar el valor de este curso para el alumno. Los programas en el texto se probaron en compiladores que cumplen con normas ANSI C en SPARCstations de Sun, en Macintosh (Think C) de Apple, y en la PC de IBM (Turbo C, Turbo C++, y Borland C++), y en el VAX/VMS de DEC (VAX C).

Este texto se apega al estándar ANSI C. Muchas características de ANSI C no funcionarán con versiones anteriores a ANSI C. Vea los manuales de consulta correspondientes a su sistema particular en relación con mayores detalles con respecto al lenguaje, u obtenga una copia de ANSI/ISO 9899: 1990, "American National Standard for Information Systems Programming Language C", que puede obtener del American National Standards Institute, 11 West 42nd Street, New York, New York 10036.

Acerca del libro

En este libro abundan características que contribuirán al aprendizaje de los alumnos.

Objetivos

Cada capítulo se inicia con un enunciado de objetivos. Esto le indica al alumno qué es lo que puede esperar, y le da una oportunidad, una vez que haya leído el capítulo, de determinar si él o ella han cumplido con estos objetivos. Ello genera confianza y es fuente de refuerzo positivo.

Citas

A los objetivos de enseñanza les siguen una serie de citas. Algunas son humorísticas, otras filosóficas y varias ofrecen pensamientos interesantes. Nuestros alumnos han expresado que disfrutaron relacionando tales citas con el material de cada capítulo.

Sinopsis

La sinopsis del capítulo ayuda al alumno a enfrentarse al material en un orden de descendente. Esto, también ayuda a los estudiantes a anticipar qué es lo que va a seguir y a establecer un ritmo coherente.

Secciones

Cada capítulo está organizado en pequeñas secciones que se ocupan de áreas clave. Las características de C se presentan en el contexto de programas completos que funcionan. Cada programa esta acompañado por una ventana que contiene el resultado que se obtiene al ejecutar el programa. Esto permite al estudiante confirmar que los programas se ejecutan como se espera. La relación de los resultados con los enunciados de los programas que los generan, es una forma excelente de aprender y reforzar los conceptos. Nuestros programas están diseñados para ejercitar las diversas características de C. La lectura cuidadosa del texto es muy similar a la introducción y ejecución de estos programas en computadora.

Ilustraciones

Se incluyen numerosos dibujos y tablas. El análisis de la diagramación estructurada de flujo, ayuda a los estudiantes a apreciar el uso de estructuras de control y de programación estructurada, incluye diagramas de flujo cuidadosamente dibujados. El capítulo sobre estructuras de datos utiliza muchos dibujos de línea para ilustrar la creación y el mantenimiento de estructuras importantes de datos, como son listas enlazadas, colas de espera, pilas y árboles binarios.

Elementos de diseño útiles

Hemos incluido cuatro elementos de diseño para ayudar a los estudiantes a concentrarse en aspectos importantes del desarrollo de programas, su prueba y depuración, su rendimiento y portabilidad. Resaltamos varios de éstos en la forma de sanas prácticas de programación, errores comunes de programación, sugerencias de rendimiento, sugerencias de portabilidad, y observaciones de ingeniería de software.

Prácticas sanas de programación

Las prácticas sanas de programación se resaltan en el texto. Al estudiante le llama la atención las técnicas que ayudan a producir mejores programas. Estas prácticas representan lo mejor que hemos podido recabar de cuatro décadas combinadas de experiencia en programación.

Errores comunes de programación

Los estudiantes que están aprendiendo un lenguaje en especial en su primer curso de programación tienden a cometer ciertos errores que son comunes. Enfocar la atención de los estudiantes a estos errores comunes de programación resulta un gran auxilio. ¡También ayuda a reducir largas filas en la parte exterior de las oficinas de los instructores durante horas hábiles!

Sugerencias de rendimiento

Encontramos que, por mucho, la meta más importante de un primer curso de programación, es escribir programas claros y comprensibles. Pero los estudiantes desean escribir programas que se ejecuten de forma más rápida, utilicen el mínimo de memoria, requieran un mínimo de tecleo, o que brillen en alguna otra forma elegante. A los estudiantes les interesa en verdad el rendimiento.

Quieren saber qué es lo que pueden hacer para "turbocargar" su programas; por tanto, hemos incluido sugerencias de rendimiento para resaltar las oportunidades que mejoran el rendimiento de los programas.

Sugerencias de portabilidad

El desarrollo de software es una actividad compleja y muy costosa. Las organizaciones que desarrollan software a menudo deben producir versiones personalizadas para una variedad de computadoras y de sistemas operativos. Por tanto, hoy día se hace gran énfasis en la portabilidad, es decir, en la producción de software que podrá ser ejecutado sin cambio en muchos sistemas de computación diferentes. Muchas personas presumen que C es el mejor lenguaje para el desarrollo de software portable. Algunas personas suponen que si implantan una aplicación en C, esta última, de forma automática, resultará portátil. Esto no es cierto. Conseguir portabilidad, requiere de un diseño cuidadoso y cauteloso. Existen muchos escollos. En sí mismo, el documento de ANSI Standard C enlista 11 páginas de dificultades potenciales. Nosotros incluimos numerosas sugerencias de portabilidad. Hemos combinado nuestra propia experiencia en la elaboración de software portable con un estudio cuidadoso de la sección estándar ANSI relativa a portabilidad, así como de dos excelentes libros que tratan sobre la portabilidad (véase la referencia Ja89 y Ra90 al final del capítulo 1).

Observaciones de ingeniería de software

Este elemento de diseño es nuevo en esta segunda edición. Hemos resumido muchas observaciones que afectan la arquitectura y la construcción de los sistemas de software, en especial en sistemas a gran escala.

Resumen

Cada uno de nuestros capítulos termina con una cantidad de dispositivos pedagógicos adicionales. Presentamos un resumen detallado del capítulo en forma de lista con viñetas. Esto auxilia a los estudiantes a revisar y reforzar conceptos clave. Entonces recopilamos y enlistamos, en orden, todas las prácticas sanas de programación, los errores comunes de programación, las sugerencias de rendimientos, las sugerencias de portabilidad y las observaciones de ingeniería de software que aparecen en el capítulo.

Terminología

Incluimos una sección de terminología con una lista alfabética de términos importantes que se definen a lo largo del capítulo. Otra vez, se trata de una confirmación redundante. A continuación resumimos las prácticas sanas de programación, los errores comunes de programación, las sugerencias de rendimiento, las sugerencias de portabilidad y las observaciones de ingeniería de software.

Ejercicios de autoevaluación

Se incluyen, para autoestudio, gran cantidad de ejercicios de autoevaluación con sus respuestas completas. Esto le da la oportunidad al estudiante de obtener confianza con el material y prepararse para intentar resolver los ejercicios regulares.

Ejercicios

Cada capítulo concluye con un conjunto sustancial de ejercicios que van desde el simple recordatorio de terminología y conceptos importantes, hasta escribir enunciados individuales en

C, a escribir pequeñas porciones de funciones en C, a escribir funciones y programas completos en C, e inclusive proyectos importantes de fin de curso. El gran número de ejercicios le permite a los instructores ajustar sus cursos a las necesidades únicas de su auditorio y variar las asignaciones del curso cada semestre. Los instructores pueden utilizar estos ejercicios para formar asignaciones para trabajo en casa, exámenes cortos y de importancia.

Un recorrido por el libro

El libro está dividido en tres partes principales. La primera parte, los capítulos 1 hasta el 14, presenta un tratamiento completo del lenguaje de programación C, incluye una introducción formal a la programación estructurada. La segunda parte única en libros de texto de C, los capítulos 15 hasta el 21, presentan un tratamiento sustancial de la programación de C++ orientada a objetos, suficiente para un curso universitario de pregraduación de alto nivel. La tercera parte, los apéndices A hasta E, presentan una variedad de materiales de consulta y referencia en apoyo al texto principal.

El capítulo 1, "Introducción", analiza qué son las computadoras, cómo funcionan, y cómo se programan. Introduce la idea de la programación estructurada y explica por qué ese conjunto de técnicas ha acelerado una revolución en la forma de escribir programas. El capítulo presenta una historia breve del desarrollo de los lenguajes de programación a partir de los de máquina, pasando por los lenguajes ensambladores hasta los de alto nivel. Se presenta el origen del lenguaje de programación C. El capítulo incluye una introducción al entorno de programación de C.

El capítulo 2, "Introducción a la programación de C", da una introducción concisa a la escritura de programas C. Se presenta un tratamiento detallado de la toma de decisiones y de operaciones aritméticas en C. Después de estudiar este capítulo, el estudiante sabrá cómo escribir programas simples, pero completos, de C.

En el capítulo 3, "Programación estructurada", es probable que sea el capítulo más importante del texto, en especial para el estudiante serio de la ciencia de la computación. Introduce la noción de los algoritmos (procedimientos) para la resolución de problemas. Explica la importancia que tiene la programación estructurada para la producción de programas que sean comprensibles, depurables, mantenibles y que es probable funcionen de forma correcta a partir del primer intento. Introduce las estructuras de control fundamental de la programación estructurada, es decir, la secuencia, la selección (if y if/else), y las estructuras de repetición (while). Explica la técnica de refinamiento descendente paso a paso, que es crítica a la producción de programas estructurados en forma correcta. Presenta la ayuda popular del diseño de programas, el seudocódigo estructurado. Los métodos y planteamientos que se utilizan en el capítulo 3 son aplicables a la programación estructurada de cualquier lenguaje de programación, y no solo de C. Este capítulo ayuda a desarrollar en el estudiante buenos hábitos de programación, preparándolo para el manejo de tareas más sustanciales de programación en el resto del texto.

El capítulo 4, "Control de programa", refina las nociones de la programación estructurada y nos presenta estructuras adicionales de control. Examina en detalle la repetición, y compara las alternativas de ciclos controlados por contador con ciclos controlados por centinela. Se presenta la estructura `por` como un medio conveniente para implantar ciclos controlados por contador. La estructura de selección **`switch`** y la estructura de repetición **`do/while`** también se presentan. El capítulo concluye con un análisis de operadores lógicos.

En el capítulo 5, "Funciones", se analiza el diseño y la construcción de módulos de programa. C incluye funciones de biblioteca estándar, funciones definidas por el programador, recursión y capacidades de llamadas por valor. Las técnicas que se presentan en el capítulo 5 son esenciales

a la producción y apreciación de programas correctamente estructurados, en especial aquellos grandes programas de software que los programadores de sistemas y los programadores de aplicaciones quizá tendrán que desarrollar en aplicaciones de la vida real. La estrategia de "divide y vencerás" se introduce como un medio eficaz para la resolución de problemas complejos; las funciones permiten al programador la división de programas complejos en componentes más sencillos interactuantes. Los estudiantes que disfrutan del tratamiento de los números y la simulación aleatoria, apreciarán el análisis del juego de dados, que hace un uso elegante de las estructuras de control. El capítulo ofrece una introducción sólida a la recursión e incluye una tabla resumiendo los 31 ejemplos y ejercicios de recursión que aparecen distribuidos a todo lo largo del libro. Algunos libros dejan la recursión para un postrer capítulo; pero sentimos que este tema se cubre mejor en forma gradual a todo lo largo del texto. La recopilación extensiva de 39 ejercicios al final del capítulo 5, incluye varios problemas clásicos de recursión, como las torres de Hanoi.

El capítulo 6, "Arreglos", analiza la estructuración de datos en arreglos, o grupos, de elementos de datos relacionados del mismo tipo. El capítulo presenta numerosos ejemplos tanto de arreglos de un solo subíndice como de arreglos con doble subíndice. Es muy importante estructurar los datos así como usar estructuras de control en el desarrollo de programas correctamente estructurados. Ejemplos en el capítulo investigan varias manipulaciones comunes de arreglos, la impresión de histogramas, la clasificación y ordenamiento de datos, el pasar arreglos a funciones, y una introducción al campo del análisis de datos de encuestas. Una característica de este capítulo es la presentación cuidadosa de la búsqueda binaria como una mejora dramática en comparación con la búsqueda lineal. Los ejercicios al fin del capítulo, incluyen una selección en especial grande de problemas interesantes y retadores. Estos incluyen técnicas de clasificación mejoradas, el diseño de un sistema de reservaciones de aerolínea, una introducción al concepto de los gráficos tipo tortuga (mismo que se hizo famoso en el lenguaje LOGO), y los problemas de la torre del caballero y de las ocho reinas, que introducen las ideas de la programación heurística, tan ampliamente empleada en el campo de la inteligencia artificial.

En el capítulo 7, "Apuntadores", se presenta una de las características más poderosas del lenguaje C. El capítulo proporciona explicaciones detalladas de los operadores de apuntador, a los que, por referencia, se les da el nombre de expresiones de apuntador, aritméticas de apuntador, de la relación entre apuntadores y arreglos, arreglos de apuntadores y apuntadores a funciones. Los ejercicios del capítulo incluyen una simulación de la carrera clásica entre la tortuga y la liebre, y los algoritmos de barajar y repartir naipes. Una sección especial titulada "Cómo construir su propia computadora" también está incluida. Esta sección explica la noción de la programación del lenguaje máquina y sigue con un proyecto que incluye el diseño y la implantación de un simulador de computadora, que permite al lector escribir y ejecutar programas en lenguaje máquina. Esta característica única del texto será en especial útil al lector que desee comprender cómo en realidad funcionan las computadoras. Nuestros alumnos disfrutan de este proyecto y a menudo implantan mejorías sustanciales; muchas de ellas están sugeridas dentro de los ejercicios. En el capítulo 12, otra sección especial guía al lector a lo largo de la elaboración o construcción de un compilador. El lenguaje máquina producido por el compilador después se ejecuta en el simulador del lenguaje máquina que se produce en el capítulo 7.

El capítulo 8, "Caracteres y cadenas", se ocupa de los fundamentos del procesamiento de datos no numéricos. El capítulo incluye un recorrido completo de las funciones de procesamiento de caracteres y de cadenas, disponibles en las bibliotecas de C. Las técnicas que aquí se analizan son de amplia utilidad en la construcción de procesadores de texto, en software de disposiciones de páginas y tipografía, y en aplicaciones de procesamiento de palabras. El capítulo incluye una recopilación interesante de 33 ejercicios que exploran aplicaciones de procesamiento de texto. El

estudiante disfrutará los ejercicios en la escritura de quintillas humorísticas, poesía aleatoria, conversión del inglés al latín vulgar, generación de palabras de siete letras que sean equivalentes a un número telefónico dado, la justificación texto, la protección de cheques, escribir una cantidad de un cheque en palabras, generación de clave Morse, conversiones métricas y de cartas reclamando deudas. ¡El último ejercicio reta al estudiante a utilizar un diccionario computarizado para crear un generador de crucigramas!

El capítulo 9, "Entrada/salida con formato", presenta todas las capacidades poderosas de formato de printf y scanf. Analizamos las capacidades de formato de salida de **`printf`** como son el redondeo de los valores de punto flotante a un número dado de decimales, la alineación de columnas de números, justificación a la derecha y a la izquierda, inserción de información literal, el forzado del signo más, impresión de ceros a la izquierda, el uso de notación exponencial, el uso de números octales y hexadecimales, y el control de los anchos y precisiones de campo. Se analizan todas las secuencias de escape de **`printf`** en relación con el movimiento del cursor, al imprimir caracteres especiales y para generar una alarma audible. Examinamos todas las capacidades de formato de entrada de **`scanf`**, incluyendo la entrada de tipos específicos de datos y el pasar por alto caracteres específicos del flujo de entrada. Se analizan todos los especificadores de conversión de **`scanf`** para la lectura de valores decimales, octales, hexadecimales, de punto flotante, de carácter y de cadenas. Analizamos las entradas, para que coincidan (o no coincidan) con los caracteres de un conjunto. Los ejercicios del capítulo prueban de forma virtual todas las capacidades de formato de entrada/salida de C.

El capítulo 10, "Estructuras, uniones, manipulaciones de bits y enumeraciones", presenta una variedad de características de importancia. Las estructuras son como los registros de Pascal y de otros lenguajes —agrupan elementos de datos de varios tipos. Las estructuras se utilizan en el capítulo 11 para formar archivos compuestos de registros de información. Las estructuras se utilizan en conjunción con los apuntadores y la asignación dinámica de memoria del capítulo 12, para formar estructuras dinámicas de datos como son listas enlazadas, colas de espera, pilas y árboles. Las uniones permiten que se utilice un área en memoria para diferentes tipos de datos en diferentes momentos; esta capacidad de compartir puede reducir los requerimientos de memoria de un programa o los requerimientos de almacenamiento secundario. Las enumeraciones proporcionan una forma conveniente de definición de constantes simbólicas útiles; esto ayuda a que los programas sean más autodocumentales. La poderosa capacidad de manipulación de bits posibilita a los programadores que escriban propagandas que hagan uso del hardware a bajo del nivel. Esto ayuda a que los programas procesen cadenas de bits, ajusten bits individuales en **`on`** o en **`off`**, y almacenen información de una forma más compacta. Estas que a menudo sólo aparecen en lenguajes ensambladores de bajo nivel, son valiosas para los programadores que escriben software de sistema, como son sistemas operativos y software de redes. Una característica del capítulo es su simulación revisada de alto rendimiento de barajar y repartir naipes. Esta es una excelente oportunidad para que el instructor haga énfasis en la calidad de los algoritmos.

El capítulo 11, "Procesamiento de archivos", analiza las técnicas que se utilizan para procesar archivos de texto con acceso secuencial y aleatorio. El capítulo se inicia con una introducción a la jerarquía de datos desde bits, pasando por bytes, campos, registros y hasta archivos. A continuación, se presenta una vista simple de archivos y flujos. Se analizan los archivos de acceso secuencial utilizando una serie de tres programas que muestran cómo abrir y cerrar archivos, cómo almacenar datos en un archivo en forma secuencial, y cómo leer datos en forma secuencial de un archivo. Los archivos de acceso aleatorio se tratan utilizando una serie de cuatro programas que muestran cómo crear de forma secuencial un archivo para acceso aleatorio, cómo leer y escribir datos a un archivo con acceso aleatorio, y cómo leer datos en forma secuencial de un archivo con

acceso aleatorio. El cuarto programa de acceso aleatorio combina muchas de las técnicas de acceso a archivos tanto secuencial como aleatorio formando un programa completo de proceso de transacciones. Los estudiantes en nuestros seminarios de industria nos han indicado después de estudiar el material relativo a procesamiento de archivos, que fueron capaces de producir programas sustanciales de procesamiento de archivo que resultaron de utilidad inmediata para sus organizaciones.

El capítulo 12, "Estructuras de datos", analiza las técnicas que se utilizan para crear estructuras dinámicas de datos. El capítulo se inicia con un análisis de estructuras autorreferenciadas y asignación dinámica de memoria. El capítulo continúa con un análisis de cómo crear y mantener varias estructuras dinámicas de datos incluyendo listas enlazadas, colas de espera (o líneas de espera), pilas y árboles. Para cada uno de los tipos de estructuras de datos, presentamos programas completos y funcionales mostrando salidas de muestra. El capítulo 12 ayuda al estudiante a dominar de verdad los apuntadores. El capítulo incluye ejemplos abundantes, utilizando indirección y doble indirección un concepto en particular difícil. Un problema al trabajar con apuntadores, es que los estudiantes tienen dificultad para visualizar las estructuras de datos y cómo se enlazan juntos sus nodos. Por tanto, hemos incluido ilustraciones para mostrar los enlaces, y la secuencia en la cual se crean. El ejemplo del árbol binario es una piedra angular soberbia para el estudio de apuntadores y de estructura dinámica de datos. Este ejemplo crea un árbol binario; obliga a eliminación duplicada; y nos inicia en recorridos recursivos de árboles en preorden, en orden y en posorden. Los estudiantes tienen un *verdadero sentido de realización* cuando estudian e implantan este ejemplo. Estiman en forma muy particular el ver que el recorrido en orden imprime los valores de los nodos en orden clasificado. El capítulo incluye una recopilación importante de ejercicios. Una parte a destacar del capítulo es la sección especial "Cómo construir su propio compilador". Los ejercicios guían al estudiante a través del desarrollo de un programa de conversión de infijos a posfijos y un programa de evaluación de expresión de posfijos. Después modificamos el algoritmo de evaluación de posfijos para generar código en lenguaje de máquina. El compilador coloca este código en un archivo (utilizando las técnicas del capítulo 11). ¡Después los estudiantes ejecutan el lenguaje de máquina producido por sus compiladores en los simuladores de software que construyeron en los ejercicios del capítulo 7!

El capítulo 13, "El preprocesador", proporciona análisis detallados de las directrices del preprocesador. El capítulo proporciona información más completa en la directriz **#incluye** que hace que se incluya una copia del archivo especificado en lugar de la directriz antes de la compilación del archivo, y de la directriz **#define** que crea constantes simbólicas y macros. El capítulo explica la compilación condicional, para permitirle al programador el control de la ejecución de las directrices del preprocesador, y la compilación del código del programa. El operador # que convierte su operando en una cadena y el operador ## que concatena dos señales también son analizados. Las cinco constantes simbólicas predefinidas (**__LINE__**, **__FILE__**, **__DATE__**, **__TIME__**, y **__STDC__**) son presentadas. Por último, se estudia el macro **assert** de la cabecera **assert.h** es valioso en la prueba, depuración, verificación y validación de programas.

El capítulo 14, "Temas avanzados", presenta varios temas avanzados que de forma ordinaria no son cubiertos en cursos de introducción. La sección 14.2 muestra cómo redirigir salida a un programa proveniente de un archivo, cómo redirigir salida de un programa para colocarse en un archivo, cómo redirigir la salida de un programa para resultar la entrada de otro programa (entubado) y agregar la salida de un programa a un archivo existente. La sección 14.3 analiza como desarrollar funciones que utilizan listas de argumentos de longitud variable. La sección 14.4 muestra como pueden ser pasados argumentos de la línea de comandos para la función **main** y

utilizados en un programa. La sección 14.5 analiza la compilación de programas cuyos componentes están dispersos en varios archivos. La sección 14.6 estudia el registro de funciones utilizando **atexit** para ser ejecutados en la terminación de un programa, terminar la ejecución de un programa con la función **exit**. La sección 14.7 estudia los calificadores de tipo **const** y **volátiles**. La sección 14.8 muestra cómo especificar el tipo de una constante numérica utilizando los sufijos de entero y de punto flotante. La sección 14.9 explica archivos binarios y el uso de archivos temporales. La sección 14.10 muestra cómo utilizar la biblioteca de manejo de señales para atrapar eventos no esperados. La sección 14.11 analiza la creación y utilización de arreglos dinámicos utilizando **calloc** y **realloc**.

En la primera edición de este texto, se incluyó una introducción de un capítulo a C++ y a la programación orientada a objetos. Durante este tiempo, muchas universidades han decidido incorporar una introducción a C++ y la programación orientada a objetos en sus cursos C. Por lo cual en esta edición, hemos expandido este tratamiento a siete capítulos con suficiente texto, ejercicios y laboratorios para un curso de un semestre.

El capítulo 15, "C++ como un C mejor", introduce las características no orientadas a objetos de C++. Estas características mejoran el proceso de escritura de programas convencionales orientados a procedimientos. El capítulo discute comentarios de una sola línea, flujo de entrada/salida, declaraciones, cómo crear nuevos tipos de datos, prototipos de función y verificación de tipos, funciones en línea (como una sustitución de los macros), parámetros de referencia, el calificador const, asignación dinámica de memoria, argumentos por omisión, el operador de resolución de ámbito unario, la homonimia de funciones, las especificaciones de vinculación y las plantillas de funciones.

El capítulo 16, "Clases y abstracción de datos", representa una maravillosa oportunidad para la enseñanza de la abstracción de datos de la "manera correcta" mediante un lenguaje (C++) dedicado de forma expresa a la implantación de tipos de datos abstractos (ADT). En años recientes, la abstracción de datos se ha convertido en un tema de importancia en los cursos de computación introductorios que se enseñan en Pascal. Conforme estamos escribiendo este libro, tomamos en consideración la presentación de la abstracción de datos en C, pero decidimos que en vez de ello se incluiría esta introducción detallada a C++. Los capítulos 16, 17 y 18, incluyen una presentación sólida de la abstracción de datos. El capítulo 16 analiza la implantación de ADT como *structs*, implantando ADT como clases de estilo C++, el acceso a miembros de clase, la separación de la interfaz de la implantación, el uso de funciones de acceso y de funciones de utilería, la inicialización de objeto mediante constructores, la destrucción de objetos mediante destructores, la asignación por omisión de copia a nivel de miembro, y la reutilización del software.

El capítulo 17, "Clases Parte II", continúa con el estudio de las clases y la abstracción de datos. El capítulo analiza la declaración y el uso de los objetos constantes, de las funciones miembro constantes, de la composición —el proceso de elaboración de clases que tienen otras clases como miembros, funciones amigos y clases amigos que tienen derechos de acceso especiales a los miembros privados de clases, el apuntador **this** permite que un objeto conozca su propia dirección, la asignación dinámica de memoria, los miembros de clase estáticos para contener y manipular datos de todo el ámbito de la clase, ejemplos de tipos de datos abstractos populares (arreglos, cadenas y colas de espera), clases contenedoras, iteradores, y clases plantilla. Las clases plantilla están entre las adiciones más recientes al lenguaje en evolución de C++. Las clases plantilla permiten al programador capturar la esencia de un tipo de datos abstracto (como una pila, un arreglo o una fija de espera) y a continuación crear, con la inclusión mínima de código adicional, versiones de esta ADP para tipos particulares (como una pila de int, una pila de float,

una fija de espera de int, etcétera). Por esta razón, las clases plantilla a menudo se conocen como tipos parametrizados.

El capítulo 18, "Homonimia de operadores", es uno de los temas más populares en los cursos de C++. Los alumnos en realidad disfrutan de este material. Encuentran que encajan de forma perfecta con el análisis de tipos de dato abstractos de los capítulos 16 y 17. La homonimia de operadores permite al programador indicarle al compilador cómo utilizar operadores existentes con objetos de nuevos tipos. C++ ya sabe cómo utilizar estos operadores con objetos de tipos incorporados, como son los enteros, puntos flotantes y caracteres. Pero suponga que creamos una nueva clase de cadena. ¿Qué es lo que significa el signo más? Muchos programadores utilizan el signo más con cadenas para significar concatenación. En este capítulo, el programador aprenderá cómo demostrar el "homónimo" del signo más de tal forma, que cuando este escrito entre dos objetos cadena en una expresión, el compilador generará una llamada de función a una "función operador" que concatenará las dos cadenas. El capítulo estudia los fundamentos de la homonimia de operadores, las restricciones en la homonimia de operadores, la homonimia con funciones miembros de clase en comparación con funciones no miembros, la homonimia de operadores unarios y binarios, y la conversión entre tipos. Una característica del capítulo es la gran cantidad de casos de estudio de importancia, es decir una clase de arreglo, una clase de cadena, una clase de fecha, una clase grande de entero, y una clase de números complejos (los dos últimos aparecen con todo el código fuente en los ejercicios).

El capítulo 19, "Herencia", trata con una de las capacidades fundamentales de los lenguajes de programación orientadas a objetos. La herencia es una forma de reutilización del software en la cual se pueden desarrollar de forma rápida clases nuevas al absorber las capacidades de clases existentes y a continuación añadiendo capacidades nuevas apropiadas. El capítulo discute las nociones de clases base y clases derivadas, miembros protegidos, herencia pública, herencia protegida, herencia privada, clases base directas, clases base indirectas, utilización de constructores y destructores en clases base y derivadas, ingeniería de software con herencia. El capítulo compara la herencia (relaciones "es una") con composición (relaciones "tiene una") e introduce relaciones "utiliza una" y "conoce una". Una característica del capítulo es la inclusión de varios estudios de caso de importancia. En particular, un estudio de caso extenso implanta una jerarquía de clase punto, círculo y cilindro. El capítulo concluye con un estudio de caso de herencia múltiple una característica avanzada de C++ en el cual la clase derivada puede ser formada al heredar atributos y comportamientos de varias clases base.

El capítulo 20, "Funciones virtuales y polimorfismo", trata con otra de las capacidades fundamentales de la programación orientada a objetos, es decir comportamiento polimorfo. Muchas clases están relacionadas mediante la herencia a una clase base común, cada objeto de clase derivada, puede ser tratado como un objeto de clase base. Esto permite a los programas que sean escritos de una forma bastante general independiente de los tipos específicos de objetos de clase derivada. Se pueden manejar nuevos tipos de objetos mediante el mismo programa, haciendo por tanto los sistemas más extensibles. El polimorfismo permite a los programas eliminar lógica compleja de intercambio en favor de una lógica más sencilla "de línea recta". Un administrador de video para un juego de video, por ejemplo, puede simplemente enviar un mensaje de dibujar a todos los objetos de una lista enlazada de objetos a ser dibujados. Cada objeto sabe como dibujarse a sí mismo. Se puede añadir al programa un nuevo objeto sin modificar el programa siempre y cuando dicho objeto también sepa como dibujarse a sí mismo. Este estilo de programación se utiliza de forma típicamente para implantar interfaces gráficas de usuario tan populares hoy en día. El capítulo discute la mecánica de la obtención del comportamiento polimórfico, mediante el uso de funciones virtuales. El capítulo hace la distinción entre clases abstractas (a

partir de las cuales no se puede producir ningún objeto) y clases concretas (a partir de las cuales se pueden producir objetos). Las clases abstractas son útiles para proporcionar una interfaz capaz de heredarse a las clases a todo lo largo de la jerarquía. Una característica del capítulo son sus dos estudios de casos polimórficos de importancia un sistema de nóminas y otra versión de la jerarquía de forma de puntos, círculo y cilindro que fue estudiada en el capítulo 19.

El capítulo 21, "C++ flujo de entrada/salida", contiene un tratamiento en extremo detallado del nuevo estilo orientado a objetos de entrada/salida introducido en C++. Se enseñan o se dan muchos cursos de C utilizando compiladores C++, y los instructores a menudo prefieren enseñar el estilo nuevo de C++ correspondiente a E/S antes de continuar utilizando el estilo anterior o más viejo de **`printf/scanf`**. El capítulo analiza las varias capacidades de E/S de C++ que incluye salida utilizando el operador de inserción de flujo, entrada con el operador de extracción de flujo, E/S de tipo seguro (una agradable mejoría sobre C), E/S con formato, E/S sin formato (para un mayor rendimiento), manipuladores de flujo para controlar la base del flujo (decimal, octal o hexadecimal), números de punto flotante, cómo controlar los anchos de campo, manipuladores definidos por usuario, estados de formato de flujo, estados de error de flujo, E/S de objetos de tipos definido por usuario, y cómo ligar flujos de salida con flujos de entrada (para asegurar que en realidad aparecen indicadores antes que el usuario introduzca respuestas).

Varios apéndices proporcionan material valioso de consulta y referencia. En particular, presentamos en el apéndice A un resumen sintáctico de C; en el apéndice B aparece un resumen de todas las funciones de biblioteca estándares de C, con explicaciones; en el apéndice C una gráfica completa de precedencia y asociatividad de operadores; en el apéndice D el conjunto de códigos de caracteres ASCII; y en el apéndice E un análisis de los sistemas numéricos binarios, octal, decimal y hexadecimal. El apéndice B fue condensado del documento estándar ANSI, con el permiso expreso por escrito del American National Standards Institute; este apéndice resulta una referencia detallada y valiosa para el programador practicante de C. El apéndice E es una guía didáctica completa sobre sistemas numéricos incluyendo muchos ejercicios de autoevaluación y sus respuestas.

Reconocimientos

Uno de los grandes placeres de escribir un libro de texto consiste en reconocer los esfuerzos de muchas personas cuyos nombres pudieran no aparecer en las portadas, pero sin cuyo trabajo, cooperación, amistad y comprensión, la elaboración de este texto hubiera resultado imposible.

HMD desea agradecer a sus colegas de la Universidad Nova Ed Simco, Clovis Tondo, Ed Lieblein, Phil Adams, Raisa Szabo, Raúl Salazar y Bárbara Edge.

Nos gustaría agradecer a nuestros amigos de Digital Equipment Corporation (Stephanie Stosur Schwartz, Sue-Lane Garrett, Janet Hebert, Faye Napert, Betsy Mills, Jennie Connolly, Bárbara Couturier y Paul Sandore), de la Sun Microsystems (Gary Morin), de la Corporation for Open Systems International (Bill Horst, David Litwack, Steve Hudson, y Linc Faurer), Informative Stages (Don Hall), Semaphore Training (Clieve Lee), y Cambridge Technology Partners (Gart Davis, Paul Sherman, y Wilberto Martínez), así como a numerosos clientes corporativos que han hecho de la enseñanza de este material en un entorno industrial una experiencia placentera.

Hemos sido afortunados de haber tenido la posibilidad de trabajar en este proyecto con un equipo talentoso y dedicado de profesionales de la publicación en Prentice Hall. Joe Scordato hizo un magnífico trabajo como editor de producción. Dolores Mars coordinó el esfuerzo complejo de revisión del manuscrito y siempre resultó de increíble ayuda cuando necesitamos asistencia. Su entusiasmo y buena disposición *son* sinceramente apreciados.

Este libro se concibió debido al estímulo, entusiasmo y persistencia de Marcia Horton, Editor en jefe. Resulta un gran crédito para Prentice Hall que sus funcionarios más importantes continúen en sus responsabilidades editoriales. Siempre nos ha impactado con ello y estamos agradecidos de ser capaces de continuar trabajando de cerca con Marcia, inclusive ante su aumento de responsabilidades administrativas.

Apreciamos los esfuerzos de nuestros revisores de la primera y segunda edición (sus afiliaciones en el momento de la revisión se enlistan entre paréntesis).

David Falconer (Universidad Estatal de California en Fullerton)
David Finkel (Worcester Polytechnic)
H. E. Dunsmore (Universidad de Purdue)
Jim Schmolze (Universidad de Tufts)
Gene Spafford (Universidad de Purdue)
Clovis Tondo (Corporación IBM y profesor visitante en la Universidad Nova)
Jeffrey Esakov (Universidad de Pensilvania)
Tom Slezak (Universidad de California, Lawrence Livermore National Laboratory)
Gary A. Wilson (Gary A. Wilson & Associates y Universidad de California, Extensión Berkeley)
Mike Kogan (Corporación IBM; Principal arquitecto de OS/2 2.0 de 32 bits)
Don Kostuch (retirado Corporación IBM; ahora instructor mundial en C, C++ y programación orientada a objetos)
Ed Lieblein (Universidad Nova)
John Carroll (Universidad Estatal de San Diego)
Alan Filipski (Universidad Estatal de Arizona)
Greg Hidley (Universidad Estatal de San Diego)
Daniel Hirschberg (Universidad de California en Irvine)
Jack Tan (Universidad de Houston)
Richard Alpert (Universidad de Boston)
Eric Bloom (Universidad Bentley College)

Estas personas escudriñaron todos los aspectos del texto e hicieron docenas de valiosas sugerencias para mejorar la exactitud y totalidad de la presentación.

Debemos una especial nota de agradecimiento al Dr. Graem Ringwood, Computer Science Dept., QMW Universidad de Londres. El doctor Ringwood envió continuamente sugerencias constructivas mientras impartía sus cursos basándose en nuestro libro. Sus comentarios y críticas jugaron un papel importante en la conformación de la segunda edición.

Tem Nieto contribuyó con largas horas de esfuerzo ayudándonos con la sección especial "Cómo construir su propio compilador" del final del capítulo 12.

También nos vemos en la obligación de agradecer a los muchos profesores, instructores, estudiantes y profesionales que enviaron sus comentarios sobre la primera edición: MacRae, Joe, Sysop del foro Autodesk AutoCad de CompuServe; McCarthy, Michael J., Director de Undergraduate Programs, Universidad de Pittsburgh; Mahmoud Fath El-Den, Departmento de Matemáticas y Ciencias de la Computación, Universidad Estatal Ft. Hays; Rader, Cyndi, Universidad Estatal Wright; Soni, Manish, Universidad de Tufts(estudiante); Bullock, Tom, Departamento de Ingeniería Eléctrica, Universidad de Florida en Gainesville (Professor of EE); Derruks, Jan, Hogeschool van Amsterdam, Technische Maritieme Faculteit, Amsterdam; Duchan, Alan, Chair, MIS Department, Escuela de Negocios Richard J. Wehle, Universidad Canisius; Kenny, Bárbara

T., Departamento de Matemáticas, Universidad Estatal de Boise; Riegelhaupt-Herzig, Scott P., Colegio Metropolitano de la Universidad de Boston, Departamento de Ciencias de la Computación; Yean, Leong Wai, Universidad Tecnológica de Nanyang, División de Tecnología de Ciencias Aplicadas, Singapur (estudiante); Abdullah, Rosni, Universidad Sains Malaysia, Departamento de Ciencias de la Computación; Willis, Bob: Cohoon, Jim, Departamento de Ciencias de la Computación, Universidad de Virginia; Tranchant, Mark, Universidad de Southhampton, Gran Bretaña (estudiante); Martignoni, Stephane, Instituto Real de Tecnología, Suecia (estudiante); Spears, Marlene, Homebrewer (fabrica cerveza)(su esposo que es estudiante utiliza nuestro libro); French, Rev. Michael D.(SJ), Departamento de Ciencias de la Computación, Universidad Loyola, Maryland; Wallace, Ted, Departamento de Ruso y Física, Universidad Dartmouth (estudiante); Wright, Kevin, Universidad de Nebraska (estudiante); Elder, Scott, (estudiante); Schneller, Jeffrey; Byrd, William, Department de Ingeniería Industrial y de Sistemas, Universidad de Florida (estudiante); Naiman, E. J., Compuware; Sedgwick, Arthur E., Dr., Departamento de Matemáticas, Estadística y Ciencias de la Computación, Universidad Dalhouside, Halifax, Nueva Escocia (usuario); Holsberg, Peter J., Profesor, Tecnología de Ingeniería, Computación y Matemáticas, Universidad Comunitaria del Condado de Mecer; Pont, Michel J., DR., Lecturer, Departamento de Ingeniería, Universidad de Leicester, Inglaterra; Linney, John, Profesor Asistente, Departamento of Ciencias de la Computación, Universidad Queen Mary and Westfield, Londres; Zipper, Herbert J., Profesor, Departamento de Ingeniería, SUNY Farmingdale; Humenik, Keith, Universidad de Maryland, Campus, Baltimore; Beeson, Michael, Departamento de Matemáticas y Ciencias de la Computación, Universidad Estatal de San José; Gingo, Peter J., Dr., Departamento de Ciencias Matemáticas, Universidad Buchtel de Artes y Ciencias; y Vaught, Lloyd, Departamento de Ciencias de la Computación, Universidad Modesto Junior.

Los autores desean hacer extensivo su especial agradecimiento a Ed Lieblein, una de las principales autoridades en el mundo sobre ingeniería de software, por su extraordinaria revisión sobre partes del material relativo a C++ y a programación orientada a objetos. El Dr. Lieblein es amigo y colega de HMD en la Universidad Nova en Fort Lauderdale, Florida, donde trabaja como profesor de tiempo completo en ciencias de la computación. El Dr. Lieblein fue Director técnico de Tartan Laboratories, una de las organizaciones líder en el desarrollo de compiladores del mundo. Antes, ocupó el cargo de Director de Computer Software and Systems en la oficina del Secretario de Defensa de Estados Unidos. Como tal, administró la iniciativa de Software DoD, un programa especial para mejorar la capacidad de software de la nación en relación con sistemas futuros de misión crítica. Inició el programa STARS del Pentágono en relación con tecnología de software y reutilización, guió el programa Ada hacia la estandarización internacional, y desempeñó un papel importante en el establecimiento del Software Engineering Institute en la Universidad Carnegie Mellon. Es en verdad un privilegio especial para nosotros el poder trabajar con el Dr. Lieblein en la Universidad Nova.

También deseamos extender una nota especial de agradecimiento al Sr. Dr. Clovis Tondo de IBM Corporation, Profesor visitante en la Universidad Nova. El Dr. Tondo fue el jefe de nuestro equipo de revisión. Sus revisiones meticulosas y completas nos enseñaron mucho en relación con las sutilezas de C y C++. El Dr. Tondo es coautor del *C Answer Book* que contiene respuestas a los ejercicios existentes en —que además se utiliza ampliamente en conjunción con —*The C Programming Language,* libro clásico que escribieron Brian Kernighan y Dennis Ritchie.

Este texto se basa en la versión de C estandarizada a través de American National Standards Institute (ANSI) en los Estados Unidos y a través del International Standards Organization (ISO) a nivel mundial. Hemos utilizado de forma extensiva materiales del documento estándar ANSI con el permiso expreso por escrito de la American National Standards Institute. Sinceramente,

apreciamos la cooperación de Mary Clare Lynch, Directora de publicaciones de ANSI, para ayudarnos a obtener los permisos de publicación necesarios. Las figuras 5.6, 8.1, 8.5, 8.12, 8.17, 8.20, 8.22, 8.30, 8.36, 9.1, 9.3, 9.6, 9.9, 9.16, 10.7 y 11.6, y el apéndice A: Sintaxis de C, y el apéndice B: Biblioteca estándar, se condensaron y adaptaron con el permiso del American National Standard for Information Systems—Programming Language C, ANSI/ISO 9899: 1990. Se pueden adquirir copias de este estándar o norma del American National Standards Institute en 11 West 42nd Street, New York, NY 10036.

Por último, nos gustaría agradecerle a Bárbara y Abbey Deitel, por su cariño, comprensión y sus enormes esfuerzos en ayudarnos a preparar el manuscrito. Aportaron innumerables horas de esfuerzo; probaron todos los programas del texto, auxiliaron en todas las fases de preparación del manuscrito e hicieron revisión de estilo de todos los borradores del texto, hasta su publicación. Su revisión minuciosa impidió que se cometieran innumerables errores. Bárbara también hizo la investigación de las citas, y Abbey sugirió el título para el libro.

Asumimos completa responsabilidad por cualquiera de los errores que hayan quedado en este texto. Agradeceríamos sus comentarios, críticas, correcciones y sugerencias para su mejoría. Por favor envíe su sugerencias para mejorar y añadir a nuestra lista de prácticas sanas de programación, errores comunes de programación, sugerencias de rendimiento, sugerencias de portabilidad y observaciones de ingeniería de software. Reconoceremos a todos los que contribuyan en la siguiente emisión de nuestro libro. Por favor dirija toda su correspondencia a nuestra dirección electrónica:

```
deitel@world.std.com
```

o si no escríbanos a la siguiente dirección:

Harvey M. Deitel (autor)
Paul J. Deitel (autor)
c/o Computer Science Editor
College Book Editorial
Prentice Hall
Englewood Cliffs, New Jersey 07632

Responderemos de inmediato.

Harvey M. Deitel
Paul J. Dietel

COMO PROGRAMAR EN C/C++

1

Conceptos de computación

Objetivos

- Comprender los conceptos básicos de computación.
- Familiarizarse con diferentes tipos de lenguajes de programación.
- Familiarizarse con la historia del lenguaje de programación C.
- Concientizarse de la biblioteca estándar C.
- Comprender el entorno de desarrollo del programa C.
- Apreciar porqué es apropiado aprender C en un primer curso de programación.
- Apreciar porqué C proporciona una base para subsiguientes estudios de programación en general y de C++ en particular.

Las cosas están siempre mejor en su principio.
Blaise Pascal

Pensamientos elevados deben tener un lenguaje elevado.
Aristophanes

Nuestra vida se malgasta a causa de detalles ... simplifique, simplifique.
Henry Thoreau

Sinopsis

1.1 Introducción

¡Bienvenido a C! Hemos trabajado duro para crear lo que esperamos sinceramente resulte para usted una experiencia educativa, informativa y entretenida. C es un lenguaje difícil, que normalmente se enseña sólo a programadores experimentados, por lo que este libro resulta único entre los libros de texto de C:

- Es apropiado para personas técnicamente orientadas, con poca o ninguna experiencia de programación.
- Es apropiado para programadores experimentados, que deseen un tratamiento profundo y riguroso del lenguaje.

¿Cómo puede un solo libro ser atracativo para ambos grupos? La respuesta estriba en que el núcleo común del libro hace énfasis en la obtención de *claridad* en los programas, mediante técnicas probadas de "programación estructurada". Los que no son programadores aprenderán programación de la forma "correcta" desde el principio. Hemos intentado escribir de una forma clara y sencilla. El libro está profusamente ilustrado. Y quizá de forma más importante, es que el libro presenta un número sustancial de programas de C operantes y muestra las salidas producidas al ejecutar estos programas en una computadora.

Los primeros cuatro capítulos presentan los fundamentos de la computación, de la programación de computadoras y del lenguaje C de programación de computadoras. Estos análisis están enmarcados en una introducción a la programación de computadoras utilizando el método estructurado. Los principiantes que han asistido a nuestros cursos nos indican que el material de estos capítulos representa una base sólida para el tratamiento más profundo de C de los capítulos 5 hasta el capítulo 14. Los programadores experimentados, por lo general, leen rápido los primeros cuatro capítulos y entonces descubren que el tratamiento de C de los capítulos 5 al 14 es tan riguroso como estimulante. Aprecian en forma particular el detallado tratamiento de apuntadores, cadenas, archivos y estructuras de datos que aparecen en los capítulos siguientes.

Muchos programadores experimentados nos han dicho que encuentran útil el tratamiento que le damos a la programación estructurada. Han estado a menudo programando en un lenguaje estructurado como Pascal, pero debido a que nunca fueron formalmente iniciados en la programación estructurada, no están escribiendo el mejor código posible. Conforme aprenden C utilizando este libro, se hacen cada vez más capaces de mejorar su estilo de programación. Por tanto , ya sea que sea usted un neófito o un programador experimentado, tenemos aquí mucho para informarle, entretenerlo y estimularlo.

La mayor parte de las personas están familiarizadas con las cosas excitantes que hacen las computadoras. En este curso, aprenderá cómo ordenarle a las computadoras que las hagan. Es el *software* (es decir, las instrucciones que usted escribe para ordenarle a la computadora a que ejecute acciones y a que tome decisiones) quien controla a las computadoras (a menudo conocido como *hardware*), y uno de los lenguajes de desarrollo de software más populares hoy día es C. Este texto proporciona una introducción a la programación en ANSI C, la versión estandarizada en 1989 tanto en los Estados Unidos, a través del American National Standards Institute (ANSI), como en el resto del mundo, a través de la International Standards Organization (ISO).

El uso de computadoras se está incrementando prácticamente en todos los campos de actividad. En una era de costos siempre crecientes, los costos de computación han venido reduciéndose de forma sorprendente, debido a increíbles desarrollos, tanto en la tecnología de hardware como de software. Las computadoras, que hace 25 años podían llenar grandes habitaciones y costaban millones de dólares, pueden ser ahora inscritas en las superficies de chips de silicón de un tamaño menor que una uña, y que quizá cuestan unos cuantos dólares cada uno. De forma irónica, el silicón es uno de los materiales más abundantes sobre la tierra —es un ingrediente de la arena común. La tecnología de los chips de silicón ha convertido a la computación en algo tan económico, que en el mundo se utilizan aproximadamente 150 millones de computadoras de uso general, auxiliando a las personas en los negocios, la industria, el gobierno y en sus vidas personales. Y este número podría con facilidad duplicarse en unos pocos años.

¿Puede C ser enseñado en un primer curso de programación, lo que se supone que es el auditorio para este libro? Así lo pensamos. Hace dos años tomamos este reto, cuando Pascal era el lenguaje que dominaba los primeros cursos de la ciencia de computación. Escribimos *Cómo programar en C*, primera edición de este texto. Cientos de universidades en todo el mundo han utilizado *Cómo programar en C*. Los cursos basados en ese libro han comprobado ser de igual eficacia que sus predecesores, basados en Pascal. No se han observado diferencias significativas, a excepción quizá que sus alumnos están más motivados, porque saben que tienen más probabilidades de utilizar C en vez de Pascal en sus cursos de niveles superiores, así como en sus carreras. Los alumnos aprendiendo C también saben que estarán mejor preparados para aprender de forma rápida C++. C++ es un super conjunto del lenguaje C, orientado a programadores que desean escribir programas orientados a objetos. Diremos más en relación con C++ en la Sección 1.14. •

De hecho, C++ es motivo de tanto interés hoy día, que en el capítulo 15 hemos decidido incluir una introducción detallada a C++ y a la programación orientada a objetos. Un fenómeno interesante, que está ocurriendo en el mercado de los lenguajes de programación, es que muchos de los proveedores clave, ahora sólo ponen en el mercado un producto combinado C/C++, en vez de ofrecer productos por separado. Esto le da a los usuarios la capacidad de continuar programando en C si así lo desean, y cuando lo consideren apropiado emigrar de forma gradual hacia C++.

Entonces, ¡ahí lo tiene! Está a punto de iniciar un camino estimulante y esperamos, lleno de satisfacciones. Conforme vaya avanzando, si desea comunicarse con nosotros, envíenos correo electrónico por Internet a `deitel@world.std.com`. Haremos toda clase de esfuerzos para responder de forma rápida. ¡Buena suerte!

1.2 ¿Qué es una computadora?

Una *computadora* es un dispositivo capaz de ejecutar cálculos y tomar decisiones lógicas a velocidades millones y a veces miles de millones de veces más rápidas de lo que pueden hacerlo los seres humanos. Por ejemplo, muchas de las computadoras personales de hoy día, pueden ejecutar decenas de millones de adiciones por segundo. Una persona utilizando una calculadora de escritorio pudiera requerir décadas para completar el mismo número de cálculos de lo que puede ejecutar una computadora personal poderosa en sólo un segundo. (Puntos a considerar: ¿cómo sabría si la persona sumó correctamente las cifras?, ¿cómo sabría si la computadora sumó correctamente las cifras?) Hoy día las *supercomputadoras* más rápidas pueden ejecutar cientos de miles de millones de sumas por segundo —¡aproximadamente tantos cálculos podrían ejecutar cientos de miles de personas en un año! y en los laboratorios de investigación ya están en funcionamiento computadoras de trillones de instrucciones por segundo.

Las computadoras procesan *datos* bajo el control de un conjunto de instrucciones que se conocen como *programas de computación*. Estos programas de computación guían a la computadora a través de conjuntos ordenados de acciones, especificados por personas a las que se conoce como *programadores de computadora*.

Los varios dispositivos (como el teclado, la pantalla, los discos, la memoria y las unidades procesadoras) que conforman un sistema de computación se conocen como el *hardware*. Los programas de computación que se ejecutan en una computadora se conocen como el *software*. Los costos del hardware han venido reduciéndose de forma drástica en años recientes, hasta el punto que las computadoras personales se han convertido en mercancía. Desafortunadamente, los costos de desarrollo de software han ido creciendo continuamente, conforme los programadores cada día desarrollan aplicaciones más poderosas y complejas, sin la capacidad de hacer mejorías paralelas en la tecnología del desarrollo del software. En este libro aprenderá métodos de desarrollo de software, que pueden reducir de forma sustancial los costos de desarrollo y acelerar el proceso de desarrollo de aplicaciones de software poderosas y de alta calidad. Estos métodos incluyen la *programación estructurada, la refinación por pasos de arriba a abajo, la funcionalización* y en el último capítulo del libro, *la programación orientada a objetos.*

1.3 Organización de la computadora

Si no se toman en cuenta las diferencias en apariencia física, virtualmente todas las computadoras pueden ser concebidas como divididas en seis *unidades lógicas* o secciones. Estas son:

1. *Unidad de entrada.* Esta es la sección "de recepción" de la computadora. Obtiene información (datos y programas de computadora) a partir de varios *dispositivos de entrada* y pone esta información a la disposición de las otras unidades, de tal forma que la información pueda ser procesada. La mayor parte de la información se introduce en las computadoras hoy día a través de teclados de tipo máquina de escribir.
2. *Unidad de salida.* Esta es la sección "de embarques" de la computadora. Toma la información que ha sido procesada por la computadora y la coloca en varios *dispositivos de salida* para dejar la información disponible para su uso fuera de la computadora. La mayor parte de la información sale de las computadoras hoy día mediante despliegue en pantallas o mediante impresión en papel.
3. *Unidad de memoria.* Esta es la sección de "almacén" de rápido acceso y de capacidad relativamente baja de la computadora. Retiene información que ha sido introducida a través de la unidad de entrada, de tal forma que esta información pueda estar de inmediato disponible para su proceso cuando sea necesario. La unidad de memoria también retiene información ya procesada, hasta que dicha información pueda ser colocada por la unidad de salida en dispositivos de salida. La unidad de memoria se conoce a menudo como *memoria o memoria primaria.*
4. *Unidad aritmética y lógica (ALU).* Esta es la sección de "fabricación" de la computadora. Es responsable de la ejecución de cálculos como es suma, resta, multiplicación y división. Contiene los mecanismos de decisión que permiten que la computadora, por ejemplo, compare dos elementos existentes de la unidad de memoria para determinar si son o no iguales.
5. *Unidad de procesamiento central (CPU).* Esta es la sección "administrativa" de la computadora. Es el coordinador de la computadora que es responsable de la supervisión de la operación de las demás secciones. El CPU le indica a la unidad de entrada cuándo debe leerse la información y colocarse en la unidad de memoria, le indica al ALU cuándo deberá utilizar información de la unidad de memoria en cálculos, y le indica a la unidad de salida cuándo enviar información de la unidad de memoria a ciertos dispositivos de salida.
6. *Unidad de almacenamiento secundario.* Esta es la sección de "almacén" a largo plazo de alta capacidad de la computadora. Los programas o los datos que no se estén utilizando de forma activa por otras unidades, están por lo regular colocados en dispositivos de almacenamiento secundario (como discos) en tanto se necesiten otra vez, es posible que sean horas, días, meses o inclusive años después.

1.4 Procesamiento por lotes, multiprogramación y tiempo compartido

Las primeras computadoras sólo eran capaces de ejecutar un *trabajo* o *tarea* a la vez. Esta forma de operación de las computadoras, a menudo se conoce como *procesamiento por lotes* de un solo usuario. La computadora ejecuta un programa a la vez al procesar datos en grupos o en *lotes*. En estos sistemas primarios, los usuarios por lo general entregaban sus trabajos al centro de cómputo en paquetes de tarjetas perforadas. Los usuarios a menudo tenían que esperar horas e inclusive días, antes que se les devolvieran impresiones a sus escritorios.

Conforme las computadoras se hicieron más poderosas, se hizo evidente que el procesamiento por lotes de un solo usuario rara vez utilizaba los recursos de la computadora de manera eficazmente. En vez de ello, se pensó que muchos trabajos o tareas podían hacer que *compartieran* los recursos de la computadora para obtener mejor utilización. Esto se conoce como *multiprogramación.* La multiprogramación implica la operación "simultánea" de muchos trabajos en una computadora —la computadora comparte sus recursos entre los trabajos que compiten por su

atención. En el caso de los primeros sistemas de multiprogramación, los usuarios aún entregaban los trabajos en paquetes de tarjetas perforadas, y tenían que esperar horas o días para los resultados.

En los años 60, varios grupos en la industria y en las universidades se hicieron pioneros en el concepto de *tiempo compartido*. El tiempo compartido es un caso especial de la multiprogramación, en el cual los usuarios tienen acceso a la computadora a través de dispositivos de entrada/salida o *terminales*. En un sistema típico de computadora a tiempo compartido, pudieran existir docenas e inclusive cientos de usuarios, compartiendo a la vez la computadora. La computadora de hecho, no ejecuta las órdenes de todos los usuarios en forma simultánea. En vez de ello, ejecuta una pequeña porción del trabajo de un usuario, y de inmediato pasa a darle servicio al siguiente. La computadora hace esto tan aprisa, que puede darle servicio a cada usuario varias veces por segundo. Por tanto , los usuarios *parece* que estuvieran ejecutando su trabajo de forma simultánea.

1.5 Computación personal, computación distribuida y computación cliente/servidor

En 1977, Apple Computer popularizó el fenómeno de la *computación personal*. Al principio, era el sueño de los aficionados a la computación. Las computadoras se hicieron lo bastante económicas para que las personas las adquirieran para su uso personal o de negocios. En 1981, IBM, el fabricante más grande del mundo de computadoras, introdujo la computadora personal IBM. Literalmente de la noche a la mañana, la computación personal se legitimizó en negocios, industrias y organizaciones gubernamentales.

Pero estas computadoras eran unidades "independientes" —las personas hacían el trabajo en sus propias máquinas y a continuación transportaban discos de ida y vuelta para compartir la información. Aunque las primeras computadoras personales no eran lo bastante poderosas para compartirse entre varios usuarios, estas máquinas podían ser enlazadas juntas en redes de computación, a veces mediante líneas telefónicas y otras en redes de área local dentro de una organización. Esto condujo al fenómeno de la *computación distribuida*, en la cual la carga de trabajo de computación de una organización, en vez de ser ejecutada de manera estrictamente en alguna instalación central de cómputo, se distribuye sobre la red a los lugares donde en realidad se ejecuta el trabajo de la organización. Las computadoras personales eran lo bastante poderosas para manejar las necesidades de cómputo de usuarios individuales, así como para manejar las tareas básicas de comunicación, de pasar la información electrónicamente de ida y vuelta.

Hoy día las computadoras personales más potentes son tan poderosas como las máquinas de un millón de dólares de hace una década. Las máquinas de escritorio más poderosas que se conocen como *estaciones de trabajo* proporcionan a usuarios individuales enormes capacidades. La información puede compartirse, con facilidad a través de redes de cómputo, donde algunas computadoras denominadas *servidores de archivo* ofrecen un almacén común de programas y de datos, que pueden ser utilizados por computadoras *cliente* distribuidas a todo lo largo de la red, y de ahí el término de *computación cliente/servicio*. C y C++ se han convertido en los lenguajes de elección para escribir software para sistemas operativos, para redes de computación y para aplicaciones distribuidas cliente/servidor.

1.6 Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel

Los programadores escriben instrucciones en diferentes lenguajes de programación, algunos comprensibles de forma directa por la computadora y otros que requieren pasos intermedios de *traducción*. Existen hoy día cientos de lenguajes de computadora. Estos pueden ser categorizados en tres tipos generales:

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

Cualquier computadora sólo puede entender directamente su propio *lenguaje máquina*. El lenguaje máquina es el "lenguaje natural" de una computadora particular. Está relacionado íntimamente con el diseño del hardware de esa computadora. Los lenguajes máquina, por lo general consisten de cadenas de números (al final reducidos a unos y a ceros) que instruyen a las computadoras para que ejecuten sus operaciones más elementales, una a la vez. Los lenguajes máquina son *dependientes de la máquina*, es decir, un lenguaje máquina particular puede ser utilizado en sólo un tipo de computadora. Los lenguajes máquina son difíciles de manejar por los seres humanos, como puede verse en la siguiente sección de un programa de lenguaje máquina, que añade pago por tiempo extra a la nómina base y almacena el resultado en la nómina bruta.

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

Conforme las computadoras se hicieron más populares, se hizo aparente que la programación en lenguaje máquina era demasiado lenta y tediosa para la mayor parte de los programadores. En vez de utilizar las cadenas de números que las computadoras pueden entender de forma directa, los programadores empezaron a usar abreviaturas similares al inglés para representar las operaciones elementales de la computadora. Estas abreviaturas similares al inglés formaron la base de los *lenguajes ensambladores*. Se desarrollaron *programas de traducción* denominados *ensambladores* para convertir los programas de lenguaje ensamblador a lenguaje máquina a la velocidad de las computadoras. La sección siguiente de un programa de lenguaje ensamblador también añade el pago de horas extras a la nómina base y almacena el resultado en una nómina bruta, pero con mayor claridad que su equivalente en lenguaje máquina:

```
LOAD    BASEPAY
ADD     OVERPAY
STORE   GROSSPAY
```

La utilización de las computadoras aumentó con rapidez con la llegada de los lenguajes ensambladores, pero estos aún necesitaban de muchas instrucciones para llevar a cabo inclusive las tareas más sencillas. Para acelerar el proceso de programación, se desarrollaron *lenguajes de alto nivel*, en los cuales se podían escribir simples enunciados para poder llevar a cabo tareas sustanciales. Los programas de traducción que convierten los programas de lenguaje de alto nivel al lenguaje máquina se llaman *compiladores*. Los lenguajes de alto nivel le permiten a los programadores escribir instrucciones que parecen prácticamente como el inglés de todos los días y contiene notaciones matemáticas por lo común utilizadas. Un programa de nómina escrito en un lenguaje de alto nivel pudiera contener un enunciado como el siguiente:

```
grosspay = basepay + overTimePay
```

Es obvio que, los lenguajes de alto nivel son mucho más deseables desde el punto de vista del programador que los lenguajes máquina o los ensambladores. C y C++ son, de entre los lenguajes de alto nivel, los más poderosos y los más utilizados.

1.7 La historia de C

C evolucionó a partir de dos lenguajes previos, BCPL y B. BCPL fue desarrollado en 1967 por Martin Richards, como un lenguaje para escribir software y compiladores de sistemas operativos.

Ken Thompson modeló muchas características de su lenguaje B siguiendo sus contrapartidas en BCPL, y utilizó B en 1970 para crear versiones iniciales del sistema operativo UNIX en los Laboratorios Bell, sobre una computadora PDP-7 de DEC. Tanto BCPL como B eran lenguajes "sin tipo" cada elemento de datos ocupaba una palabra "en memoria" y quedaba a cargo del programador el tratar un elemento de datos como si se tratara de un número entero o de un número real.

El lenguaje C fue derivado del lenguaje B por Dennis Ritchie, de los Laboratorios Bell, y al inicio se implantó en 1972 en una computadora PDP-11 de DEC. C al inicio se hizo muy conocido como lenguaje de desarrollo del sistema operativo UNIX. Hoy día, virtualmente todos los sistemas principales están escritos en C y/o C++. A lo largo de las últimas dos décadas, C se ha hecho disponible para la mayor parte de las computadoras. C es independiente del hardware. Con un diseño cuidadoso, es posible escribir programas en C que sean *portátiles* hacia la mayor parte de las computadoras. C utiliza muchos de los conceptos importantes de BCPL y de B, además de añadir los tipos de datos y otras características poderosas.

Hacia finales de los 70, C había evolucionado a lo que hoy se conoce como C "tradicional". La publicación en 1978 del libro de Kernighan y de Ritchie, *The C Programming Language*, atrajo gran atención sobre este lenguaje. Esta publicación se convirtió en uno de los libros científicos de computadoras de más éxito de todos los tiempos.

La expansión rápida de C sobre varios tipos de computadoras (denominadas a veces *plataformas de hardware*) trajo consigo muchas variantes. Estas eran similares, pero a menudo no eran compatibles. Esto resultaba en un problema serio para los desarrolladores de programas, que necesitaban escribir códigos que pudieran funcionar en varias plataformas. Se hizo cada vez más evidente que era necesaria una versión estándar de C. En 1983, se creó el comité técnico X3J11, bajo el American National Standards Committee on Computers and Information Processing (X3), para "proporcionar una definición no ambigua e independiente de máquina del lenguaje". En 1989 el estándar o norma quedó aprobado. El documento se conoce como ANSI/ISO 9899: 1990. Se pueden ordenar copias de este documento del American National Standards Institute, cuya dirección se menciona en el prefacio de este texto. La segunda edición de Kernighan y Ritchie, que se publicó en 1988, refleja esta versión que se conoce como ANSI C, la cual ahora se utiliza en todo el mundo (Ke88).

Sugerencia de portabilidad 1.1

Dado que C es un lenguaje independiente del hardware y ampliamente disponible, las aplicaciones que están escritas en C pueden ejecutarse con poca o ninguna modificación en una amplia gama de sistemas distintos de cómputo.

1.8 La biblioteca estándar de C

Como aprenderá en el capítulo 5, los programas C consisten de módulos o piezas que se denominan *funciones*. Usted puede programar todas las funciones que necesita para formar un programa C, pero la mayor parte de los programadores de C aprovechan una gran recopilación de funciones existentes, que se conocen como la *Biblioteca estándar C*. Entonces, para aprender el "universo" C, de hecho existen dos partes. El primero es aprender el lenguaje C mismo, y el segundo es aprender como utilizar las funciones de la Biblioteca estándar C. A lo largo de este libro, analizarémos muchas de estas funciones. El apéndice B (condensado y adaptado a partir del documento estándar de ANSI C mismo) enumera todas las funciones disponibles en la biblioteca estándar C. El libro escrito por Plauger (Pl92) es de lectura obligatoria para aquellos programadores que necesitan un profundo conocimiento de las funciones de biblioteca, cómo implantarlas y utilizarlas para escribir código portátil.

Usted será estimulado en este curso a utilizar un método de *bloques constructivos* para la creación de programas. Evite volver a inventar la rueda. Utilice piezas existentes —esto se conoce como *reutilización del software* y como veremos en el capítulo 15, es piedra angular del campo en desarrollo de la programación orientada a objetos. Cuando esté programando en C, por lo regular utilizará los siguientes bloques constructivos:

- Funciones de la biblioteca estándar de C
- Funciones que debe crear usted mismo
- Funciones a su disposición, que crearan otras personas.

La ventaja de crear su propias funciones, es que sabrá con exactitud como funcionan. Estará en condición de examinar el código C. La desventaja es el esfuerzo y el tiempo que se gasta en el diseño y el desarrollo de nuevas funciones.

El uso de funciones existentes elimina el tener que volver a inventar la rueda. En el caso de las funciones estándar ANSI, usted sabe que están escritas con cuidado, y lo sabe porque está utilizando funciones que están disponibles en todas las implantaciones de ANSI C y, por lo mismo sus programas tendrán una mayor oportunidad de ser portátiles.

Sugerencia de rendimiento 1.1

El usar funciones de la biblioteca estándar ANSI, en vez de escribir sus propias versiones comparables, puede mejorar el rendimiento de los programas, porque estas funciones están escritas de forma cuidadosa para que se ejecuten con eficacia.

Sugerencia de portabilidad 1.2

El usar funciones de biblioteca estándar ANSI en vez de escribir sus propias versiones comparables, puede mejorar la portabilidad del programa porque estas funciones están incluidas en casi todas las implantaciones de ANSI C.

1.9 Otros lenguajes de alto nivel

Se han desarrollado cientos de lenguajes de alto nivel, pero sólo unos pocos han alcanzado una amplia aceptación. *FORTRAN* (FORmula TRANslator) fue desarrollado por IBM entre 1954 y 1957, para uso en aplicaciones científicas y de ingeniería, que requieran de complejos cálculos matemáticos. FORTRAN es aún muy utilizado.

COBOL (COmmon Business Oriented Language) fue desarrollado en 1959 por un grupo de fabricantes de computadoras y de usuarios industriales y de gobierno. COBOL se utiliza sobre todo en aplicaciones comerciales, que requieren manipulación precisa y eficiente de grandes cantidades de datos. Hoy día, más de la mitad del software de negocios se programa aún en COBOL. Mas de un millón de personas están empleadas como programadores de COBOL.

Pascal fue diseñado casi al mismo tiempo que C. Se concibió para uso académico. En relación con Pascal diremos más en la sección siguiente.

1.10 Programación estructurada

Durante los años 60, el desarrollo de software se encontró con severas dificultades. Por lo regular los programas de entrega del software se retrasaban, sus costos excedían en gran medida los presupuestos, y los productos terminados no eran confiables. Las personas empezaron a darse

cuenta que el desarrollo de software era una actividad mucho más compleja de lo que se habían imaginado. La actividad de investigación de los años 60 dió como resultado la evolución de la *programación estructurada* —un método disciplinado de escribir programas que sean claros, que se demuestre que son correctos y fáciles de modificar. En el capítulo 3 y capítulo 4 se da una visión general de los principios de la programación estructurada. El resto del texto analiza el desarrollo de los programas estructurados de C.

Uno de los resultados más tangibles de esta investigación, fue el desarrollo en 1971 hecho por el profesor Nicklaus Wirth del lenguaje de programación Pascal. Pascal, al que se le da ese nombre en honor a Blaise Pascal, matemático y filósofo del siglo XVII, fue diseñado para la enseñanza de la programación estructurada en entornos académicos, y se convirtió con rapidez en el lenguaje introductorio de programación de la mayor parte de las universidades. Por desgracia, el lenguaje carece de muchas características necesarias para hacerlo útil en aplicaciones comerciales, industriales y de gobierno, por lo que no ha sido muy aceptado en esos últimos ámbitos. Quizá la historia registre que la verdadera significación del Pascal fue su elección como base del lenguaje de programación *Ada*.

Ada fue desarrollado bajo el patrocinio del Departamento de Defensa de los Estados Unidos (DOD) durante los años 70 y principio de los 80. Se estaban utilizando cientos de lenguajes distintos para producir los sistemas masivos de software de comando y de control de DOD. DOD deseaba un solo lenguaje que pudiera llenar sus objetivos. Pascal fue seleccionado como base, pero el lenguaje final Ada, es muy distinto de Pascal. Este lenguaje se llamó así en honor a Lady Ada Lovelace, hija del poeta Lord Byron. A Lady Lovelace se le da por lo general el crédito de haber escrito el primer programa de computación del mundo a principios de 1800. Una capacidad importante de Ada se conoce como *multitareas*; esto permite a los programadores especificar qué actividades deben ocurrir en paralelo. Otros lenguajes muy utilizados de alto nivel que hemos analizado incluyendo C y C++ permiten al programador escribir programas que sólo ejecuten una actividad a la vez. Está pendiente ver si Ada cumple sus objetivos de producir un software confiable y reducir de forma sustancial los costos de desarrollo y mantenimiento del software.

1.11 Los fundamentos del entorno de C

Todos los sistemas C consisten, en general, de tres partes: el entorno, el lenguaje y la biblioteca estándar C. En el siguiente análisis se explica el entorno típico de desarrollo de C, que se muestra en la figura 1.1.

Los programas C casi siempre pasan a través de seis fases para su ejecución (figura 1.1). Estas fases son: *editar, preprocesar, compilar, enlazar, cargar* y *ejecutar*. Nos estamos concentrando en este momento en el sistema típico UNIX, basado en C. Si usted no está utilizando un sistema UNIX, refiérase a los manuales de su sistema, o pregunte a su instructor cómo llevar a cabo estas tareas en su entorno.

La primera fase consiste en editar un archivo. Esto se ejecuta con un *programa de edición*. El programador escribe un programa C utilizando el editor, y si es necesario hace correcciones. El programa a continuación se almacena en un dispositivo de almacenamiento secundario, como sería un disco. Los nombres de archivo de los programas C deben terminar con la extensión `.c` Dos editores muy utilizados en sistemas UNIX son `vi` y `emacs`. Los paquetes de software C/C++, como son Borland C++ para las PC de IBM y compatibles, y Symantec C++ para el Macintosh de Apple, tienen editores incorporados, que están integrados en el entorno de programación. Suponemos que el lector sabe cómo editar un programa.

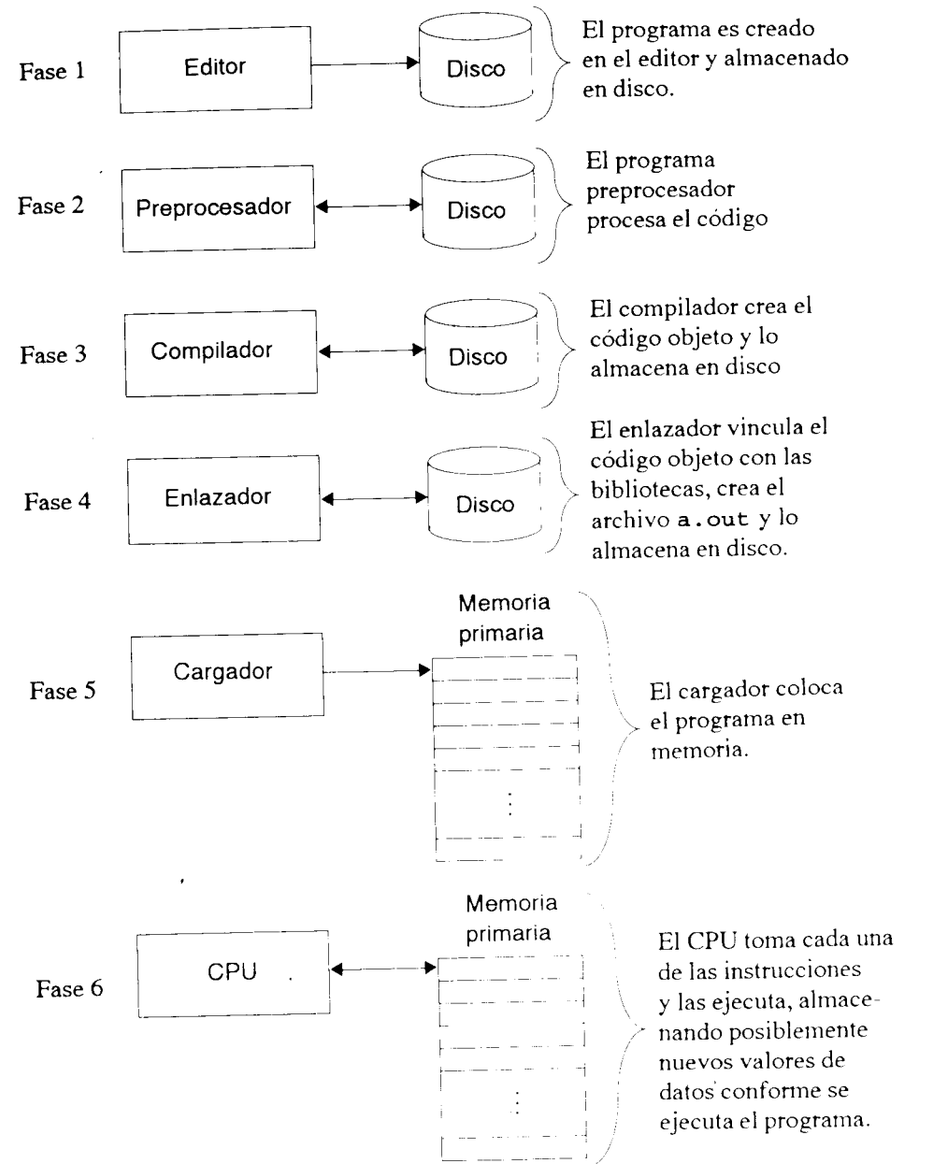


Fig. 1.1 Un entorno típico de C.

A continuación el programador da el comando de *compilar* el programa. El compilador traduce el programa C a código de lenguaje máquina (que también se conoce como *código objeto*). En un sistema C, un programa *preprocesador* ejecuta de forma automática antes de la fase de

traducción. El preprocesador C obedece comandos especiales que se llaman *directrices de preprocesador,* que indican que antes de su compilación se deben de ejecutar ciertas manipulaciones sobre el programa. Estas manipulaciones por lo regular consisten en la inclusión de otros archivos en el archivo a compilar y en el remplazo de símbolos especiales con texto de programa. Las directrices de preprocesador más comunes se analizan en los primeros capítulos; una presentación detallada de todas las características del preprocesador aparece en el capítulo 13. El preprocesador es invocado de manera automática por el compilador, antes que el programa sea convertido a lenguaje máquina.

La cuarta fase se conoce como *enlace*. Los programas C por lo general contienen referencias a funciones definidas en algún otro lugar como en bibliotecas estándar o en bibliotecas de un grupo de programadores que trabajen en un proyecto en particular. Entonces, el código objeto producido por el compilador C típicamente contendrá "huecos" debido a estas partes faltantes. Un *enlazador* vinculará el código objeto con el código de las funciones faltantes para producir una *imagen ejecutable* (sin ninguna parte faltante). En un sistema típico basado en UNIX, el comando para compilar y enlazar un programa es **cc**. Por ejemplo, para compilar y enlazar un programa de nombre **welcome.c**, escriba:

```
cc welcome.c
```

en la indicación de UNIX y presione la tecla de entrar. Si el programa compila y enlaza en forma correcta, se producirá un archivo con el nombre de **a.out**. Esta es la imagen ejecutable de nuestro programa **welcome.c**.

La quinta fase se llama *cargar*. Antes de que un programa pueda ser ejecutado, el mismo debe de ser colocado en memoria. Esto se lleva a cabo mediante el *cargador*, que toma la imagen ejecutable del disco y la transfiere a la memoria.

Por último, la computadora, bajo el control de su CPU, ejecuta el programa, una instrucción a la vez. Para cargar y ejecutar el programa en un sistema UNIX, escribimos **a.out** en la indicación de UNIX, y oprimimos la tecla de entrar.

La mayor parte de los programas en C reciben o proporcionan datos. Ciertas funciones de C toman sus entradas de **stdin** (*el dispositivo de entrada estándar*) que por lo regular queda asignado al teclado, pero **stdin** puede ser conectado a otro dispositivo. Los datos salen a **stdout** (el *dispositivo estándar de salida*) que por lo regular es la pantalla de la computadora, pero que puede ser conectado a otro dispositivo. Cuando decimos que un programa imprime un resultado, queremos decir que el resultado aparece desplegado en una pantalla. Los datos pueden salir a otros dispositivos, como son discos o impresoras. Existe también un *dispositivo estándar* de error, que se conoce como **stderr**. El dispositivo **stderr** (conectado a la pantalla), se utiliza para el despliegue de los mensajes de error. Es común encaminar los datos regulares de salida, es decir **stdout**, a un dispositivo distinto de la pantalla, mientras se conserva **stderr** asignado a la pantalla, de tal forma que el usuario de inmediato quede informado de errores.

1.12 Notas generales en relación con C y este libro

C es un lenguaje difícil. En ocasiones los programadores experimentados de C se enorgullecen de ser capaces de crear usos extraños, retorcidos y complicados del lenguaje. Esto es una mala práctica de programación. Hace que los programas sean difíciles de leer, es probable que se comporten en forma extraña y sean más difíciles de probar y de depurar. Este libro está organizado para programadores neófitos, por lo que hacemos hincapié en escribir programas claros y bien

estructurados. Una de las metas clave de este libro es conseguir *claridad* en el programa a través de las técnicas probadas de la programación estructurada y a través de las muchas buenas prácticas de programación relacionadas.

Práctica sana de programación 1.1

Escriba sus programas C de una forma simple y sencilla. Esto a veces se conoce como KIS (del inglés "keep it simple"). No "le saque punta al lenguaje" intentando "rarezas".

Habrá escuchado que C es un lenguaje portátil, y que los programas escritos en C pueden ser ejecutados en muy diversas computadoras. *La portabilidad es una meta elusiva*. El documento estándar ANSI (An90) enlista 11 páginas de temas sutiles de portabilidad. Se han escrito libros completos tratando el tema de la portabilidad en C (Ja89)(Ra90).

Sugerencia de portabilidad 1.3

Aunque es posible escribir programas portátiles, existen muchos problemas entre diferentes implantaciones de C y diferentes computadoras, que dificultan la portabilidad a alcanzar. La simple escritura de programas en C no garantiza la portabilidad.

Hemos hecho un cuidadoso recorrido del documento estándar ANSI C y auditado nuestra presentación contra éste, en relación con su integridad y exactitud. Sin embargo, C es un lenguaje muy rico, y existen algunas sutilezas del lenguaje y algunos temas avanzados que no hemos cubierto. Si necesita detalles técnicos adicionales sobre ANSI C, sugerimos que lea el documento estándar ANSI C mismo o el manual de referencia en Kernighan y Ritchie (Ke88).

Hemos limitado nuestras exposiciones a ANSI C. Muchas características de ANSI C no son compatibles con implantaciones anteriores de C, por lo que usted podrá encontrar que algunos de los programas de este texto no funcionan en compiladores C antiguos.

Práctica sana de programación 1.2

Lea los manuales correspondientes a la versión de C que esté utilizando. Consulte con frecuencia estos manuales para asegurar que está consciente de la gran cantidad de características de C y que está utilizando estas características de forma correcta.

Práctica sana de programación 1.3

Su computadora y su compilador son buenos maestros. Si no está seguro de cómo funciona una característica de C, escriba un programa de muestra con dicha característica, compile y ejecute el programa y vea qué es lo que ocurre.

1.13 C Concurrente

En los Laboratorios Bell mediante continuos esfuerzos de investigación se han desarrollado otras versiones de C. Gehani (Ge89) ha desarrollado *C Concurrente* un superconjunto de C que incluye capacidades de especificación de actividades múltiples en paralelo. Lenguajes como C Concurrente y funciones de sistemas operativos que dan soporte a paralelismo a aplicaciones de usuarios, se harán cada vez más populares en la siguiente década, conforme vaya aumentando el uso de *multiprocesadores* (es decir, computadoras con más de un CPU). A la fecha de esta escritura, C Concurrente es primordialmente aún un lenguaje de investigación. Los cursos de sistemas operativos y los libros de texto (De90) por lo regular incluyen análisis sustancial de la programación concurrente.

1.14 Programación orientada a objetos y C++

Otro superconjunto de C, es decir C++, fue desarrollado por Stroustrup (St86) en los Laboratorios Bell. C++ proporciona un cierto número de características que "engalanan" el lenguaje C. Pero lo que es aún más importante, permite llevar a cabo *programación orientada a objetos*.

Los *objetos* son en esencia *componentes* de software reutilizables que modelan elementos del mundo real. Está en marcha una revolución en la comunidad del software. Sigue siendo una meta no alcanzable la elaboración de software rápida, correcta y económica, y esto en un momento en que las demandas del software nuevo y más poderoso están encumbrándose.

Los desarrolladores de software están descubriendo que, utilizando un diseño e implantación modular orientada a objetos, puede hacer que los grupos de desarrollo de software sean de 10 a 100 veces más productivos que lo que era posible mediante técnicas convencionales de programación.

Se han desarrollado muchos lenguajes orientados a objetos. Se cree en general que C++ se convertirá en el lenguaje dominante de implantación de sistemas en la segunda parte de la década de los 90.

Muchas personas sienten que la mejor estrategia educacional hoy día es dominar C, y a continuación estudiar C++. Por ello, hemos dispuesto los capítulos 15 al 21, introduciendo programación orientada a objetos y C++. Esperamos que el lector los encuentre valiosos, y que los capítulos lo estimularán a seguir estudios posteriores de C++ una vez que haya terminado el estudio de este libro.

Resumen

- El software (es decir las instrucciones que usted escribe para darle órdenes a la computadora para que ejecute acciones y tome decisiones) es el que controla las computadoras (que a menudo se conocen como hardware).
- ANSI C es la versión del lenguaje de programación C estandarizada en 1989, tanto en los Estados Unidos, a través del American National Standards Institute (ANSI), como en todo el mundo a través de la International Standards Organization (ISO).
- Computadoras que hace 25 años pudieran haber llenado grandes habitaciones y costado millones de dólares, pueden ser ahora inscritas en la superficie de chips de silicón más pequeños que una uña, y que quizá cuestan cada una de ellas unos cuantos dólares.
- Aproximadamente se utilizan 150 millones de computadoras de uso general en todo el mundo, auxiliando a la personas en: negocios, industria, gobierno y en sus vidas personales. Esta cifra podría duplicarse con facilidad en pocos años.
- Una computadora es un dispositivo capaz de llevar a cabo cálculos y tomar decisiones lógicas a velocidades millones y a veces miles de millones más rápidas de lo que lo pueden hacer los seres humanos.
- Las computadoras procesan datos bajo control de programas de computación.
- Los diversos dispositivos (como: teclado, pantalla, discos, memoria y las unidades de procesamiento), que forman un sistema de cómputo, se conocen como hardware.
- Los programas de computación que se ejecutan en una computadora se conocen como software.
- La unidad de entrada es la sección de "recepción" de la computadora. La mayor parte de la información hoy día entra en las computadoras a través de teclados de tipo máquina de escribir.

- La unidad de salida es la sección de "embarques" de la computadora. La mayor parte de la información sale de las computadoras hoy día desplegándose en pantallas o imprimiéndose en papel.
- La unidad de memoria es la sección de "almacén" de la computadora, y a menudo se llama memoria o memoria primaria.
- La unidad aritmética y lógica (ALU) ejecuta cálculos y toma decisiones.
- La unidad de procesamiento central (CPU) es el coordinador de la computadora y es responsable de supervisar la operación de las otras secciones.
- Los programas y los datos que no están en uso activo por otras unidades, por lo regular están colocados en dispositivos de almacenamiento secundario (como discos) hasta que se necesitan de nuevo.
- En el procesamiento por lotes de un solo usuario, la computadora ejecuta un programa a la vez, mientras procesa datos en grupos o en lotes.
- La multiprogramación implica la operación "simultánea" de muchas tareas en la computadora —la computadora comparte sus recursos entre las tareas.
- El tiempo compartido es un caso especial de la multiprogramación, en el cual los usuarios tienen acceso a la computadora desde terminales. Al parecer los usuarios están operando en forma simultánea.
- Con la computación distribuida, la computación de una organización está distribuida, mediante redes a los lugares en donde es ejecutado el trabajo real de la organización.
- Los servidores de archivos almacenan programas y datos que pueden ser compartidos por las computadoras cliente, distribuidas a todo lo largo de la red, de ahí el término computación cliente/servidor.
- Cualquier computadora sólo puede entender de forma directa su propio lenguaje máquina.
- Los lenguajes máquina por lo general consisten de cadenas de números (reducidos en forma última a unos y a ceros) que instruyen a las computadoras para que ejecuten sus operaciones más elementales, una a la vez. Los lenguajes máquina son dependientes de la máquina.
- Las abreviaturas similares al inglés forman la base de los lenguajes ensambladores. Los ensambladores traducen los programas en lenguaje ensamblador a lenguaje máquina.
- Los compiladores traducen programas de lenguajes de alto nivel a lenguaje máquina. Los lenguajes de alto nivel contienen palabras inglesas y notaciones matemáticas convencionales.
- C se conoce como el lenguaje de desarrollo del sistema operativo UNIX.
- Es posible escribir programas en C que resulten portátiles para la mayor parte de las computadoras.
- El estándar ANSI C fue aprobado en 1989.
- FORTRAN (FORmula TRANslator) se utiliza para aplicaciones matemáticas.
- COBOL (COmmon Business Oriented Language) se usa sobre todo para aplicaciones comerciales, que requieren de una manipulación precisa y eficiente en datos de grandes cantidades.
- La programación estructurada es un método disciplinado para escribir programas que resulten claros, demostrablemente correctos y fáciles de modificar.
- Pascal fue diseñado para enseñar programación estructurada en entornos académicos.
- Ada fue desarrollada bajo el patrocinio del Departamento de Defensa de los Estados Unidos (DOD) utilizando a Pascal como base.

UNIVERSIDAD DE LAS AMERICAS
FACULTAD DE INGENIERIA

- La capacidad de *multitareas* de Ada le permite a los programadores especificar actividades en paralelo.
- Todos los sistemas C consisten de tres partes: entorno, lenguaje y las bibliotecas estándar. Las funciones de biblioteca no forman parte del lenguaje C mismo; estas funciones ejecutan operaciones como entrada/salida y cálculos matemáticos.
- Los programas C por lo regular pasan a través de seis fases para ser ejecutados: edición, preproceso, compilación, enlace, carga, y ejecución.
- El programador escribe un programa utilizando un editor, y si es necesario hace correcciones.
- Un compilador traduce un programa C en código de lenguaje máquina (o código objeto).
- El preprocesador de C obedece directrices de preprocesador, que por lo regular indican otros archivos que se deben incluir en el archivo a compilar, y que ciertos símbolos deben ser remplazados con texto de programa.
- Un enlazador vincula el código objeto con el código de funciones faltantes, para producir una imagen ejecutable (sin ninguna pieza faltante).
- Un cargador toma del disco una imagen ejecutable, y la transfiere a la memoria.
- Una computadora bajo el control de su CPU ejecutará el programa una instrucción a la vez.
- Ciertas funciones de C (como **`scanf`**), toman su entrada de **`stdin`** (el dispositivo de entrada estándar), que por lo regular está asignado al teclado.
- Los datos salen a **`stdout`** (el dispositivo de salida estándar), que por lo regular es la pantalla de la computadora.
- También existe un dispositivo estándar de error, que se conoce como **`stderr`**. El dispositivo **`stderr`** (la pantalla), se utiliza para el despliegue de mensajes de error.
- Aunque es posible escribir programas portátiles, existen muchos problemas entre diferentes implantaciones de C y diferentes computadoras, que pueden hacer la portabilidad difícil de alcanzar.
- El C concurrente es un superconjunto de C, que incluye capacidades para especificar la ejecución de varias actividades en paralelo.
- C++ proporciona capacidades para hacer programación orientada a objetos.
- Los objetos son en esencial componentes de software reutilizables, que modelan elementos del mundo real.
- Se cree que C++ se convertirá en el lenguaje dominante de implantación de sistemas en los años finales de 1990.

Terminología

`.c` extensión
Ada
ALU
ANSI C
método de bloques constructivos
C
preprocesador C
biblioteca estándar C
unidad aritmética y lógica (ALU)
ensamblador
lenguaje ensamblador
procesamiento por lotes
independiente de la máquina
lenguaje máquina
memoria
unidad de memoria

C++
unidad de procesamiento central (CPU)
claridad
cliente
computación cliente/servidor
COBOL
compilador
computadora
programa de computación
programador de computación
C Concurrente
CPU
datos
computación distribuida
editor
entorno
imagen ejecutable
ejecutar un programa
servidor de archivo
FORTRAN
función
funcionalización
hardware
plataforma de hardware
lenguaje de alto nivel
dispositivo de entrada
unidad de entrada
entrada/salida (E/S)
enlazador
cargador
unidades lógicas
dependiente de la máquina
multiprocesador
multiprogramación
multitareas
lenguaje natural de una computadora
objeto
código objeto
programación orientada a objetos
dispositivo de salida
unidad de salida
Pascal
computadora personal
portabilidad
memoria primaria
lenguaje de programación
ejecutar un programa
pantalla
software
reutilización del software
error estándar (**`stderr`**)
entrada estándar (**`stdin`**)
salida estándar (**`stdout`**)
programa almacenado
programación estructurada
supercomputadora
tarea
terminal
tiempo compartido
refinamiento con paso de arriba a abajo
programa traductor
UNIX
estación de trabajo

Prácticas sanas de programación

1.1 Escriba sus programas C de una forma simple y sencilla. Esto a veces se conoce como KIS (del inglés "keep it simple"). No "le saque punta al lenguaje" intentando "rarezas".

1.2 Lea los manuales correspondientes a la versión de C que esté utilizando. Consulte a menudo estos manuales para asegurarse que está consciente de la gran cantidad de características de C y que está utilizando estas características de forma correcta.

1.3 Su computadora y su compilador son buenos maestros. Si no está seguro de cómo funciona una característica de C, escriba un programa de muestra con dicha característica, compile y ejecute el programa y vea qué es lo que ocurre.

Sugerencias de portabilidad

1.1 Dado que C es un lenguaje independiente de hardware y muy disponible, las aplicaciones que están escritas en C pueden ejecutarse con poca o ninguna modificación en una amplia gama de distintos sistemas de cómputo.

1.2 El usar funciones de biblioteca estándar ANSI en vez de escribir sus propias versiones comparables, puede mejorar la portabilidad del programa porque estas funciones están incluidas en casi todas las implantaciones de ANSI C.

1.3 Aunque es posible escribir programas portátiles, existen muchos problemas entre diferentes implantaciones de C y diferentes computadoras, que dificultan la portabilidad a alcanzar. La simple escritura de programas en C no garantiza la portabilidad.

Sugerencia de rendimiento

1.1 El usar funciones de la biblioteca estándar ANSI, en vez de escribir sus propias versiones comparables, puede mejorar el rendimiento de los programas, porque estas funciones están escritas con cuidado para que se ejecuten con eficacia.

Ejercicios de autoevaluación

1.1 Llene los espacios en blanco en cada uno de los siguientes:

a) La compañía que en el mundo inició el fenómeno de la computadora personal fue_____.
b) La computadora que legitimizó la computadora personal en los negocios y en la industria fue la _____.
c) Las computadoras procesan datos bajo el control de conjuntos de instrucciones que se conocen como _____ de computación.
d) Las seis unidades lógicas claves de la computadora son los _____, _____, _____, _____, _____, y los _____.
e) _____ es un caso especial de multiprogramación en el cual los usuarios tienen acceso a la computadora mediante dispositivos que se conocen como terminales.
f) Las tres clases de lenguajes que se analizaron en el capítulo son_____, _____, y _____.
g) Los programas que traducen los programas de lenguaje de alto nivel al lenguaje máquina se llaman_____.
h) C se conoce ampliamente como el lenguaje de desarrollo del sistema operativo _____.
i) Este libro presenta la versión de C que se conoce como _____C la cual fue recientemente estandarizada a través de la American National Standards Institute.
j) El lenguaje _____ fue desarrollado por Wirth para la enseñanza de la programación estructurada en las universidades.
k) El departamento de la defensa desarrolló el lenguaje Ada con una capacidad que se conoce como _____ la cual permite a los programadores especificar que varias actividades pueden proceder en paralelo.

1.2 Llene los espacios en blanco en cada una de las siguientes oraciones en relación con el entorno de C.

a) Los programas de C se escriben por lo regular en una computadora utilizando un programa _____.
b) En un sistema C se ejecuta automáticamente un programa _____ antes que empiece la fase de traducción.
c) Los dos tipos más comunes de directrices de preprocesador son _____, y _____.
d) El programa _____ combina la salida del compilador con varias funciones de biblioteca a fin de producir una imagen ejecutable.
e) El programa _____ transfiere la imagen ejecutable del disco a la memoria.
f) Para cargar y ejecutar el programa recién compilado en un sistema UNIX, escriba _____.

Respuestas a los ejercicios de autoevaluación

1.1 a) Apple. b) Computadora personal de IBM. c) programas. d) unidad de entrada, unidad de salida, unidad de memoria, unidad de aritmética y lógica (ALU), unidad de procesamiento central (CPU), unidad de almacenamiento secundario. e) tiempo compartido. f) lenguajes de máquina, lenguajes ensambladores, y lenguajes de alto nivel. g) compiladores. h) UNIX. i) ANSI. j) Pascal. k) multitareas.

1.2 a) editor, b) preprocesador. c) incluyendo otros archivos en el archivo a compilarse, y remplazando símbolos especiales por texto de programa. d) enlazador. e) cargador. f) `a.out`.

Ejercicios

1.3. Clasifique cada uno de los elementos siguientes como hardware o software.

a) CPU
b) compilador C
c) ALU
d) preprocesador C
e) unidad de entrada
f) programa de procesamiento de texto

1.4 ¿Por qué escribiría usted un programa en un lenguaje independiente de máquina en vez de un lenguaje dependiente de máquina?. ¿Por qué sería más apropiado un lenguaje dependiente de máquina para escribir ciertos tipos de programas?

1.5 Los programas de traducción como son los ensambladores y los compiladores, convierten programas de un lenguaje (que se conoce como lenguaje *fuente*) a otro lenguaje (el cual se conoce como lenguaje *objeto*). Determine cuál de los siguientes enunciados son ciertos y cuáles falsos.

a) Un compilador traduce programas de alto nivel en lenguaje objeto.
b) Un ensamblador traduce programas de lenguaje fuente en programas de lenguaje máquina.
c) Un compilador convierte programas de lenguaje fuente en programas de lenguaje objeto.
d) Los lenguajes de alto nivel son normalmente dependientes de máquina.
e) Un programa de lenguaje máquina requiere de traducción antes de que el programa pueda ser ejecutado en una computadora.

1.6 Llene los espacios en blanco en cada uno de los enunciados siguientes.

a) Los dispositivos a partir de los cuales los usuarios tienen acceso a sistemas de computación de tiempo compartido por lo común se conocen como _____.
b) Un programa de computación que convierte programas de lenguaje ensamblador al lenguaje de máquina se llama _____.
c) La unidad lógica de la computadora que recibe información desde fuera de la misma para su uso se conoce como _____.
d) El proceso de instruir a la computadora para la solución de problemas específicos se llama _____.
e) ¿Qué tipo de lenguaje de computadora utiliza abreviaturas similares al inglés para instrucciones de lenguaje máquina? _____.
f) ¿Cuáles son las seis unidades lógicas de la computadora? _____.
g) ¿Cuál unidad lógica de la computadora envía información ya procesada por la computadora hacia varios dispositivos, de tal forma que esta información pueda ser utilizada fuera de la computadora? _____.
h) El nombre genérico de un programa que convierte programas escritos en un lenguaje específico de computadora al lenguaje máquina es_____.
i) ¿Qué unidad lógica de la computadora guarda la información? _____.
j) ¿Qué unidad lógica de la computadora ejecuta los cálculos? _____.
k) ¿Qué unidad lógica de la computadora toma decisiones lógicas? _____.

l) La abreviatura que por lo general se utiliza para la unidad de control de la computadora es _____.

m) El nivel de lenguaje de la computadora más conveniente para el programador para rápida y fácilmente escribir programas es _____.

n) El lenguaje orientado a los negocios más común y de amplio uso hoy en día es _____.

o) El único lenguaje que una computadora puede entender directamente se llama el lenguaje _____ de la computadora.

p) ¿Qué unidad lógica de la computadora coordina las actividades de todas las demás unidades lógicas? _____.

1.7. Diga si cada uno de los siguientes es verdadero o falso. Explique sus respuestas.

a) Los lenguajes máquina son en general dependientes de la máquina.

b) El tiempo compartido en realidad hace operar a varios usuarios a la vez en una computadora.

c) Al igual que otros lenguajes de alto nivel, C se considera en general como independiente de la máquina.

1.8. Analice el significado de cada uno de los nombres siguientes en el entorno UNIX:

a) `stdin`

b) `stdout`

c) `stderr`

1.9. ¿Qué capacidad clave se proporciona en C concurrente que no está disponible en ANSI C?.

1.10. ¿Por qué se le da tanta atención a la programación orientada a objetos hoy día en general y a C++ en particular?

Lectura recomendada

(An 90) ANSI, *American National Standard for Information Systems Programming Language C (ANSI Document ANSI/ISO 9899: 1990)*, New York, NY: American National Standards Institute, 1990.

Este es el documento de definición para ANSI C. El documento está disponible para su venta de la American National Standards Institute, 1430 Broadway, New York, New York 10018.

(De90) Deitel, H. M., *Operating Systems* (segunda edición), Reading. MA: Addison-Wesley Publishing Company, 1990.

Un libro de texto para el curso tradicional de ciencia de la computación en sistemas operativos. Los capítulos 4 y 5 presentan un análisis extenso de la programación concurrente.

(Ge89) Gehani, N., y W. D. Roome, *The Concurrent C Programming Language*, Summit, NJ:Silicon Press, 1989.

Es el libro definitorio correspondiente a C concurrente —un superconjunto del lenguaje C que permite a los programadores ejecutar ejecución en paralelo de muchas actividades. También incluye un resumen de C++ concurrente.

(Ja89) Jaeschke, R., *Portability and the C Language*, Indianapolis, IN: Hayden Books, 1989.

Este libro analiza la escritura de programas portátiles en C. Jaeschke sirvió tanto en los comités de estándares ANSI como en el ISO C.

(Ke88) Kernighan, B. W., y D. M Ritchie, *The C Programming Language* (segunda edición), Englewood Cliffs, NJ: Prentice Hall, 1988.

Este libro es el clásico en su campo. El libro se utiliza de forma extensa en cursos de C y en seminarios para programadores establecidos, e incluye un manual de consulta excelente. Ritchie es el autor del lenguaje C y uno de los codiseñadores del sistema operativo UNIX.

(Pl92) Plauger, P. J., *The Standard C Library*, Englewood Cliffs, NJ: Prentice Hall, 1992.

Define y demuestra la utilización de las funciones de la biblioteca estándar C. Plauger sirvió como director del subcomité de bibliotecas del comité que desarrolló el estándar ANSI C, y ahora sirve como Convenor del comité ISO del cual resultó C.

(Ra90) Rabinowitz, H., y C. Schaap, *Portable C*, Englewood Cliffs, NJ: Prentice Hall, 1990.

Este libro se desarrolló para un curso sobre portabilidad que se dio en los Laboratorios Bell de AT&T. Rabinowitz está con el laboratorio de inteligencia artificial de la corporación NYNEX, y Schaap es un funcionario importante en la corporación Delft Consulting.

(Ri78) Ritchie, D. M.; S. C. Johnson; M. E. Lesk; y B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language", *The Bell System Technical Journal*, Vol. 57, No. 6 Part 2, julio-agosto 1978, pp. 1991-2019.

Este es uno de los artículos clásicos de introducción al lenguaje C. Apareció en una emisión especial del *Bell System Technical Journal* dedicado al "sistema de tiempo compartido UNIX"

(Ri84) Ritchie, D. M., "The UNIX System: The Evolution of the UNIX Time-Sharing System", *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Part 2, octubre de 1984, pp. 1577-1593.

Un artículo clásico sobre el sistema operativo UNIX. Este artículo apareció en una edición especial del *Bell System Technical Journal* totalmente dedicado "al sistema UNIX"

(Ro84) Rosler, L., "The UNIX System: The Evolution of C —Past and Future", *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Parte 2, octubre de 1984, pp. 1685-1699.

Un excelente artículo para seguir al (Ri78) para aquel lector interesado en rastrear la historia de C y las raíces del esfuerzo de normalización de ANSI C. Apareció en una edición especial del *Bell System Technical Journal* "dedicado al sistema UNIX"

(Si84) Stroustrup, B., "The UNIX System: Data Abstraction in C", *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8 Parte 2, octubre 1984, pp. 1701-1732.

Este artículo clásico de introducción a C++. Apareció en una edición especial del *Bell System Technical Journal* dedicado al "sistema UNIX".

(St91) Stroustrup, B. *The C++ Programming Language* (segunda edición), Reading, MA: Addison-Wesley Series in Computer Science, 1991.

Este libro es la referencia definitoria de C++, un superconjunto de C que incluye varias mejoras a C, especialmente características para la programación orientada a objetos. Stroustrup desarrolló C++ en los Laboratorios Bell de AT&T.

(To89) Tondo, C. L., y S. E. Gimpel, *The C Answer Book*, Englewood Cliffs. NJ: Prentice Hall, 1989.

Este libro único proporciona las respuestas a los ejercicios de Kernighan y Ritchie (Ke88). Los autores demuestran un estilo de programación ejemplar, y proporcionan pensamientos en sus métodos de resolución de problemas y en su decisiones de diseño. Tondo trabaja en IBM y en la Universidad de Nova en Ft. Lauderdale, Florida. Gimpel es un asesor.

2

Introducción a la programación en C

Objetivos

- Ser capaz de escribir programas simples de computación en C.
- Ser capaz de utilizar enunciados simples de entrada y de salida.
- Familiarizarse con los tipos fundamentales de datos.
- Comprender los conceptos de la memoria de la computadora.
- Ser capaz de utilizar operadores aritméticos.
- Comprender la precedencia de los operadores aritméticos.
- Ser capaz de escribir enunciados simples de toma de decisiones.

¿Qué hay en tu nombre? ¡Lo que llamamos rosa exhalaría el mismo grato perfume con cualquiera otra denominación!
William Shakespeare
Romeo y Julieta

Seguí el curso normal ... el que marcan las diferentes ramas de la aritmética la ambición, la confusión, la fealdad y la mofa.
Lewis Carroll

Los precedentes que establecen en forma deliberada los sabios deben ponderarse con detenimiento.
Henry Clay

Sinopsis

Resumen • Terminología • Errores comunes de programación • Prácticas sanas de programación • Sugerencia de portabilidad • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios.

2.1 Introducción

El lenguaje C facilita un método estructurado y disciplinado para el diseño de programas de computación. En este capítulo presentamos la programación en C, dando varios ejemplos que ilustran muchas características importantes de C. Cada ejemplo se analiza de forma cuidadosa, enunciado por enunciado. En los capítulos 3 y 4 presentaremos una introducción a la *programación estructurada* en C. A partir de ahí el método estructurado será utilizado a todo lo largo del resto del texto.

2.2 Un programa simple en C: Cómo imprimir una línea de texto

C utiliza algunas notaciones que pudieran parecer raras a personas que no han programado computadoras. Empezamos analizando un programa simple en C. Nuestro primer ejemplo imprime una línea de texto. El programa y la correspondiente salida en pantalla del programa, se muestran en la figura 2.1.

Aun cuando este programa es simple, ilustra varias características importantes del lenguaje C. Ahora estudiemos en detalle cada línea del programa.

```
/* A first program in C */

main()
{
   printf("Welcome to C!\n");
}
```

```
Welcome to C!
```

Fig. 2.1 Programa para imprimir texto.


```
/* A first program in C */
```

empieza con `/*` y termina con `*/`, indicando que esta línea es un *comentario*. Los programadores insertan comentarios para *documentar* los programas y mejorar la legibilidad de los mismos. Al ejecutarse el programa, los comentarios no hacen que la computadora realice ninguna acción. Los comentarios serán ignorados por el compilador de C y no harán que se genere ningún código objeto en lenguaje máquina. El comentario `A first program in C` simplemente describe el objetivo del programa. Los comentarios también ayudan a otras personas a leer y comprender su programa, pero demasiados comentarios podrían hacer que un programa sea difícil de leer.

Error común de programación 2.1

*Olvidar terminar un comentario con */.*

Error común de programación 2.2

*Iniciar un comentario con los caracteres */ o terminar un comentario con los caracteres /*.*

La línea

```
main()
```

forma parte de todo programa de C. Los paréntesis después de `main` indican que `main` es un bloque constructivo del programa conocido como una *función*. Los programas en C contienen una o más funciones, una de las cuales deberá de ser `main`. Todos los programas en C empiezan a ejecutarse en la función `main`.

Práctica sana de programación 2.1

Todas las funciones deberán ser precedidas por un comentario que describa el objeto de la función.

La *llave izquierda* `{`, debe de iniciar el *cuerpo* de cada función. Una *llave derecha* correspondiente debe dar por terminada cada función. Este par de llaves, y la porción de programa existente entre ambas, también se conoce como un *bloque*. El bloque es una importante unidad de programa en C.

La línea

```
printf("Welcome to C!\n");
```

instruye a la computadora para que ejecute una *acción*, es decir que imprima en la pantalla la *cadena* de caracteres descritas por las comillas. Una cadena a veces se conoce como una *cadena de caracteres*, un *mensaje* o una *literal*. Toda la línea, incluyendo a `printf`, sus *argumentos* dentro de los paréntesis, y el *punto y coma* (;), se llama un *enunciado*. Todo enunciado debe terminar con un punto y coma (también conocido como *terminador de enunciado*). Cuando se ejecuta el enunciado anterior `printf`, imprime en pantalla el mensaje `Welcome to C!`. Los caracteres por lo regular se imprimirán exactamente como aparecen entre las dobles comillas del enunciado `printf`. Advierta que los caracteres `\n` no aparecieron impresos en pantalla. La diagonal invertida (\) se llama un *carácter de escape*. Indica que `printf` se supone debe ejecutar algo extraordinario. Cuando se encuentra con una diagonal invertida, `printf` mira hacia adelante, lee el siguiente carácter y lo combina con la diagonal invertida para formar una *secuencia de escape*. La secuencia de escape `\n` significa *nueva línea*, y hace que en pantalla el cursor se coloque al principio de la siguiente línea. Otras secuencias de escape comunes se listan en la figura 2.2. La función `printf` es una de las muchas funciones incluidas en la *Biblioteca estándar de C* (enlistada en el Apéndice B).

Secuencia de escape	Descripción
`\n`	Nueva línea. Coloca el cursor al principio de la siguiente línea.
`\t`	Tabulador horizontal. Mueve el cursor al siguiente tabulador.
`\r`	Retorno de carro. Coloca el cursor al principio de la línea actual; no avanza a la línea siguiente.
`\a`	Alerta. Hace sonar la campana del sistema.
`\\`	Diagonal invertida. Imprime un carácter de diagonal invertida en un enunciado `printf`.
`\"`	Doble comilla. Imprime un carácter de doble comilla en un enunciado `printf`.

Fig. 2.2 Algunas secuencias de escape comunes.

Las dos últimas secuencias de escape de la figura 2.2 pudieran parecer raras. Dada que la diagonal invertida tiene una significación especial para **printf**, es decir, la reconoce como un carácter de escape en vez de un carácter para su impresión, utilizamos una doble diagonal invertida (\\) para indicar que una sola diagonal invertida debe de ser impresa. La impresión de una doble comilla también presenta un problema para **printf**, porque por lo regular supone que una comilla doble marca el límite de una cadena, y que una doble comilla por sí misma, de hecho no debe ser impresa. Al utilizar la secuencia de escape \" le informamos a **printf** que imprima una doble comilla.

La *llave derecha*, }, indica que se ha llegado al final de **main**.

Error común de programación 2.3

*Escribir en un programa el nombre de la función de salida **printf** como solo **print**.*

Dijimos que **printf** hace que la computadora ejecute una *acción*. Conforme cualquier programa se ejecuta, lleva a cabo una variedad de acciones y el programa toma *decisiones*. Al final de este capítulo, analizaremos la toma de decisiones. En el capítulo 3, explicaremos con mayor detalle este *modelo de acción/decisión de la programación*.

Es importante advertir que las funciones estándar de biblioteca, como **printf** y **scanf**, no forman parte del lenguaje de programación C. Por lo tanto, por ejemplo, el compilador no podrá encontrar un error de ortografía en **printf** o en **scanf**. Cuando el compilador compila un enunciado **printf**, sólo deja espacio libre en el programa objeto para una "llamada" a la función de biblioteca. Pero el compilador no sabe dónde están las funciones de biblioteca. Quien lo sabe es el enlazador. Por lo tanto, cuando el enlazador se ejecuta, localiza las funciones de biblioteca e inserta las llamadas apropiadas a esas funciones de biblioteca, dentro del programa objeto. Entonces queda el programa objeto "completo" y listo para su ejecución. De hecho, un programa enlazado a menudo se llama un *ejecutable*. Si el nombre de la función está mal escrito, será el enlazador quien encuentre el error, porque no será capaz de hacer coincidir el nombre existente en el programa C con el nombre de cualquier función conocida existente en las bibliotecas.

Práctica sana de programación 2.2

*El último carácter impreso por una función que haga cualquier impresión, debería ser una nueva línea (**\n**). Esto asegura que la función dejará el cursor de pantalla colocado al principio de una nueva línea.*

Prácticas de esta naturaleza fomentan la reutilización del software —una meta clave en los entornos de desarrollo de software.

Práctica sana de programación 2.3

Haga un nivel de sangría (tres espacios) en todo el cuerpo de cada función dentro de las llaves que definen el cuerpo de la función. Esto enfatiza la estructura funcional de los programas y ayuda a hacer que los programas sean más legibles.

Práctica sana de programación 2.4

Defina una regla convencional para el tamaño de la sangría que prefiera y a continuación aplíquela de forma uniforme. La tecla del tabulador puede ser utilizada para crear sangrías, pero los tabuladores pudieran variar. Recomendamos utilizar ya sea tabuladores de 1/4 de pulgada, o contar a mano tres espacios por cada uno de los niveles de sangría.

La función **printf** puede imprimir **Welcome to C!** de varias formas diferentes. Por ejemplo, el programa de la figura 2.3 produce la misma salida o resultado que el programa de la figura 2.1. Esto es así porque **printf** continúa imprimiendo donde se detuvo el anterior **printf** en su impresión. El primer **printf** imprime **Welcome** seguido por un espacio, y el segundo **printf** empieza a imprimir de inmediato, a continuación del espacio.

Un solo **printf** puede imprimir varias líneas, utilizando caracteres de nueva línea como se ve en la figura 2.4. Cada vez que se encuentra con la secuencia de escape **\n** (nueva línea), **printf** se coloca al principio de la siguiente línea.

```
/* Printing on one line with two printf statements */

main()
{
   printf("Welcome ");
   printf("to C!\n");
}
```

```
Welcome to C!
```

Fig. 2.3 Cómo imprimir en una línea utilizando enunciados separados **printf**.

```
/* Printing multiple lines with a single printf */

main()
{
   printf("Welcome\nto\nC!\n");
}
```

```
Welcome
to
C!
```

Fig. 2.4 Cómo imprimir en líneas múltiples con un solo **printf**.

2.3 Otro programa simple en C: cómo sumar dos enteros

Nuestro siguiente programa utiliza la función **scanf** de la biblioteca estándar para obtener dos enteros escritos sobre el teclado por el usuario, calcular la suma de estos valores, e imprimir el resultado utilizando **printf**. Tanto el programa como la salida de muestra aparecen en la figura 2.5.

El comentario **/* Addition program */** declara el objetivo del programa. La línea

```
# include <stdio.h>
```

es una directriz del *preprocesador de C*. Las líneas que se inician con el signo de # son procesadas por el preprocesador, antes de la compilación del programa. Esta línea de forma específica le indica al preprocesador que incluya dentro del programa el contenido del archivo de *cabecera de entrada/salida estándar* (**stdio.h**). Este archivo de cabecera contiene información y declaraciones utilizadas por el compilador al compilar funciones estándar de biblioteca de entrada y salida como son **printf**. El archivo de cabecera también contiene información que ayuda al compilador a determinar si las llamadas a las funciones de biblioteca han sido escritas de manera correcta. En el capítulo 5 explicaremos con mayor detalle el contenido de los archivos de cabecera.

Práctica sana de programación 2.5

Aunque la inclusión de **<stdio.h>** *es opcional, deberá ser incluida en cualquier programa C que utilice funciones de entrada/salida estándar de biblioteca. Esto ayuda al compilador a auxiliarle a usted a localizar errores en la fase de compilación de su programa, más bien que en la fase de ejecución (donde los errores resultan más costosos de corregir).*

```
/* Addition program */
#include <stdio.h>

main()
{
   int integer1, integer2, sum;      /* declaration */

   printf("Enter first integer\n");  /* prompt */
   scanf("%d", &integer1);           /* read an integer */
   printf("Enter second integer\n"); /* prompt */
   scanf("%d", &integer2);           /* read an integer */
   sum = integer1 + integer2;        /* assignment of sum */
   printf("Sum is %d\n", sum);       /* print sum */

   return 0;  /* indicate that program ended successfully */
}
```

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

Fig. 2.5 Un programa de suma.

Como se ha indicado, todos los programas inician su ejecución con **main**. La llave izquierda { marca el principio del cuerpo de **main** y la llave derecha correspondiente marca el final de **main**. La línea

```
int integer1, integer2, sum;
```

es una *declaración*. Las letras **integer1**, **integer2**, y **sum** son los nombres de *variables*. Una variable es una posición en memoria donde se puede almacenar un valor para uso de un programa. Esta declaración especifica que las variables **integer1**, **integer2**, y **sum** son del tipo ***int***, lo que significa que estas variables contendrán valores *enteros*, es decir, valores tales como 7, -11, 0, 31914, y similares. Todas las variables deben de declararse con un nombre y un tipo de datos, de inmediato después de la llave izquierda que inicia el cuerpo de **main**, antes de que puedan ser utilizadas en un programa. En C existen otros tipos de datos, además de **int**. En una declaración se pueden declarar varias variables del mismo tipo. Podíamos haber escrito tres declaraciones, una para cada variable, pero la declaración anterior es más concisa.

Práctica sana de programación 2.6

Coloque un espacio después de cada coma (,) para hacer los programas más legibles.

Un nombre de variable en C es cualquier *identificador* válido. Un identificador es una serie de caracteres formados de letras, dígitos y subrayados (_) que no se inicien con un dígito. Un identificador puede tener cualquier longitud, pero según con el estándar ANSI C sólo se requieren los primeros 31 caracteres para su reconocimiento por los compiladores de C. C es *sensible* —lo que quiere decir que para C las letras mayúsculas y minúsculas son diferentes, por lo que **a1** y **A1** son identificadores distintos.

Error común de programación 2.4

Usar una letra mayúscula donde debería haberse usado una minúscula (por ejemplo al escribir ***Main*** *en vez de* ***main****).*

Sugerencia de portabilidad 2.1

Utilice identificadores de 31 o menos caracteres. Esto auxilia a asegurar la portabilidad y puede evitar algunos errores sutiles de programación.

Práctica sana de programación 2.7

La selección de nombres de variables significativos ayuda a la autodocumentación de un programa, es decir, se requerirán de menos comentarios.

Práctica sana de programación 2.8

La primera letra de un identificador utilizado como un nombre simple de variable, deberá ser una letra minúscula. Más adelante, en el texto, le daremos significación especial a aquellos identificadores que empiezan con una letra mayúscula y aquellos que utilizan todas mayúsculas.

Práctica sana de programación 2.9

Los nombres de variables de varias palabras pueden auxiliar a la legibilidad de un programa. Evite juntar palabras separadas, como en `totalcommissions`*. En vez de ello, separe las palabras con subrayados como* `total_commissions`*, o bien, si desea reunirlas, empiece cada palabra después de la primera con una letra mayúscula, como en* `totalCommissions`*.*

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERIA
DEPARTAMENTO DE

Las declaraciones deben de ser colocadas después de la llave izquierda de una función y antes de *cualquier* enunciado ejecutable. Por ejemplo, en el programa de la figura 2.5, la inserción de la declaración después de la primera **printf** generaría un error de sintaxis. Se causa un *error de sintaxis* cuando el compilador no puede reconocer un enunciado. El compilador por lo general emite un mensaje de error para auxiliar al programador en la localización y corrección del enunciado incorrecto. Los errores de sintaxis son violaciones al lenguaje. Los errores de sintaxis también se conocen como *errores de compilación o errores en tiempo de compilación.*

Error común de programación 2.5

Colocar declaraciones de variables entre enunciados ejecutables.

Práctica sana de programación 2.10

Separe las declaraciones y los enunciados ejecutables en una función mediante una línea en blanco, para enfatizar dónde terminan las declaraciones y dónde empiezan los enunciados ejecutables.

El enunciado

```
printf ("Enter first integer\n");
```

imprime la literal **Enter first integer** en la pantalla y se coloca al principio de la línea siguiente. Este mensaje se llama un porque le dice al usuario que debe de tomar una acción específica.

El enunciado

```
scanf ("%d", &integer1);
```

utiliza ***scanf*** para obtener un valor del usuario. La función **scanf** toma la entrada de la entrada estándar, que normalmente es el teclado. Este **scanf** tiene dos argumentos, **"%d"** y **&integer1**. El primer argumento, la *cadena de control de formato*, indica el tipo de dato que deberá ser escrito por el usuario. El *especificador de conversión* **%d** indica que los datos deberán de ser un entero (la letra **d** significa "entero decimal"). El signo de % en este contexto es tratado por **scanf** (y como veremos por **printf**) como un carácter de escape (como \)y la combinación **%d** es una secuencia de escape (como sería **\n**). El segundo argumento de **scanf** empieza con un ampersand (**&**) —llamado el *operador de dirección* en C— seguido por un nombre de variable. El ampersand, al ser combinado con el nombre de variable, le indica a **scanf** la posición en memoria en la cual está almacenada la variable **integer1**. La computadora a continuación almacena el valor correspondiente a **integer1** en dicha posición. El uso de ampersand (**&**) resulta con frecuencia confuso para los programadores neófitos o para personas que hayan programado en otros lenguajes que no requieren de esta notación. Por ahora, sólo recuerde de anteceder cada variable en todos los enunciados **scanf** con un ampersand. Algunas excepciones a esta regla, se analizan en los capítulos 6 y 7. El verdadero significado del uso del ampersand será aclarado una vez que estudiemos los apuntadores en el capítulo 7.

Cuando la computadora ejecute el **scanf** anterior, esperará a que el usuario escriba un valor para la variable **integer1**. El usuario responderá escribiendo un entero y presionando la *tecla de retorno* (a veces conocida como *tecla de entrar*) para enviar el número a la computadora. La computadora entonces asigna este número, o *valor* a la variable **integer1**. Cualquier referencia subsecuente a **integer1** dentro del programa utilizará este mismo valor. Las funciones **printf** y **scanf** facilitan la interacción entre el usuario y la computadora. Dado que esta interacción se

asemeja a un diálogo, a menudo se conoce como *computación conversacional* o *computación interactiva.*

El enunciado

```
printf ("Enter second integer\n");
```

imprime el mensaje **Enter second integer** en la pantalla, y a continuación posiciona el cursor al principio de la siguiente línea. También este **printf** indica al usuario que tome acción.

El enunciado

```
scanf ("%d", &integer2);
```

obtiene un valor para la variable **integer2.** El *enunciado de asignación*

```
sum = integer1 + integer2;
```

calcula la suma de las variables **integer1** e **integer2** y asigna el resultado a la variable **sum** utilizando el *operador de asignación* **=**. El enunciado se lee como, "**sum** *obtiene* el valor de **integer1 + integer2**". La mayor parte de los cálculos se ejecutan en enunciados de asignación. El operador = y el operador + se llaman *operadores binarios,* porque cada uno de ellos tiene *dos operandos.* En el caso del operador **+**, los dos operandos son **integer1** e **integer2**. En el caso del operador **=**, los dos operandos son **sum** y el valor de la expresión **integer1 + integer2**.

Práctica sana de programación 2.11

Coloque espacios en blanco a ambos lados de un operador binario. Esto hace resaltar al operador y hace que el programa sea mas legible.

Error común de programación 2.6

El cálculo en un enunciado de asignación debe de aparecer en el lado derecho del operador =. Es un error de sintaxis colocar un cálculo del lado izquierdo de un operador de asignación.

El enunciado

```
printf("Sum is %d\n", sum);
```

utiliza la función **printf** para imprimir en pantalla la literal **Sum is** seguida del valor numérico de la variable **sum**. Esta **printf** tiene dos argumentos, **"Sum is %d\n"** y **sum**. El primer argumento es la cadena de control de formato. Contiene algunos caracteres literales que se desplegarán, así como el especificador de conversión **%d**, que indica que deberá imprimirse un entero. El segundo argumento especifica el valor a ser impreso. Advierta que el especificador de conversión para un entero es idéntico tanto en **printf** como en **scanf**. Esto es cierto para la mayor parte de los tipos de datos en C.

Los cálculos también pueden llevarse a cabo dentro de enunciados **printf**. Podríamos haber combinado los dos enunciados previos en el siguiente enunciado

```
printf("Sum is %d\n", integer1 + integer2);
```

El enunciado

```
return 0;
```

pasa el valor 0 de regreso al entorno del sistema operativo en el cual se está ejecutando el programa. Esto le indica al sistema operativo que el programa fue ejecutado con éxito. Para más información sobre cómo informar de algún tipo de falla del programa, refiérase a los manuales de su entorno de sistema operativo particular.

La llave derecha, }, indica que se ha llegado al final de la función **main**.

Error común de programación 2.7

Olvidarse uno o ambos de las dobles comillas que rodean a la cadena de control de formato en un `printf` *o un* `scanf`.

Error común de programación 2.8

Olvidar el signo de `%` *en una especificación de conversión en la cadena de control de formato de un* `printf` *o* `scanf`.

Error común de programación 2.9

Colocar una secuencia de escape como `\n` *fuera de la cadena de control de formato de un* `printf` *o de un* `scanf`.

Error común de programación 2.10

Olvidar incluir las expresiones cuyos valores deben de ser impresas en una `printf` *que contiene especificadores de conversión.*

Error común de programación 2.11

No proporcionar en una cadena de control de formato `printf` *un especificador de conversión, cuando se requiere de uno para imprimir una expresión.*

Error común de programación 2.12

Colocar dentro de la cadena de control de formato la coma, que se supone debe separar la cadena de control de formato de la expresiones a imprimirse.

Error común de programación 2.13

Olvidar anteceder una variable en un enunciado `scanf` *con un ampersand, cuando dicha variable debería, de hecho, estar precedida por un ampersand.*

En muchos sistemas, este error de tiempo de ejecución se conoce como una "falla de segmentación" o bien una "violación de acceso". Este tipo de error ocurre cuando el programa de un usuario intenta tener acceso a una parte de la memoria de la computadora, en la cual el programa del usuario no tiene privilegios de acceso. La causa precisa de este error será explicada en el capítulo 7.

Error común de programación 2.14

Anteceder una variable incluida en un enunciado `printf` *con un ampersand, cuando de hecho esa variable no debería ser precedida por un ampersand.*

En el capítulo 7, estudiaremos apuntadores y veremos casos en los cuales desearemos anteceder un nombre de variable con un ampersand para imprimir la dirección de dicha variable. Durante los siguientes varios capítulos, sin embargo, los enunciados **printf** no deberán de incluir estos ampersand.

2.4 Conceptos de memoria

Los nombres de variables como **integer1**, **integer2** y **sum** de hecho corresponden a *localizaciones* o *posiciones* en la memoria de la computadora. Cada variable tiene un *nombre*, un *tipo* y un *valor*.

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

```
scanf ("%d", &integer1);
```

el valor escrito por el usuario se coloca en una posición de memoria a la cual se ha asignado el nombre **integer1**. Suponga que el usuario escribe el número **45** como el valor para **integer1**. La computadora colocará **45** en la posición **integer1**, como se muestra en la figura 2.6.

Siempre que se coloca un valor en una posición de memoria, este valor sustituye el valor anterior existente en dicha posición. Dado que la información previa resulta destruida, el proceso de leer información a la memoria se llama *lectura destructiva*.

Regresando de nuevo a nuestro programa de suma, cuando el enunciado

```
scanf ("%d", &integer2);
```

se ejecuta, suponga que el usuario escribe el valor **72**. Este valor se coloca en la posición **integer2**, y la memoria aparece como se muestra en la figura 2.7. Advierta que en memoria estas posiciones no necesariamente serán adyacentes.

Una vez que el programa ha obtenido los valores de **integer1** e **integer2**, suma estos valores y coloca la suma en la variable **sum**. El enunciado

```
sum = integer1 + integer2;
```

Fig. 2.6 Una posición de memoria mostrando el nombre y valor de una variable.

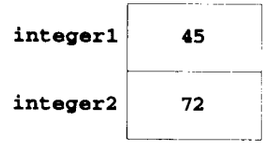


Fig. 2.7 Posiciones de memoria una vez que se han introducido ambas variables.

que ejecuta la suma también involucra lectura destructiva. Esto ocurre cuando la suma calculada de **integer1** y de **integer2** se coloca en la posición **sum** (destruyendo el valor que pudiera haber existido ya en **sum**). Una vez que **sum** esté calculada, la memoria aparece como en la figura 2.8. Note que los valores de **integer1** e **integer2** aparecen de forma exacta como estaban antes de haber sido utilizados en el cálculo de **sum**. Estos valores fueron utilizados pero no destruidos, al ejecutar el cálculo a la computadora. Entonces, cuando un valor se lee de la posición de memoria, el proceso se conoce como *lectura no destructiva.*

2.5 Aritmética en C

La mayor parte de los programas C ejecutan cálculos aritméticos. Los *operadores aritméticos de C* se resumen en la figura 2.9. Advierta el uso de varios símbolos especiales, no utilizados en álgebra. El *asterisco* (*) indica multiplicación y el *signo de por ciento* (%) denota el operador *módulo,* que se presenta más adelante. En álgebra, si deseamos multiplicar *a* por *b* simplemente colocamos estos nombres de variable de una sola letra uno al lado del otro, como en *ab.* En C, sin embargo, si hiciéramos esto, **ab** se interpretaría como un nombre solo de dos letras (o un identificador). Por lo tanto, C (y en general otros lenguajes de programación) requiere que se denote la multiplicación en forma explícita, utilizando el operador *****, como en **a*b**.

Los operadores aritméticos son todos operadores binarios. Por ejemplo, la expresión **3 + 7** contiene el operador binario + y los operandos **3** y **7**.

integer1	**45**
integer2	**72**
sum	**117**

Fig. 2.8 Localizaciones de memoria después de un cálculo.

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

Fig. 2.9 Operadores aritméticos de C.

La *división de enteros* da como resultado también un entero. Por ejemplo, la expresión **7/4** da como resultado **1**, y la expresión **17/5** da como resultado **3**. C tiene el operador de módulo, %, que proporciona el residuo después de una división de enteros. El operador de módulo es un operador entero, que puede ser utilizado sólo con operandos enteros. La expresión **x%y** resulta o entrega el residuo, después de que **x** haya sido dividido por **y**. Por lo tanto, **7%4** da como resultado **3**, y **17%5** da como resultado **2**. Analizarémos muchas aplicaciones interesantes del operador de módulo.

Error común de programación 2.15

Un intento de dividir entre cero, por lo regular resulta no definido en sistemas de cómputo y en general da como resultado un error fatal, es decir, un error que hace que el programa se termine de inmediato sin haber ejecutado con éxito su tarea. Los errores no fatales permiten que los programas se ejecuten hasta su término, produciendo a menudo resultados incorrectos.

Las expresiones aritméticas en C deben de ser escritas en una *línea continua,* para facilitar la escritura de programas en la computadora. Entonces, expresiones tales como "**a** dividido por **b**" debe de estar escrito como **a/b,** de tal forma que todos los operadores y operandos aparezcan en una sola línea. La notación algebraica

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

en general no es aceptable para compiladores, aunque si existen algunos paquetes de software de uso especial que aceptan notaciones más naturales, para expresiones matemáticas complejas.

Los paréntesis se utilizan en las expresiones de C de manera muy similar a como se usan en las expresiones algebraicas. Por ejemplo, para multiplicar **a** por la cantidad **b+c** escribimos:

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

C calculará las expresiones aritméticas en una secuencia precisa, determinada por las *reglas de precedencia de operadores* que siguen y, que en general son las mismas que las que se siguen en álgebra.

1. Primero se calculan expresiones o porciones de expresiones contenidas dentro de pares de paréntesis. Entonces, *los paréntesis pueden ser utilizados para obligar a un orden de evaluación en cualquier secuencia deseada por el evaluador. Los paréntesis* se dicen que están en el "más alto nivel de precedencia". En el caso de paréntesis *anidados* o *incrustados,* se evalúa primero la expresión en el par de paréntesis más interno.
2. A continuación, se calculan las operaciones de multiplicación, división y módulo. Si una expresión contiene varias multiplicaciones, divisiones y módulos, la evaluación avanzará de izquierda a derecha. Se dice que la multiplicación, división y módulo tienen el mismo nivel de precedencia.
3. Por último, se calculan las operaciones de suma y de resta. Si una expresión contiene varias operaciones de suma y de resta, la evaluación avanzará de izquierda a derecha. La suma y la resta también tienen el mismo nivel de precedencia.

Las reglas de precedencia de operadores son guías de acción, que le permiten a C calcular expresiones en el orden correcto. Cuando decimos que una evaluación o cálculo avanza de izquierda a derecha, nos estamos refiriendo a la *asociatividad* de los operadores. Veremos que algunos operadores se asocian de derecha a izquierda. En la figura 2.10 se resumen estas reglas de precedencia de operadores.

Operador (es)	Operación (es)	Orden de cálculo (precedencia)
( )	Paréntesis	Se calculan primero. Si los paréntesis están anidados, la expresión en el par más interno se evalúa primero. Si existen varios pares de paréntesis "en el mismo nivel" (es decir no anidadas), se calcularán de izquierda a derecha.
*, /, o bien %	Multiplicación, División y Módulo	Se evalúan en segundo lugar. Si existen varias, se calcularán de izquierda a derecha.
+ o bien -	Suma o Resta	Se calculan al último. Si existen varios, serán evaluados de izquierda a derecha.

Fig. 2.10 Precedencia de operadores aritméticos.

Veamos ahora varias expresiones a la luz de las reglas de precedencia de operadores. Cada ejemplo enlista una expresión algebraica y su equivalente en C.

El ejemplo siguiente calcula el promedio aritmético (la media) de cinco términos:

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

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

Se requieren los paréntesis, porque la división tiene una precedencia más alta que la suma. Toda la cantidad `(a + b + c + d + e)` debe de dividirse entre `5`. Si los paréntesis se omiten por error, `a + b + c + d + e / 5` que se calcula incorrectamente como

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

El siguiente ejemplo es la ecuación de una línea recta:

Algebra: $y = mx + b$

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

No se requieren paréntesis. Primero se evalúa la multiplicación, porque ésta tiene una precedencia más alta que la suma.

El siguiente ejemplo contiene módulo (%), multiplicación, división, adición y substracción:

Algebra: $z = pr\%q + w/x - y$

C:
```
z =   p * r % q + w / x - y;
      1   2   4   3   5
```


Los números en círculos bajo el enunciado indican el orden en el cual valorará o calculará C los operadores. La multiplicación, el módulo y la división, serán evaluadas primero, en un orden de izquierda a derecha (es decir, se asocian de izquierda a derecha), en vista de que tienen precedencia mayor que la suma y la resta. La suma y la resta serán calculadas después. También ellas se evaluarán de izquierda a derecha.

No todas las expresiones con varios pares de paréntesis contienen paréntesis anidados. La expresión

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

no contiene paréntesis anidados. En vez de ello, se dice que estos paréntesis están "en el mismo nivel". En esta situación, se calcularán las expresiones C en paréntesis primero y en un orden de izquierda a derecha.

Para desarrollar una mejor comprensión de las reglas de precedencia de operadores, veamos cómo se calcula un polinomio de segundo grado.

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

Los números en círculos bajo el enunciado indican el orden en el cual ejecuta C las operaciones. En C no existe un operador aritmético para exponenciación, por lo cual hemos representado x^2 como `x * x`. La biblioteca C estándar incluye la función `pow` ("potencia") para ejecutar la exponenciación. Debido a ciertos aspectos sutiles relacionados con los tipos de datos requeridos por `pow`, vamos a posponer la explicación detallada de `pow` hasta el capítulo 4.

Suponga `a = 2, b =3, c=7,` y `x = 5`. En la figura 2.11 se ilustra cómo se calcula el polinomio de segundo grado ya mostrado.

2.6 Toma de decisiones: Operadores de igualdad y relacionales

Los enunciados ejecutables de C, llevan a cabo ya sea *acciones* (como son cálculos o entradas o salidas de datos), o toman *decisiones* (veremos pronto varios ejemplos de éstos). Pudiéramos tomar una decisión en un programa, por ejemplo, para determinar si la calificación de una persona en un examen es mayor que o igual a 60, y si así es, imprimir el mensaje "¡Felicitaciones! usted pasó". Esta sección introduce una versión simple de la *estructura de control* `if` de C, que permite a un programa tomar una decisión basada en la veracidad o falsedad de alguna declaración de hecho, conocida como una *condición*. Si la condición se cumple (si la condición es *cierta o verdadera*) se ejecuta el enunciado existente en el cuerpo de la estructura `if`. Si la condición no se cumple (es decir, si la condición es *falsa*) el enunciado del cuerpo no se ejecuta. Independientemente de que se ejecute o no el enunciado del cuerpo, una vez terminada la estructura `if`, la ejecución sigue adelante con el enunciado, después de la estructura `if`.

Las condiciones en las estructuras `if` se forman utilizando los *operadores de igualdad* y los *operadores relacionales* resumidos en la figura 2.12. Los operadores relacionales tienen un mismo nivel de precedencia y se asocian de izquierda a derecha. Los operadores de igualdad tienen un nivel de precedencia menor que los operadores relacionales y también se asocian de izquierda a derecha. (Nota: en C, una condición puede de hecho ser cualquier expresión que genere un cero (falsa) o un no cero (verdadera). Veremos muchas aplicaciones de lo anterior a lo largo de este libro).

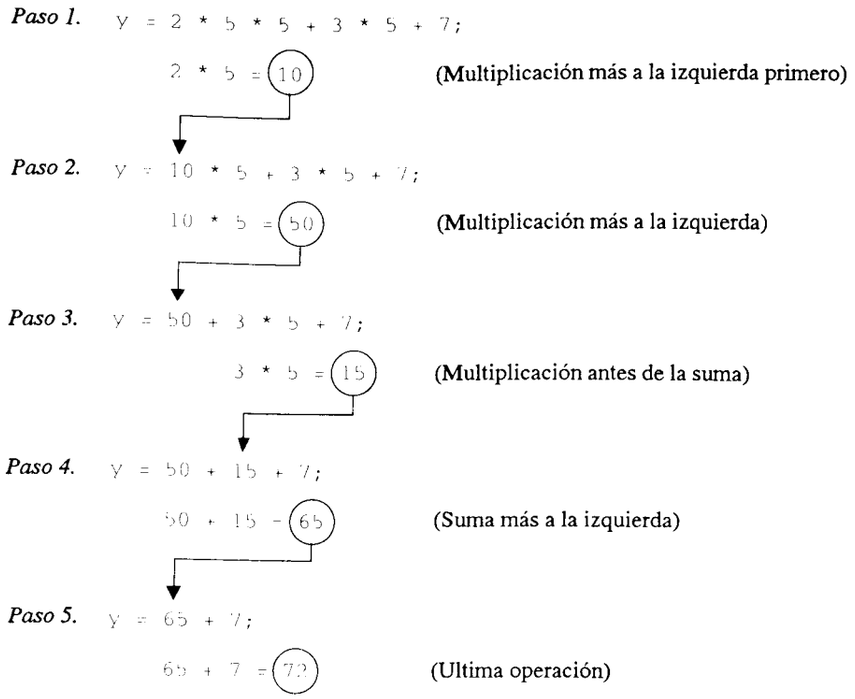


Fig. 2.11 Cálculo y un polinomio de segundo grado.

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

Fig. 2.12 Operadores de igualdad y relacionales.

Error común de programación 2.16

Ocurrirá un error de sintaxis si los dos símbolos en cualquiera de los operadores `==`, `!=`, `>=` y `<=` están separados por espacios.

Error común de programación 2.17

Ocurrirá un error de sintaxis si los dos símbolos en cualquiera de los operadores `!=`, `>=`, y `<=`, se invierten como en `=!`, `=>`, y `=<`, respectivamente.

Error común de programación 2.18

Confundir el operador de igualdad `==` con el operador de asignación `=`.

Para evitar esta confusión, el operador de igualdad deberá ser leído como "doble igual" y el operador de asignación debe de ser leído como "obtiene". Como veremos pronto, confundir estos dos operadores no podría causar por necesidad un error de sintaxis fácil de reconocer, pero pudiera causar errores lógicos muy sutiles.

Error común de programación 2.19

Colocar un punto y coma inmediatamente a la derecha del paréntesis derecho, después de la condición, en una estructura `if`.

El ejemplo de la figura 2.13 utiliza seis enunciados `if` para comparar dos números escritos por el usuario. Si se satisface la condición en cualquiera de estos enunciados `if`, se ejecuta el enunciado `printf` asociado con dicho `if`. El programa así como tres salidas de ejecución de muestra aparecen en la figura.

Advierta que el programa de la figura 2.13 utiliza `scanf` para recibir la entrada de dos números. Cada especificador de conversión tiene un argumento correspondiente, en el cual se almacenará un valor. El primer `%d` convierte un valor a almacenar en la variable `num1` y el segundo `%d` convierte un valor a almacenar en la variable `num2`. El hacer una sangría en el cuerpo de cada enunciado `if` y colocar líneas en blanco por arriba y por abajo de cada enunciado `if` mejora la legibilidad del programa. También, note que cada enunciado `if` de la figura 2.13 tiene en su cuerpo un solo enunciado. En el capítulo 3 mostrarémos cómo especificar enunciados `if` con cuerpos de varios enunciados.

Práctica sana de programación 2.12

Haga una sangría en el o los enunciados del cuerpo de una estructura `if`.

Práctica sana de programación 2.13

Coloque una línea en blanco antes y después de cada estructura de control en un programa para mejor legibilidad.

Práctica sana de programación 2.14

Deberá procurar que no exista más de un enunciado en cada línea de un programa.

Error común de programación 2.20

Colocar comas (cuando no se necesita ninguna) entre los especificadores de conversión en la cadena de control de formato en un enunciado `scanf`.

UNIVERSIDAD DE LA REPUBLICA
FACULTAD DE INGENIERIA
DEPARTAMENTO

```
/* Using if statements, relational
   operators, and equality operators */
#include <stdio.h>

main()
{
   int num1, num2;

   printf("Enter two integers, and I will tell you\n");
   printf("the relationships they satisfy: ");
   scanf("%d%d", &num1, &num2);   /* read two integers */

   if (num1 == num2)
      printf("%d is equal to %d\n", num1, num2);

   if (num1 != num2)
      printf("%d is not equal to %d\n", num1, num2);

   if (num1 < num2)
      printf("%d is less than %d\n", num1, num2);

   if (num1 > num2)
      printf("%d is greater than %d\n", num1, num2);

   if (num1 <= num2)
      printf("%d is less than or equal to %d\n",
             num1, num2);

   if (num1 >= num2)
      printf("%d is greater than or equal to %d\n",
             num1, num2);

   return 0;   /* indicate program ended successfully */
}
```

Fig. 2.13 Cómo utilizar operadores de igualdad y relacionales (parte 1 de 2).

El comentario en la figura 2.13 está dividido sobre dos líneas o renglones. En los programas C, los caracteres de *espacio en blanco* como tabuladores, nueva línea, y espacios, por lo regular son ignorados. Por lo tanto, los enunciados y comentarios pueden ser divididos entre varios renglones. No es correcto, sin embargo, dividir identificadores.

Práctica sana de programación 2.15

Un enunciado largo puede ser dividido en varios renglones. Si un enunciado debe ser dividido en varias líneas, escoja puntos de corte que tengan sentido (como sería después de una coma en una lista separada por comas). Si un enunciado se divide en dos o más renglones, haga una sangría en todas las líneas subsecuentes.

La gráfica de la figura 2.14 muestra la precedencia de los operadores presentados en este capítulo. Los operadores se muestran de arriba hacia abajo en orden decreciente de precedencia. Advierta que el signo igual es también un operador. Todos estos operadores, a excepción del operador de asignación =, se asocian de izquierda a derecha. El operador de asignación (=)se asocia de derecha a izquierda.


```
Enter two integer, and I will tell you
the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

Fig. 2.13 Cómo utilizar operadores de igualdad y relacionales (parte 2 de 2).

Practica sana de programación 2.16

Refiérase a la gráfica de precedencia de operadores al escribir expresiones que contengan muchos operadores. Confirme que los operadores en la expresión se ejecutan en el orden correcto. Si no está totalmente seguro del orden de evaluación en una expresión compleja, utilice paréntesis para obligar a un orden, exactamente como lo haría en expresiones algebraicas. Asegúrese de observar que algunos de los operadores de C, como el operador de asignación (=) se asocian de derecha a izquierda, en vez de izquierda a derecha.

Operadores	Asociatividad
( )	de izquierda a derecha
* / %	de izquierda a derecha
+ -	de izquierda a derecha
< <= > >=	de izquierda a derecha
== !=	de izquierda a derecha
=	de derecha a izquierda

Fig. 2.14 Precedencia y asociatividad de los operadores hasta ahora analizados.

Algunas de las palabras que hemos utilizado en los programas de C de este capítulo en particular **int**, **return** e **if** son palabras reservadas del lenguaje. El conjunto completo de palabras reservadas de C se muestra en la figura 2.15. Estas palabras tienen un significado especial para el compilador de C, por lo que el programador debe tener cuidado de no utilizar estas palabras como identificadores como serían nombres de variables. En este libro, analizaremos todas estas palabras reservadas.

En este capítulo, hemos presentado muchas características importantes del lenguaje de programación de C, incluyendo la impresión de datos en la pantalla, introducir datos del usuario, llevar a cabo cálculos y tomar decisiones. En el siguiente capítulo elaboraremos más sobre estas técnicas, conforme nos iniciamos en la *programación estructurada*. El estudiante se irá familiarizando con las técnicas de sangría. Estudiaremos cómo especificar el orden en el cual se ejecutan los enunciados —esto se llama *flujo de control*.

Resumen

- Los comentarios empiezan con **/*** y terminan con ***/**. Los programadores insertan comentarios para documentar los programas y mejorar su legibilidad. Los comentarios no hacen que la computadora lleve a cabo acción alguna cuando se ejecute el programa.
- La directriz de preprocesador #include **<stdio.h>** le indica al compilador que incluya el archivo de cabecera de entrada/salida estándar dentro del programa. Este archivo contiene información utilizada por el compilador para verificar la precisión de las llamadas de funciones de entrada y de salida, como son **scanf** y **printf**.
- Los programas de C están formados de funciones, una de las cuales debe de ser **main**. Todos los programas de C empiezan su ejecución en la función **main**.
- La función **printf** puede ser utilizada para imprimir una cadena contenida entre comillas, y para imprimir los valores de expresiones. Al imprimir valores enteros, el primer argumento de una función **printf** —la cadena de control de formato— contiene el especificador de conversión **%d** y cualquier otros caracteres que deban de ser impresos; el segundo argumento es la expresión cuyo valor será impresa. Si más de un entero se imprimirá, entonces la cadena de control de formato contendrá un **%d** para cada uno de los enteros, y los argumentos separados con coma, que siguen a la cadena de control de formato, contendrán las expresiones cuyos valores serán o irán a ser impresos.

Palabras clave

auto	**break**	**case**	**char**
const	**continue**	**default**	**do**
double	**else**	**enum**	**extern**
float	**for**	**goto**	**if**
int	**long**	**register**	**return**
short	**signed**	**sizeof**	**static**
struct	**switch**	**typedef**	**union**
unsigned	**void**	**volatile**	**while**

Fig. 2.15 Palabras reservadas de C.

- La función **scanf** obtiene valores que el usuario por lo regular escribe en el teclado. Su primer argumento es la cadena de control de formato, que le indica a la computadora cuál es el tipo de datos que debe ser introducido por el usuario. El especificador de conversión %d indica que los datos deberán de ser un entero. Cada uno de los argumentos restantes corresponde a uno de los especificadores de conversión existentes en la cadena de control de formato. Cada nombre de variable normalmente es precedida por un ampersand (**&**), conocido como el operador de dirección en C. El ampersand, al ser combinado con un nombre de variable, le indica a la computadora la posición en memoria donde se almacenará el valor. La computadora entonces almacena el valor en esa posición.
- Todas las variables en un programa C deben ser declaradas antes de que puedan ser utilizadas por el programa.
- Un nombre de variable en C es cualquier identificador válido. Un identificador es una serie de caracteres consistiendo de letras, dígitos y subrayados (_). Los identificadores no pueden empezar con un dígito. Los identificadores pueden tener cualquier longitud; sin embargo, sólo los primeros 31 caracteres son significativos, de acuerdo con el estándar ANSI.
- C diferencia minúsculas de mayúsculas.
- La mayor parte de los cálculos se ejecutan en enunciados de asignación.
- Todas las variables almacenadas en la memoria de la computadora tienen un nombre, un valor y un tipo.
- Siempre que se coloque un nuevo valor en una posición en memoria, substituye el valor anterior existente en esa misma posición. Dado que esta información anterior queda destruida, el proceso de leer información hacia la memoria se conoce como lectura destructiva.
- El proceso de leer un valor desde la memoria se conoce como lectura no destructiva.
- Las expresiones aritméticas en C deben ser escritas en un solo renglón, para facilitar escribir programas en la computadora.
- C calcula las expresiones aritméticas en una secuencia precisa, definida por las reglas de precedencia de operadores y de asociatividad.
- El enunciado **if** permite al programador tomar una decisión cuando se cumple cierta condición. El formato para un enunciado **if** es

  if (*condición*)
  enunciado

- Si la condición es verdadera, el enunciado en el cuerpo de **if** se ejecuta. Si la condición es falsa, el enunciado del cuerpo será saltado.
- Las condiciones en los enunciados **if**, se forman por lo común mediante el uso de operadores de igualdad y operadores relacionales. El resultado de utilizar estos operadores es siempre la observación de "verdadero" o "falso". Note que las condiciones pueden ser cualquier expresión que genere un valor cero (falso) o un valor no cero (verdadero).

Terminología

acción
modelo de acción/decisión
operador de dirección
ampersand (**&**)
argumento
operadores aritméticos

operador de asignación (=)
enunciado de asignación
asociatividad de operadores
asterisco (*)
carácter de escape diagonal invertida (\)
operadores binarios
bloque
cuerpo de una función
llaves {}
C
diferencia minúsculas de mayúsculas
cadena de caracteres
palabras reservada de C
comentario
error de compilación
error en tiempo de compilación
condición
cadena de control
computación conversacional
especificador de conversión
preprocesador de C
Biblioteca estándar de C
especificador de conversión %d
decisión
toma de decisiones
declaración
lectura destructiva
división entre cero
tecla de entrar
operador de asignación signo igual (=)
operadores de igualdad
  == "igual a"
  != "no es igual a"
carácter de escape
secuencia de escape
falso
error fatal
flujo de control
cadena de control de formato
función
identificador
estructura de control **if**
sangría
int
entero
división de enteros
computación interactiva
palabras clave
asociatividad de izquierda a derecha
literal
posición
main
memoria
posición en memoria
mensaje
operador de módulo (%)
operador de multiplicación (*)
nombre
paréntesis anidados
carácter de nueva línea (\n)
lectura no destructiva
error no fatal
no cero (verdadero)
operando
operador
paréntesis ()
carácter de escape signo de por ciento (%)
precedencia
función **printf**
indicador
operadores relacionales
  > "es mayor que"
  < "es menor que"
  >= "es mayor que o igual a"
  <= "es menor que o igual a"
palabras reservadas
tecla de retorno
asociatividad de derecha a izquierda
reglas de precedencia de operadores
función **scanf**
terminador de enunciado punto y coma (;)
archivo de cabecera de entrada/salida estándar
enunciado
terminador de enunciado (;)
stdio.h
en línea recta
cadena
programación estructurada
error de sintaxis
verdadero
subrayado (_)
valor
variable
nombre de variable
tipo de variable
valor de variable
caracteres de espacio en blanco
cero (falso)

Errores comunes de programación

2.1 Olvidar terminar un comentario con */
2.2 Iniciar un comentario con los caracteres */ o terminar un comentario con los caracteres /*
2.3 Escribir en un programa el nombre de la función de salida **printf** como solo **print.**
2.4 Usar una letra mayúscula donde debería haberse usado una minúscula (por ejemplo al escribir **Main** en vez de **main**).
2.5 Colocar declaraciones de variables entre enunciados ejecutables.
2.6 El cálculo en un enunciado de asignación debe de aparecer en el lado derecho del operador =. Es un error de sintaxis colocar un cálculo del lado izquierdo de un operador de asignación.
2.7 Olvidarse uno o ambas de las dobles comillas que rodean a la cadena de control de formato en un **printf** o un **scanf**.
2.8 Olvidar el signo de % en una especificación de conversión en la cadena de control de formato de un **printf** o **scanf**.
2.9 Colocar una secuencia de escape como \n fuera de la cadena de control de formato de un **printf** o de un **scanf**.
2.10 Olvidar incluir las expresiones cuyos valores deben de ser impresas en un **printf** que contiene especificadores de conversión.
2.11 No proporcionar en una cadena de control de formato **printf** un especificador de conversión cuando se requiere de uno para imprimir una expresión.
2.12 Colocar dentro de la cadena de control de formato la coma que se supone debe separar la cadena de control de formato de las expresiones a imprimirse.
2.13 Olvidar de anteceder una variable en un enunciado **scanf** con un ampersand cuando dicha variable debería, de hecho, estar precedida por un ampersand.
2.14 Anteceder una variable incluida en un enunciado **printf** con un ampersand, cuando, de hecho esa variable no debería ser precedida por un ampersand.
2.15 Un intento de dividir entre cero, por lo regular resulta no definido en sistemas de cómputo y en general da como resultado fatal, es decir, un error que hace que el programa se termine de inmediato sin haber ejecutado de forma exitosa su tarea. Los errores no fatales permiten que los programas se ejecuten hasta su término, produciendo a menudo resultados incorrectos.
2.16 Ocurrirá un error de sintaxis si los dos símbolos en cualquiera de los operadores ==, !=, >= y <= están separados por espacios.
2.17 Ocurrirá un error de sintaxis si los dos símbolos en cualquiera de los operadores, !=, >= y <= se invierten como en =!, =>, y =<, respectivamente.
2.18 Confundir el operador de igualdad == con el operador de asignación =.
2.19 Colocar un punto y coma de inmediato a la derecha del paréntesis derecho después de la condición en una estructura **if**.
2.20 Colocar comas (cuando no se necesita ninguna) entre los especificadores de conversión en la cadena de control de formato en un enunciado **scanf**.

Prácticas sanas de programación

2.1 Todas las funciones deberán ser precedidas por un comentario que describa el objeto de la función.
2.2 El último carácter impreso por una función que hace cualquier impresión, debería ser una nueva línea (\n). Esto asegura que la función dejará el cursor de pantalla colocado al principio de una nueva línea. Acuerdos de esta naturaleza estimulan la reutilización del software —una meta clave en los entornos de desarrollo de software.
2.3 Haga una sangría de todo el cuerpo de cada función en un nivel (tres espacios) dentro de las llaves que definen el cuerpo de la función. Esto enfatiza la estructura funcional de los programas y ayuda a hacer que los programas sean más legibles.

2.4 Defina una regla convencional para el tamaño de la sangría que prefiera y a continuación aplíquela de manera uniforme. La tecla del tabulador puede ser utilizada para crear sangrías, pero los tabuladores pueden variar. Recomendamos utilizar ya sea tabuladores de 1/4 de pulgada o contar a mano tres espacios por cada uno de los niveles de sangría.

2.5 Aunque la inclusión de `<stdio.h>` es opcional, deberá ser incluida en cualquier programa C que utilice funciones de entrada/salida estándar de biblioteca. Esto ayuda al compilador a auxiliarle a usted a localizar errores en la fase de compilación de su programa, más bien que en la fase de ejecución (donde los errores resultan más costosos de corregir).

2.6. Coloque un espacio después de cada coma (,) para hacer los programas más legibles.

2.7 La selección de nombres de variables significativos ayuda a la autodocumentación de un programa, es decir, se requerirán de menos comentarios.

2.8 La primera letra de un identificador utilizado como un nombre simple de variable, deberá ser una letra minúscula. Más adelante en el texto le daremos significación especial a aquellos identificadores que empiezan con una letra mayúsculas y aquellos que utilizan todas mayúsculas.

2.9 Los nombres de variables de varias palabras pueden auxiliar a la legibilidad de un programa. Evite reunir las palabras separadas juntas como en `totalcommissions`. En vez de ello separe las palabras con subrayados como `total_commission`, o bien, si desea reunirlas, empiece cada palabra después del primero con una letra mayúscula como en `totalCommissions`.

2.10 Separe las declaraciones y los enunciados ejecutables en una función mediante una línea en blanco para enfatizar dónde terminan las declaraciones y dónde empiezan los enunciados ejecutables.

2.11 Coloque espacios en blanco a ambos lados de un operador binario. Esto hace resaltar al operador y permite que el programa sea mas legible.

2.12 Haga una sangría en el o los enunciados del cuerpo de una estructura `if`.

2.13 Coloque una línea en blanco antes y después de cada estructura de control en un programa para mejor legibilidad.

2.14 Deberá procurar que no exista más de un enunciado en cada línea de un programa.

2.15 Un enunciado largo puede ser dividido en varios renglones. Si un enunciado debe ser dividido en varias líneas, escoja puntos de corte que tengan sentido (como sería después de una coma en una lista separada por comas). Si un enunciado se divide en dos o más renglones, haga una sangría en todas las líneas subsecuentes.

2.16 Refiérase a la gráfica de precedencia de operadores al escribir expresiones que contengan muchos operadores. Confirme que los operadores en la expresión se ejecutan en el orden correcto. Si no está totalmente seguro del orden de evaluación en una expresión compleja, utilice paréntesis para obligar al orden, exactamente como lo haría en expresiones algebraicas. Asegúrese de observar que algunos de los operadores de C como el operador de asignación (=) se asocian de derecha a izquierda en vez de izquierda a derecha.

Sugerencia de portabilidad

2.1 Utilice identificadores de 31 o menos caracteres. esto auxilia a asegurar la portabilidad y puede evitar algunos errores sutiles de programación.

Ejercicios de autoevaluación

2.1 Llene cada uno de los siguientes espacios vacíos:

a) Cada programa en C empieza su ejecución en la función _____.

b) La _____ empieza el cuerpo de cada función y la _____ termina el cuerpo de cada función.

c) Cada enunciado termina con un _____

d) La función estándar de biblioteca _____ despliega información en la pantalla.

e) La secuencia de escape `\n` representa el carácter _____ que hace que se coloque el cursor en el principio de la siguiente línea en la pantalla.

f) La función estándar de biblioteca _____ se utiliza para obtener datos del teclado.

g) El especificador de conversión _____ se utiliza en una cadena de control de formato `scanf` para indicar que entrará un entero y en una cadena de control de formato `printf` para indicar que se sacará un entero.

h) Siempre que se coloca en una posición de memoria un nuevo valor, este valor borra el valor anterior en dicha posición. Este proceso se conoce como lectura _____.

i) Cuando se lee un valor desde una posición de memoria, el valor en dicha posición se conserva; esto se conoce como lectura _____.

j) El enunciado _____ se utiliza para tomar decisiones.

2.2 Indique si cada uno de los siguientes es verdadero o falso. Si es falso explique por qué.

a) Cuando la función `printf` es llamada ésta siempre empieza a imprimir en el principio de una nueva línea.

b) Los comentarios hacen que la computadora imprima el texto encerrado entre `/*` y `*/` en la pantalla al ejecutarse el programa.

c) La secuencia de escape `\n` cuando se utiliza en una cadena de control de formato `printf` hace que el cursor se coloque en el principio de la siguiente línea en pantalla.

d) Todas las variables deben ser declaradas antes de que puedan ser utilizadas.

e) Deben dársele un tipo a todas las variables cuando son declaradas.

f) C considera a las variables `number` y `NuMbEr` como idénticas.

g) Las declaraciones pueden aparecer en cualquier parte del cuerpo de una función.

h) Todos los argumentos que sigan a la cadena de control de formato en una función `printf` deben estar precedidas de un ampersand (`&`).

i) El operador de módulo (%) puede ser usado sólo con operandos enteros

j) Los operadores aritméticos `*`, `/`, `%`, `+`, y `-` tienen todos el mismo nivel de precedencia.

k) Verdadero o falso: los siguientes nombres de variable son idénticos en todos los sistemas de ANSI C.

```
thisisasuperduperlongname1234567
thisisasuperduperlongname1234568
```

l) Cierto o falso: un programa C que imprime tres líneas de salida debe contener tres enunciados `printf`

2.3 Escriba sólo un enunciado C para conseguir cada uno de los siguientes:

a) Declarar las variables `c`, `thisVariable`, `q76354`, y `number` para que sean del tipo `int`.

b) Indique al usuario para que escriba un entero. Termine su mensaje indicador con dos puntos (`:`), seguido por un espacio y deje el cursor colocado después de este espacio.

c) Lea un entero del teclado y almacene el valor escrito en una variable entera `a`.

d) Si la variable `number` no es igual a `7`, imprima `"The variable number is not equal to 7".`

e) Imprima el mensaje `"This is a C program."` sobre una sola línea.

f) Imprima el mensaje `"This is a C program."` sobre dos líneas donde la primera termine en C.

g) Imprima el mensaje `"This is a C program."` colocando cada palabra en una línea por separado.

h) Imprima el mensaje `"This is a C program."` con cada palabra separada por tabuladores.

2.4 Escriba un enunciado (o comentario) para cumplir con cada uno de lo siguiente:

a) Declare que un programa calculará el producto de tres enteros.

b) Declare las variables `x`, `y`, `z` y `result` que sean del tipo `int`.

c) Indíquele al usuario que escriba tres enteros.

d) Lea tres enteros del teclado y almacénelos en las variables `x`, `y`, `z`.

e) Calcule el producto de los tres enteros contenidos en las variables **x**, **y**, **z**, y asígnele el resultado a la variable `result`.
f) Imprima `"The product is"` seguido por el valor de la variable `result`.

2.5 Utilizando los enunciados que escribió en el ejercicio 2.4, escriba un programa completo que calcule el producto de tres enteros.

2.6 Identifique y corrija los errores de cada uno de los enunciados siguientes:

a)
```
printf ("The value is %d\n", &number);
```
b)
```
scanf("%d%d", &number1, number2);
```
c)
```
if (c < 7);
   printf("C is less than 7\n");
```
d)
```
if (c => 7)
   printf("C is equal to or less than 7\n");
```

Respuestas a los ejercicios de autoevaluación

2.1 a) `main`. b) llave izquierda ({), llave derecha (}). c)punto y coma. d) `printf`. e) nueva línea. f) `scanf`. g) %d. h) destructivo. i) no destructivo. j)`if`.

2.2 a) Falso. La función `printf` siempre comienza la impresión en el lugar donde esté colocado el cursor y esto puede ser en cualquier parte de una línea sobre la pantalla.
b) Falso. Los comentarios no hacen que se realice acción alguna al ejecutarse el programa. Se utilizan para documentar programas y mejorar su legibilidad.
c) Verdadero.
d) Verdadero.
e) Verdadero.
f) Falso. C diferencia minúsculas de mayúsculas, por lo tanto, estas variables cada una es diferente y única.
g) Falso. Las declaraciones deben aparecer después de la llave izquierda del cuerpo de una función y antes de cualquier enunciado ejecutable.
h) Falso. Los argumentos en una función `printf` por lo regular no deberán ser precedidas por un ampersand. Los argumentos que siguen a la cadena de control de formato en una función `scanf` normalmente deberán ser antecedidos por un ampersand. Analizaremos las excepciones en el capítulo 6 y 7.
i) Verdadero.
j) Falso. Los operadores `*`, `/` y `%` están en un mismo nivel de precedencia, y los operadores + y -, están en un nivel inferior de precedencia,
k) Falso. Algunos sistemas pudieran distinguir entre identificadores más largos de 31 caracteres.
l) Falso. Un enunciado `printf` con varias secuencias de escape **n** puede imprimir varias líneas.

2.3 a)
```
int c, thisVariable, q76354, number;
```
b)
```
printf("Enter an integer: ");
```
c)
```
scanf("%d", &a);
```
d)
```
if (number != 7)
   printf("The variable number is not equal to 7.\n");
```
e)
```
printf("This is a C program.\n");
```
f)
```
printf("This is a C\nprogram.\n");
```
g)
```
printf("This\nis\na\nC\nprogram.\n");
```
h)
```
printf("This\tis\ta\tC\tprogram.\n");
```

2.4 a)
```
/* Calculate the product of three integers */
```
b)
```
int x, y, z, result;
```
c)
```
printf("Enter three integers: ");
```


d)
```
scanf("%d%d%d", &x, &y, &z);
```
e)
```
result = x * y * z;
```
f)
```
printf("The product is %d\n", result);
```

2.5
```
/* Calculate the product of three integers */
#include <stdio.h>

main()
{
  int x, y, z, result;

  printf("Enter three integers: ");
  scanf("%d%d%d", &x, &y, &z);
  result = x * y * z;
  printf("The product is %d\n", result);
  return 0;
}
```

2.6 a) Error: `&number`. Corrección: eliminar el `&`. Más adelante en el texto analizaremos excepciones a esto.
b) Error: `number2` no tiene un ampersand. Corrección: `number2` debería decir `&number2`. Más adelante en el texto analizaremos excepciones a lo anterior.
c) Error: punto y coma después del paréntesis derecho de la condición en una enunciado `if`. Corrección: elimine el punto y coma después del paréntesis derecho. Nota: el resultado de este error es que el enunciado `printf` se ejecutará independiente de que la condición en el enunciado `if` resulte verdadera. El punto y coma después del paréntesis derecho se considera como un enunciado vacío —un enunciado que no hace nada.
d) Error: El operador relacional => debe de ser cambiado por >=.

Ejercicios

2.7 Identifique y corrija los errores en cada uno de los siguientes enunciados (Nota: pudieran existir más de un error por cada enunciado):

a)
```
scanf("d", value);
```
b)
```
printf("The product of %d and %d is %d"\n, x, y);
```
c)
```
firstNumber + secondNumber = sumOfNumbers
```
d)
```
if (number => largest)
   largest == number;
```
e)
```
*/ Program to determine the largest of three integers /*
```
f)
```
Scanf("%d", anInteger);
```
g)
```
printf("Remainder of %d divided by %d is\n", x, y, x % y);
```
h)
```
if (x = y);
   printf(%d is equal to %d\n", x, y);
```
i)
```
printf("The sum is %d\n," x + y);
```
j)
```
Printf ("The value you entered is: %d\n, &value);
```

2.8 Llene los espacios vacíos en cada uno de los siguientes:
a) _____ se utilizan para comentar un programa y mejorar su legibilidad.
b) La función utilizada para imprimir información en la pantalla es _____.
c) Un enunciado C que toma una decisión es _____.
d) Los cálculos por lo regular se ejecutan por enunciados _____.
e) La función _____ introduce valores del teclado.

2.9 Escriba solo un enunciado de C o una línea que cumpla con cada uno de lo siguiente:
a) Imprima el mensaje `"Enter two numbers."`
b) Asigne el producto de las variables **b** y **c** a la variable **a**.
c) Declare que un programa ejecuta un cálculo de muestra de nómina (es decir, utiliza texto que auxilia a documentar un programa).
d) Introduzca dos valores enteros del teclado y coloque estos valores en las variables enteras **a**, **b** y **c**.

2.10 Declare cuales de los siguientes son verdaderos y cuales son falsos. Explique sus respuestas.
a) Los operadores C se evalúan de izquierda a derecha.
b) Los siguientes son todos nombres válidos de variable: `_under_bar_`, `m928134`, `t5`, `j7`, `her_sales`, `his_account_total`, `a`, `b`, `c`, `z`, `z2`.
c) El enunciado `printf("a = 5;");` es un ejemplo típico de un enunciado de asignación.
d) Una expresión aritmética válida en C que no contenga paréntesis se evalúa de izquierda a derecha.
e) Lo siguientes son todos nombres inválidos de variables: `3g`, `87`, `67h2`, `h22`, `2h`.

2.11 Llene los espacios vacíos en cada uno de los siguientes:
a) ¿Qué operaciones aritméticas tienen el mismo nivel de precedencia que la multiplicación? _____.
b) ¿Cuando los paréntesis están anidados, qué conjunto de paréntesis serán calculados en primer término en una expresión aritmética? _____.
c) Una posición en la memoria de la computadora que puede contener valores diferentes en tiempos diferentes a lo largo de la ejecución de un programa se conoce como _____.

2.12. ¿Qué es lo que, si es que algo, se imprime cuando se ejecutan cada uno de los enunciados de C siguientes? Si no se imprime nada, entonces conteste "nada". Suponga que `x = 2 y y =3`

a) `printf("%d", x);`
b) `printf("%d", x + x);`
c) `printf("x=");`
d) `printf("x=%d", x);`
e) `printf("%d = %d", x + y, y + x);`
f) `z = x + y;`
g) `scanf ("%d%d", &x, &y);`
h) `/* printf("x + y = %d", x + y); */`
i) `printf ("\n");`

2.13 ¿Qué es, si es que es algo, de los enunciados siguientes contienen variables involucradas en lecturas destructivas?.

a) `scanf("%d%d%d%d%d", &b, &c, &d, &e, &f);`
b) `p = i + j +k + 7;`
c) `printf("Destructive read-in");`
d) `printf("a = 5");`

2.14 Dada la ecuación $y = ax^3 + 7$, ¿cuál de los que siguen, si es que existe alguno, son enunciados correctos de C correspondientes a esta ecuación?.

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

2.15 Declare el orden de cálculo de los operadores en cada uno de los enunciados de C siguientes, y muestre el valor de **x** después de que se ejecute cada uno de ellos.

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

2.16 Escriba un programa que solicite al usuario que introduzca dos números, tome los dos números del usuario, e imprima la suma, el producto, la diferencia, el cociente y el módulo de los dos números.

2.17 Escriba un programa que imprima los números 1 a 4 en un mismo renglón. Escriba el programa utilizando los siguientes métodos.
a) Utilizando un enunciado `printf` sin especificadores de conversión.
b) Utilizando un enunciado `printf` con cuatro especificadores de conversión.
c) Utilizando cuatro enunciados `printf`.

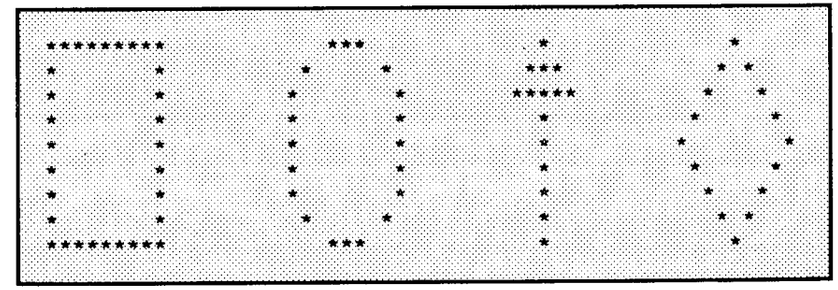
2.18 Escriba un programa que solicite al usuario que escriba dos enteros, tome los números del usuario y a continuación imprima el número mayor seguido por las palabras `"is larger"`. Si los números son iguales, que imprima el mensaje `"These numbers are equal"`. Utilice sólo la forma de una selección del enunciado `if` que aprendió en este capítulo.

2.19 Escriba un programa de C que entre tres enteros diferentes del teclado, y a continuación imprima la suma, el promedio, el producto, el más pequeño y el más grande de estos números. Utilice sólo la forma de una selección del enunciado `if`, que usted aprendió en este capítulo. El diálogo en pantalla deberá aparecer como sigue:

```
Input three different integers: 13 27 14
Sum is 54
Average is 18
Product is 4914
Smallest is 13
Largest is 27
```

2.20 Escriba un programa que lea el radio de un círculo y que imprima el diámetro del mismo, su circunferencia y su área.Utilice el valor constante 3.14159 para "pi". Efectúe cada uno de estos cálculos dentro del enunciado o enunciados `printf` y utilice el especificador de conversión `%f`. (Nota: en este capítulo, hemos estudiado únicamente constantes y variables enteras. En el capítulo 3 veremos números de punto flotante, es decir, valores que pueden tener puntos decimales).

2.21 Escriba un programa que imprima un recuadro, un oval, una flecha y un diamante, como sigue:



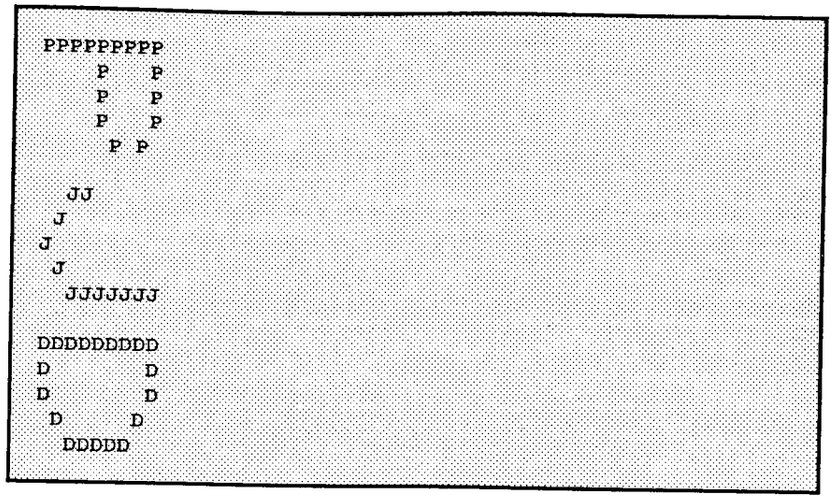
2.22 ¿Qué es lo que imprime el código siguiente?

```
printf("*\n**\n***\n****\n*****\n");
```

2.23 Escriba un programa que lea cinco enteros y a continuación determine e imprima cuáles son el mayor y el menor entero en el grupo. Utilice sólo las técnicas de programación que aprendió en este capítulo.

2.24 Escriba un programa que lea un entero y determine e imprima si es par o impar. (*Sugerencia:* utilice el operador de módulo. Un número par es un múltiplo de dos. Cualquier múltiplo de dos deja un residuo de cero al ser dividido entre dos).

2.25 Imprima sus iniciales en letras de bloque hacia abajo de la página. Construya cada letra de bloque utilizando las letras que representa como sigue:



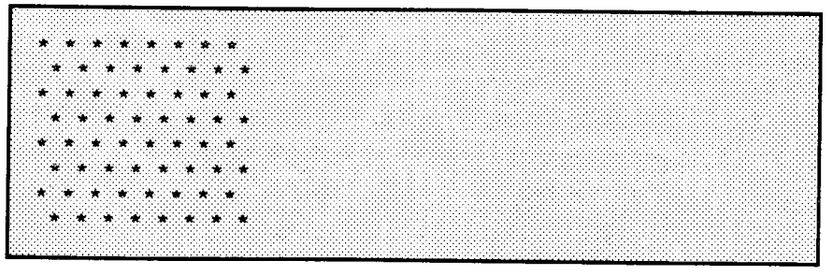
```
PPPPPPPPP
    P    P
    P    P
    P    P
     P P

  JJ
 J
J
 J
  JJJJJJJ

DDDDDDDDD
D        D
D        D
 D      D
  DDDDD
```

2.26 Escriba un programa que lea dos enteros y que determine e imprima si el primero es un múltiplo del segundo. (*Sugerencia:* utilice el operador de módulo).

2.27 Despliegue un patrón cuadriculado utilizando ocho enunciados `printf`, y a continuación despliegue el mismo patrón con el mínimo posible de enunciados `printf`.



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

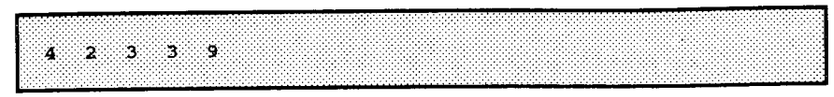
2.28 Distinga entre los términos error fatal y error no fatal. ¿Por qué preferiría experimentar un error fatal en vez de un error no fatal?

2.29 He aquí una mirada hacia adelante. En este capítulo usted aprendió los enteros y el tipo `int`. C también puede representar letras mayúsculas, letras minúsculas y una gran variedad de símbolos especiales. C utiliza enteros pequeños en forma interna para representar cada carácter diferente. El conjunto de caracteres que utiliza una computadora y las representaciones de entero correspondientes para estos caracteres se conoce como el conjunto de caracteres de la computadora. Por ejemplo, usted puede imprimir el equivalente en enteros de la `A`, ejecutando el enunciado

```
printf("%d", 'A');
```

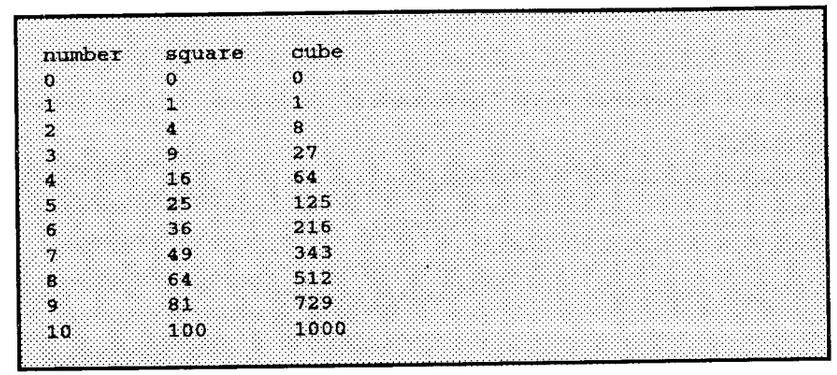

Escriba un programa C que imprima los equivalentes en enteros de algunas letras mayúsculas, letras minúsculas, dígitos y símbolos especiales. Como mínimo, determine los equivalentes en enteros de los siguientes: `A B C a b c 0 1 2 $ * + /` así como del carácter de espacio en blanco.

2.30 Escriba un programa que entre un número de cinco dígitos, separe el número en sus dígitos individuales e imprima los dígitos separados unos de otros mediante tres espacios. Por ejemplo, si el usuario escribe `42339` el programa debería imprimir



```
4   2   3   3   9
```

2.31 Utilizando sólo las técnicas aprendidas en este capítulo, escriba un programa que calcule los cuadrados y los cubos de los números del 1 al 10 y que utilice tabuladores para imprimir la siguiente tabla de valores:



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

3

Desarrollo de programas estructurados

Objetivos

- Comprender las técnicas fundamentales de resolución de problemas.
- Tener la capacidad de desarrollar algoritmos mediante el proceso de refinamiento descendente paso a paso.
- Tener la capacidad de utilizar la estructura de selección `if` y la estructura de selección `if/else` para elegir acciones.
- Tener la capacidad de utilizar la estructura de repetición `while` para ejecutar enunciados repetidamente en un programa.
- Comprender la repetición controlada por contador y la repetición controlada por centinela.
- Comprender la programación estructurada.
- Tener la capacidad de utilizar los operadores incrementales, decrementales, y de asignación.

El secreto del éxito es la constancia en el propósito.
Benjamin Disraeli

Movámonos todos un lugar.
Lewis Carroll

La rueda ha dado un giro completo.
William Shakespeare
King Lear

¡Cuántas manzanas habrán caído sobre la cabeza de Newton antes que comprendiera lo que le estaban sugiriendo!
Robert Frost
Comment

Sinopsis

Resumen • Terminología • Errores comunes de programación • Prácticas sanas de programación • Sugerencias de rendimiento • Observaciones de ingeniería de software • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios.

3.1 Introducción

Antes de escribir un programa para resolver un problema particular, es esencial tener comprensión completa del mismo, y un método planeado de forma cuidadosa para su resolución. Los dos capítulos siguientes analizan técnicas que facilitan el desarrollo de los programas de cómputo estructurados. En la Sección 4.11, presentamos un resumen de la programación estructurada, que reúne las técnicas desarrolladas tanto aquí, como en el capítulo. 4.

3.2 Algoritmos

La solución a cualquier problema de cómputo involucra la ejecución de una serie de acciones, en un orden específico. Un *procedimiento* para resolver un problema en términos de

1. las *acciones* a ejecutarse, y
2. el *orden* en el cual estas acciones deben de ejecutarse

se llama un *algoritmo*. El siguiente ejemplo demuestra la importancia de especificar de forma correcta el orden en el cual se deben de ejecutar las acciones.

Veamos el "algoritmo de levantarse" que debe de seguir un ejecutivo junior para salir de la cama y llegar al trabajo:

Salir de la cama.
Quitarse los pijamas.
Darse una ducha.
Vestirse.
Desayunar.
Utilizar el vehículo común para llegar al trabajo.

Mediante esta rutina el ejecutivo llega al trabajo bien preparado para tomar decisiones críticas. Suponga, sin embargo, que los mismos pasos se llevan a cabo en un orden ligeramente distinto:

Salir de la cama.
Quitarse los pijamas.
Vestirse.
Darse una ducha.
Desayunar.
Utilizar el vehículo común para llegar al trabajo.

En este caso, nuestro ejecutivo junior aparecerá en el trabajo totalmente mojado. La especificación del orden en un programa de cómputo en el cual los enunciados deben ser ejecutados se conoce como *control de programa*. En éste y en el capítulo siguiente, investigaremos las capacidades de control de programa de C.

3.3 Seudocódigo

El *seudocódigo* es un lenguaje artificial e informal que auxilia a los programadores a desarrollar los algoritmos. El seudocódigo que presentamos aquí es en particular útil para desarrollar algoritmos que deberán ser convertidos en programas estructurados de C. El seudocódigo es similar al inglés coloquial; es cómodo y amigable, aunque no se trate de un lenguaje verdadero de programación de computadoras.

De hecho, los programas en seudocódigo no son ejecutados sobre computadoras. Más bien, sólo ayudan al programador "a pensar" un programa, antes de intentar escribirlo en un lenguaje de programación como C. En este capítulo, damos varios ejemplos de cómo se puede utilizar con eficacia el seudocódigo, en el desarrollo de programas estructurados de C.

El seudocódigo consiste solo de caracteres, por lo que los programadores pueden de forma cómoda escribir los programas en seudocódigo en una computadora, utilizando un programa de edición. Sobre demanda la computadora puede desplegar o imprimir una copia nueva en seudocódigo del programa. Un programa preparado cuidadosamente en seudocódigo, puede ser convertido con fácilidad en el programa C correspondiente. Esto se lleva a cabo en muchos casos sólo remplazando enunciados en seudocódigo por sus equivalentes en C.

El seudocódigo incluye sólo enunciados de acción aquellos que deben ser ejecutados cuando el programa haya sido convertido de seudocódigo a C, y luego ejecutado en C. Las declaraciones no son enunciados ejecutables. Son mensajes para el compilador. Por ejemplo, la declaración

```
int i;
```

sólo le indica al compilador el tipo de la variable `i`, así como instruye al compilador que reserve espacio en memoria para esta variable. Pero esta declaración no causa ninguna acción como sería entrada, salida o cálculo para que se ejecute o ocurra al ejecutarse el programa. Algunos programadores deciden enlistar al principio de un programa en seudocódigo cada variable y mencionar de forma breve el objeto de cada una de ellas. Repetimos, el seudocódigo es una ayuda informal para el desarrollo del programa.

3.4 Estructuras de control

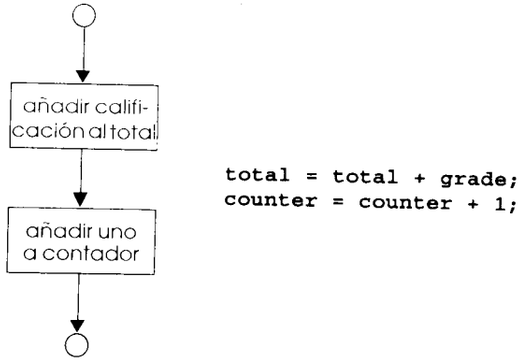
Por lo regular, en un programa los enunciados son ejecutados uno después del otro, en el orden en que aparecen escritos. Esto se conoce como *ejecución secuencial*. Varios enunciados de C, que pronto analizaremos, le permiten al programador especificar que el enunciado siguiente a ejecutar pueda ser otro diferente del que sigue en secuencia. Esto se conoce como *transferencia de control*.

Durante los años 60 , se hizo claro que el uso indiscriminado de transferencias de control era la causa de gran cantidad de dificultades experimentadas por los grupos de desarrollo de software. El dedo acusador apuntaba al *enunciado* **goto**, que le permite al programador especificar una transferencia de control a uno de amplia gama de destinos posibles, dentro de un programa. La noción de lo que se conoce como *programación estructurada* se convirtió prácticamente en sinónimo de "*eliminación de* **goto**".

Las investigaciones de Bohm y de Jacopini[1] habían demostrado que los programas podían ser escritos sin ningún enunciado **goto**. Para los programadores el reto se convirtió en modificar sus estilos a "programación sin **goto**". Y no fue sino hasta entrados los años 70' que la profesión de la programación en general empezó a tomar en serio la programación estructurada. Los resultados han sido impresionantes, ya que los grupos de desarrollo de software han informado reducciones en tiempos de desarrollo, entrega a tiempo más frecuente de sistemas y terminación dentro de presupuesto más frecuente de los proyectos de software. La clave de estos éxitos es simplemente que los programas producidos con las técnicas estructuradas, son más claros, más fáciles de depurar y de modificar, y tal vez más libres de fallas desde el primer momento.

El trabajo de Bohm y Jacopini demostró que todos los programas podrían ser escritos en términos de sólo tres *estructuras de control*, a saber, la *estructura de secuencia*, la *estructura de selección* y la *estructura de repetición*. En C la estructura de secuencia está en esencia interconstruída. A menos de que se indique lo contrario, la computadora ejecutará automáticamente enunciados C, uno después de otro, en el orden en el cual se han escrito. El segmento de *diagrama de flujo* de la figura 3.1 ilustra la estructura de secuencias de C.

Un diagrama de flujo es una representación gráfica de un algoritmo o de una porción de un algoritmo. Los diagramas de flujo se trazan utilizando ciertos símbolos de uso especial como son




```
total = total + grade;
counter = counter + 1;
```

Fig. 3.1 Estructura de secuencia de diagrama de flujo de C.

[1] Bohm, C y G. Jacopini "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, Mayo 1966, pp. 336-371.

rectángulos, diamantes, óvalos y pequeños círculos; estos símbolos están conectados entre sí por flechas, conocidas como *líneas de flujo*.

Al igual que el seudocódigo, los diagramas de flujo son útiles para el desarrollo y la representación de algoritmos, aunque la mayor parte de los programadores prefieren el seudocódigo. Los diagramas de flujo muestran con claridad cómo operan las estructuras de control; sólo para eso es que en este texto los utilizaremos.

Vea el segmento de diagrama de flujo correspondiente a la estructura de secuencia de la figura 3.1. Utilizamos un *símbolo rectángulo*, también conocido como *símbolo de acción*, para indicar cualquier tipo de acción, incluyendo un cálculo o una operación de entrada/salida. Las líneas de flujo de la figura indican el orden en el cual las acciones se ejecutarán primero, **grade** deberá ser sumado a **total** y a continuación **1** deberá ser sumado a **counter**. C nos permite tener todas las acciones que deseemos en una estructura de secuencia. Como pronto veremos, en cualquier lugar donde una acción se pueda colocar, podremos colocar varias acciones en secuencia.

Al trazar un diagrama de flujo que represente un algoritmo completo, el primer símbolo utilizado en el diagrama es un *símbolo oval* que contiene la palabra "Inicio"; y el último símbolo utilizado también es un símbolo oval que contiene la palabra "Fin". Al dibujar sólo una porción de un algoritmo, como en el caso de la figura 3.1, se omiten los símbolos ovales, y en su lugar se utilizan *símbolos de pequeños círculos*, que también se conocen como *símbolos de conexión*.

Quizás el símbolo de diagrama de flujo más importante es el *símbolo diamante*, también conocido como *símbolo de decisión*, mismo que indica donde se debe tomar una decisión. Analizaremos el símbolo diamante en la siguiente sección.

C proporciona tres tipos de estructuras de selección. La estructura de selección **if** (Sección 3.5), ya sea ejecute (elige) una acción, si una condición es verdadera, o pasa por alto la acción, si la condición es falsa. La estructura de selección **if/else** (Sección 3.6) ejecuta una acción, si la condición es verdadera, o ejecuta una acción diferente, si la condición es falsa. La estructura de selección **switch** (que se estudia en el capítulo 4) ejecuta una de entre muchas acciones diferentes, dependiendo del valor de una expresión.

La estructura **if** se llama *estructura de una sola selección*, porque selecciona o ignora una acción. La estructura **if/else** se conoce como *estructura de doble selección*, porque selecciona entre dos acciones distintas. La estructura **switch** se conoce como una *estructura de selección múltiple*, porque selecciona entre muchas acciones diferentes.

C proporciona tres tipos de estructuras de repetición, es decir **while** (Sección 3.7), así como **do/while** y **for** (ambas analizadas en el capítulo 4).

Es todo. C tiene sólo siete estructuras de control: de secuencia, tres tipos de selección y tres de repetición. Cada programa de C se forma al combinar tantos de cada tipo de estructura de control como sean apropiados, en relación con el algoritmo que resuelve el programa. Como en el caso de la estructura de secuencia de la figura 3.1, veremos que cada estructura de control contiene dos símbolos de círculo pequeños, uno en el punto de entrada a la estructura de control y uno en el punto de salida. Estas *estructuras de control de una sola entrada/una sola salida* facilitan la construcción de programas. Las estructuras de control pueden ser agregadas unas a otras, conectando el punto de salida de una estructura de control con el punto de entrada de la siguiente. Esto es muy parecido a la forma en que los niños apilan bloques de construcción, por lo que llamamos lo anterior *apilamiento de estructuras de control*. Aprenderemos que sólo hay otra forma en que las estructuras de control puedan quedar conectadas—un método conocido como *anidar estructuras de control*. Por lo tanto, cualquiera que sea el programa C que tengamos que construir en el futuro, podrá ser elaborado partiendo de sólo siete tipos diferentes de estructuras de control, combinadas de dos distintas formas.

3.5 La estructura de selección if

Una estructura de selección se utiliza para elegir entre cursos alternativos de acción. Por ejemplo, suponga que en un examen 60 es la calificación de aprobado. El enunciado en seudocódigo

If student's grade is greater than or equal to 60
Print "Passed"

determina si la condición "la calificación del estudiante es mayor que o igual a 60" es verdadera o falsa. Si la condición es verdadera, entonces se imprime "Aprobó" y el siguiente enunciado en seudocódigo en orden se "ejecuta" (recuerde que el seudocódigo no es un lenguaje de programación verdadero). Si la condición resulta falsa, se ignora la impresión, y se ejecuta el siguiente enunciado del seudocódigo en su orden. Note que la segunda línea de esta estructura de selección está con sangría. Esta sangría es opcional, pero es muy recomendada, ya que auxilia a enfatizar la estructura inherente de los programas estructurados. Aplicaremos convenciones de sangría de forma cuidadosa a lo largo de este texto. El compilador de C ignora los *espacios en blanco* como son los espacios en blanco, los tabuladores y las nuevas líneas que son utilizadas para sangrías y para el espaciamiento vertical.

Prácticas sanas de programación 3.1

La aplicación consistente de reglas convencionales responsables para las sangrías mejora en forma importante la legibilidad de los programas. Sugerimos un tabulador de tamaño fijo de aproximadamente 1/4 de pulgada o de tres espacios por cada sangría.

El enunciado anterior en seudocódigo *If* puede ser escrito en C como

```
If (grade >= 60)
  printf("Passed\n");
```

Advierta que el código C sigue de cerca al seudocódigo. Esta es una de las propiedades del seudocódigo que lo hace una herramienta tan útil para el desarrollo de programas.

Práctica sana de programación 3.2

A menudo se utiliza el seudocódigo durante el proceso de diseño para "pensar en voz alta" un programa. Posteriormente el programa en seudocódigo se convierte a C.

El diagrama de flujo de la figura 3.2 ilustra la estructura **if** de una sola selección. Este diagrama de flujo contiene el símbolo quizás de mayor importancia en la diagramación de flujo el *símbolo diamante*, también conocido como *símbolo de decisión*, y que indica que una decisión debe ser tomada. El símbolo de decisión contiene una expresión, como es una condición, que puede resultar verdadera o falsa. Dos líneas de flujo parten del símbolo de decisión. Una indica la dirección a tomarse cuando la expresión dentro del símbolo es verdadera; la otra indica la dirección a tomarse cuando la expresión es falsa. Aprendimos en el capítulo 2 que las decisiones pueden ser tomadas, basadas en condiciones que contengan operadores relacionales o de igualdad. De hecho, se puede tomar una decisión basándose en cualquier expresión si la expresión se iguala a cero, se considera como falsa, y si la expresión se iguala a no cero, se considera como verdadera.

Note que la estructura **if**, también, es una estructura de una entrada/una salida. Pronto aprenderemos que los diagramas de flujo para las demás estructuras de control también contienen (además de símbolos de pequeños círculos y líneas de flujo) sólo símbolos rectangulares para indicar acciones a ejecutarse, y símbolos diamante para indicar decisiones a tomarse. Este es el modelo de programación acción/decisión, sobre el cual estamos haciendo énfasis.

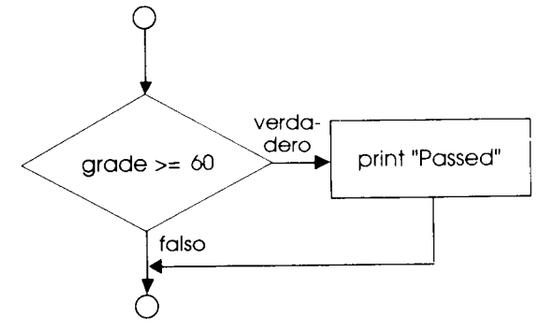


Fig. 3.2 Diagrama de flujo en C de estructura de una selección.

Podemos visualizar siete contenedores, cada uno de ellos conteniendo sólo estructuras de control de uno de los siete tipos. Estas estructuras de control están vacías. No hay nada escrito en los rectángulos y en los diamantes. La tarea de los programadores, entonces, es ensamblar un programa utilizando tantos de cada tipo de estructuras de control como el algoritmo demande, combinando estas estructuras de control, en sólo dos formas posibles (apilando o anidado), y a continuación rellenando las acciones y las decisiones de forma adecuada para el algoritmo. Analizaremos la variedad de formas en las cuales las acciones y las decisiones pueden quedar escritas.

3.6 La estructura de selección if/else

La estructura de selección **if** ejecuta una acción indicada sólo cuando la condición es verdadera; de lo contrario la acción es pasada por alto. La estructura de selección **if/else** permite que el programador especifique que se ejecuten acciones distintas cuando la condición sea verdadera que cuando la condición sea falsa. Por ejemplo, el enunciado en seudocódigo

If student's grade is greater than or equal to 60
Print "Passed"
else
Print "Failed"

imprime *Passed*, si la calificación del alumno es mayor que o igual a 60 e imprime *Failed* si la calificación del alumno es menor de 60. En cualquiera de los casos, después de haber terminado la impresión, se ejecutará el siguiente enunciado del seudocódigo. Advierta que el cuerpo de *else* también queda con sangría.

Práctica sana de programación 3.3

Haga sangrías en ambos cuerpos de los enunciados de una estructura **if/else**.

Cualquier regla convencional de sangrías que escoja deberá aplicarse con cuidado a lo largo de sus programas. Un programa que no siga reglas de espaciamiento uniformes es difícil leer.

Práctica sana de programación 3.4

Si existen varios niveles de sangría, cada nivel deberá tener sangría con una cantidad igual de espacio.

El seudocódigo anterior correspondiente a la estructura *If/else* puede ser escrito en C como

```
if (grade  >= 60)
   printf("Passed\n");
else
   printf("Failed\n");
```

El diagrama de flujo de la figura 3.3 ilustra muy bien el flujo de control de la estructura **if/else**. Otra vez, advierta que (además de los pequeños círculos y flechas) los únicos símbolos en el diagrama de flujo son rectángulos (para las acciones) y un diamante (para una toma de decisiones). Continuamos enfatizando este modelo de computación de acción/decisión. Imagínese otra vez un contenedor profundo, que contenga tantas estructuras de doble selección vacías como sean necesarias, para elaborar cualquier programa en C. Otra vez la tarea del programador es ensamblar estas estructuras de selección (mediante apilamiento y anidación), con cualesquiera otras estructuras de control requeridas por el algoritmo, y rellenar los rectángulos y diamantes vacíos con acciones y decisiones, apropiadas al algoritmo que se está implantando.

C tiene el *operador condicional (?:)* que está relacionado de cerca con la estructura **if/else**. El operador condicional es el único *operador ternario* de C —utiliza tres operandos. Los operandos, junto con el operador condicional, conforman una *expresión condicional*. El primer operando es una condición, el segundo operando es el valor de toda la expresión condicional si la condición es verdadera, y el tercer operando es el valor de toda la expresión condicional si la condición es falsa. Por ejemplo, el enunciado **printf**

```
printf("%s\n", grade >= 60 ? "Passed" : "Failed");
```

contiene una expresión condicional, que evalúa la cadena literal como **"Passed"** si la condición **grade >= 60** es verdadera y evalúa la cadena literal como **"Failed"** si la condición es falsa.

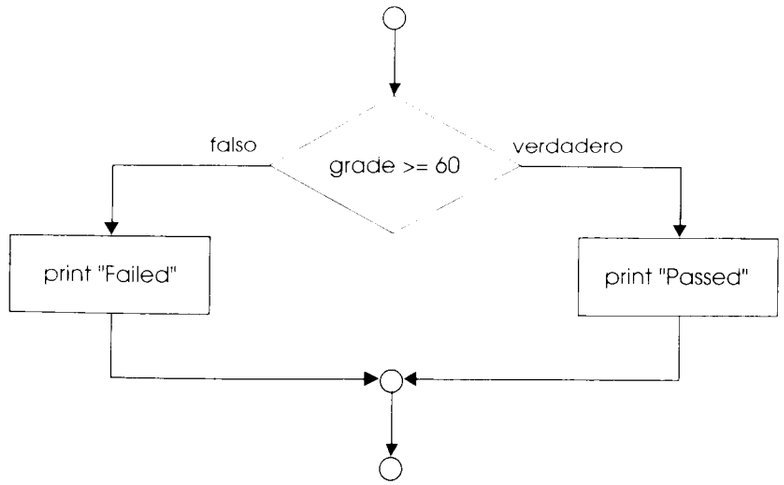


Fig. 3.3 Diagrama de flujo en C de la estructura de doble selección **if/else**.

La cadena de control de formato de **printf** contiene la especificación de conversión **%s**, para la impresión de una cadena de caracteres. Por lo tanto, el enunciado anterior **printf** lleva a cabo en esencia lo mismo que el enunciado anterior **if/else**.

Los valores en una expresión condicional también pueden ser acciones a ejecutar. Por ejemplo, la expresión condicional

```
grade  >= 60 ? printf("Passed\n") : printf("Failed\n");
```

Se lee como sigue "si **grade** es mayor que o igual a **60**, entonces **printf("Passed\n")**, y de lo contrario **printf("Failed\n")**". Esto, también, resulta comparable a la estructura anterior **if/else**. Veremos que los operadores condicionales pueden ser utilizados en algunas situaciones donde los enunciados **if/else** no pueden ser usados.

Las *estructuras* ***if/else*** *anidadas* prueban para muchos casos, colocando estructuras **if/else** dentro de estructuras **if/else**. Por ejemplo, el siguiente enunciado en seudocódigo imprimirá **A** para calificaciones de examen mayores que o iguales a **90**, **B** para calificaciones mayores que o iguales a **80**, **C** para calificaciones mayores que o iguales a **70**, **D** para calificaciones mayores que o iguales a **60**, y **F** para todas las demás calificaciones.

```
(If) Si la calificación del alumno es mayor que o igual a 90
     Imprima "A"
(else) de lo contrario
     (If) Si la calificación del alumno es mayor que o igual a 80
          Imprima "B"
     (else) de lo contrario
          (If) Si la calificación del alumno es mayor que o igual a 70
               Imprima "C"
          (else) de lo contrario
               (If) Si la calificación del alumno es mayor que o igual a 60
                    Imprima "D"
               (else) de lo contrario
                    Imprima "F"
```

Este seudocódigo puede ser escrito en C como

```
if (grade >= 90)
   printf("A\n");
else
   if (grade >= 80)
      printf("B\n");
   else
      if (grade >= 70)
         printf("C\n");
      else
         if (grade >= 60)
            printf("D\n");
         else
            printf("F\n");
```

Si la variable **grade** es mayor que o igual a 90, las primeras cuatro condiciones resultarán ciertas, pero sólo se ejecutará el enunciado **printf** después de la primera prueba. Después de que se

haya ejecutado ese **printf**, la parte **else** del enunciado "exterior" **if/else** será pasado por alto. Muchos programadores de C prefieren escribir la estructura **if** anterior, como sigue:

```
if (grade >= 90)
   printf("A\n");
else if (grade >= 80)
   printf("B\n");
else if (grade >= 70)
   printf("C\n");
else if (grade >= 60)
   printf("D\n");
else
   printf("F\n");
```

Por lo que se refiere al compilador de C, ambas formas resultan equivalentes. Esta última forma es popular porque evita las profundas sangrías del código hacia la derecha. A menudo dichas sangrías dejan poco espacio sobre la línea, obligando a la división de líneas y, por lo tanto, reduciendo la legibilidad del programa.

La estructura de selección **if** espera un enunciado dentro de su cuerpo. Para incluir varios enunciados en el cuerpo de un **if**, encierre el conjunto de enunciados entre llaves ({ y }). Un conjunto de enunciados contenidos dentro de un par de llaves se conoce como un *enunciado compuesto*.

Observación de ingeniería de software 3.1

Un enunciado compuesto puede ser colocado en cualquier parte de un programa, donde pueda ser colocado un enunciado sencillo.

El siguiente ejemplo incluye un enunciado compuesto en la parte **else** de una estructura **if/else**.

```
if (grade >= 60)
   printf("Passed.\n");
else {
   printf("Failed.\n");
   printf("You must take this course again.\n");
}
```

En este caso, si la calificación es menor que 60, el programa ejecutará los dos enunciados **printf** existentes en el cuerpo de **else** e imprimirá

```
Failed.
You must take this course again.
```

Advierta las llaves que rodean ambos enunciados en la cláusula **else**. Estas llaves son importantes. Sin ellas, el enunciado

```
printf("You must take this course again.\n");
```

quedaría fuera del cuerpo de la parte **else** del **if**, y sería ejecutada, independientemente de si la calificación es menor que 60.

Error común de programación 3.1

Olvidar una o ambas de las llaves que delimitan un enunciado compuesto.

Un error de sintaxis será detectado por el compilador. Un error lógico hará su efecto durante la ejecución. Un error lógico fatal hará que un programa falle y que termine de forma prematura. Un error lógico no fatal permitirá que continúe el programa, pero produciendo resultados incorrectos.

Error común de programación 3.2

*Colocar en una estructura **if** un punto y coma después de la condición, llevará a un error lógico en estructuras **if** de una sola selección y a un error de sintaxis en estructuras **if** de doble selección.*

Práctica sana de programación 3.5

Algunos programadores prefieren escribir las llaves de principio y de terminación de los enunciados compuestos antes de escribir en el interior de dichas llaves los enunciados individuales. Esto les ayuda a evitar la omisión de una o ambas de las llaves.

Observación de ingeniería de software 3.2

Igual que un enunciado compuesto puede ser colocado en cualquier lugar donde pudiera colocarse un enunciado sencillo, también es posible que no haya ningún enunciado de ningún tipo, es decir, un enunciado vacío. El enunciado vacío se representa colocando un punto y coma (;) donde por lo regular debería estar el enunciado.

En esta sección, hemos presentado la noción de un enunciado compuesto. Un enunciado compuesto puede contener declaraciones (como lo hace el cuerpo de **main**, por ejemplo). Si es así, el enunciado compuesto se conoce como un *bloque*. Las declaraciones dentro de un bloque deben colocarse al principio de éste, antes de cualquier enunciado de acción. En el capítulo 5 analizaremos la utilización de los bloques. Hasta entonces, el lector deberá tratar de evitar de utilizar bloques (a excepción del cuerpo de **main**, naturalmente).

3.7 La estructura de repetición while

Una *estructura de repetición* le permite al programador especificar que se repita una acción, en tanto cierta condición se mantenga verdadera. El enunciado en seudocódigo

(While) En tanto queden elementos en mi lista de compras
Adquirir elemento siguiente y tacharlo de la lista

describe la repetición que ocurre durante una salida de compras. La condición, "there are more items on my shopping list" puede ser verdadera o falsa. Si es verdadera, entonces la acción, "Purchase next item and cross it off my list" se ejecutará. Esta acción se ejecutará en forma repetida, en tanto la condición sea verdadera. El enunciado o enunciados contenidos en la estructura de repetición *while* constituyen el cuerpo del *while*. El cuerpo de la estructura *while* puede ser un enunciado sencillo o un enunciado compuesto.

Eventualmente, la condición se hará falsa (cuando se haya adquirido el último elemento de la lista de compras y se haya tachado de la misma). Llegado a este punto, la repetición se termina, y se ejecutará el enunciado en seudocódigo que sigue de inmediato después de la estructura de repetición.

Error común de programación 3.3

*No incluir en el cuerpo de una estructura **while** una acción que haga que la condición existente en **while** en algun momento se convierta en falsa. Por lo regular, esta estructura de repetición no terminará jamás —un error conocido como "ciclo infinito".*

Error común de programación 3.4

Escribir la palabra clave `while` *con una* `W` *como* `While` *(recuerde que C es un lenguaje que es sensible a la caja tipográfica). Todas las palabras reservadas de C, como* `while`, `if` *y* `else`, *sólo contienen letras minúsculas.*

Como ejemplo de un **while** real, considere un segmento de programa diseñado para encontrar la primera potencia de 2 superior a 1000. Suponga la variable de entero **product** inicializada a 2. Cuando la estructura de repetición **while** siguiente termine de ejecutarse, **product** contendrá la respuesta deseada:

```
product = 2;

while (product <= 1000)
product = 2 * product;
```

El diagrama de flujo de la figura 3.4 enseña con claraidad el flujo de control en la estructura de repetición **while.** Una vez más, advierta que (en adición a los pequeños círculos y flechas) el diagrama de flujo contiene un símbolo rectangular y un diamante. Imagine, de nuevo, un contenedor profundo de estructuras vacías **while,** que pueden ser apiladas y anidadas con otras estructuras de control, para formar una implantación estructurada del flujo de control de un algoritmo. Los rectángulos y diamantes vacíos, se llenarán entonces con decisiones y acciones apropiadas. El diagrama de flujo muestra con claridad la repetición. La línea de flujo que parte del rectángulo retorna a la decisión misma, que es probada cada vez dentro del ciclo, hasta que de forma eventual la decisión se convierte en falsa. Llegado este momento, se sale de la estructura **while** y el control pasa al siguiente enunciado del programa.

Cuando se escribe la estructura **while,** el valor de **product** es 2. La variable **product** se multiplica repetidamente por 2, asumiendo los valores 4, 8, 16, 32, 64, 128, 256, 512 y 1024 sucesivamente. Cuando **product** se convierte en 1024, la condición en la estructura **while, product <= 1000,** se hace falsa. Con ello se termina la repetición y el valor final de **product** es 1024. La ejecución del programa continúa con el enunciado que sigue después de **while.**

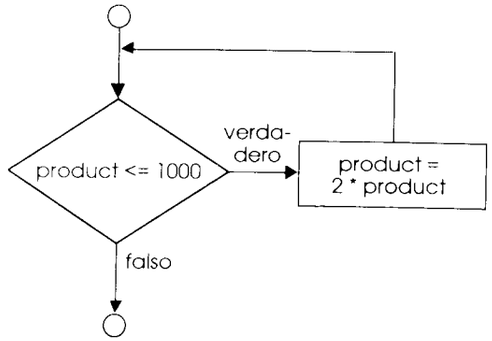


Fig. 3.4 Diagrama de flujo de la estructura de repetición **while.**

3.8 Cómo formular algoritmos: Estudio de caso 1 (repetición controlada por contador)

A fin de ilustrar cómo se desarrollan los algoritmos, resolveremos algunas variantes de un programa de promedios de clase. Considere el siguiente enunciado de programa:

> *Una clase de diez alumnos hizo un examen. Las calificaciones (enteros en el rango 0 a 100) correspondientes a este examen están a su disposición. Determine el promedio de la clase en este examen.*

El promedio de la clase es igual a la suma de las calificaciones dividida por el número de alumnos. El algoritmo para resolver este problema en una computadora, debe introducir cada una de las calificaciones, ejecutar el cálculo de promedio e imprimir el resultado.

Utilicemos el seudocódigo, enlistemos las acciones a ejecutarse, y especifiquemos el orden en el cual estas acciones deberán ser ejecutadas. Utilizaremos *repetición controlada por contador,* para introducir las calificaciones una a la vez. Esta técnica utiliza una variable llamada *contador* para definir el número de veces que deberá ejecutarse un conjunto de enunciados. En este ejemplo, la repetición terminará cuando el contador exceda de 10. En esta sección presentamos el algoritmo en seudocódigo (figura 3.5), así como el programa en C correspondiente (figura 3.6). En la siguiente sección, mostraremos cómo se desarrollan los algoritmos en seudocódigo. La repetición controlada por contador se conoce a menudo como *repetición definida* porque, antes de que se inicie la ejecución del ciclo, el número de repetición es conocido.

Advierta las referencias en el algoritmo a un total y a un contador. Un *total* es una variable, utilizada para acumular la suma de una serie de valores. Un contador es una variable, utilizada para contar en este caso, para contar el número de calificaciones capturadas. Por lo regular las variables utilizadas para almacenar totales, deberán ser inicializadas a cero antes de ser utilizadas en un programa; de lo contrario la suma incluiría el valor anterior almacenado en la posición en memoria de ese total. Las variables de contador se inicializan a cero o a uno, dependiendo en su uso (presentaremos ejemplos mostrando cada uno de ellos). Una variable sin inicializar contiene un *valor "basura"* que es el valor almacenado por última vez, en la posición de memoria reservada para la misma.

Error común de programación 3.5

Si no se inicializa un contador o un total, los resultados de su programa probablemente estarán incorrectos. Esto es un ejemplo de un error lógico.

```
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
    Input the next grade
    Add the grade into the total
    Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

Fig. 3.5 Algoritmo en seudocódigo, que utiliza repetición controlada por contador para resolver el problema de promedio de clase.

```
/* Class average program with
   counter-controlled repetition */
#include <stdio.h>

main()
{
   int counter, grade, total, average;

   /* initialization phase */
   total = 0;
   counter = 1;

   /* processing phase */
   while (counter <= 10) {
      printf("Enter grade: ");
      scanf("%d", &grade);
      total = total + grade;
      counter = counter + 1;
   }

   /* termination phase */
   average = total / 10;
   printf("Class average is %d\n", average);

   return 0;   /* indicate program ended successfully */
}
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

Fig. 3.6 Programa en C y ejecución de muestra del problema de promedio de clase utilizando repetición controlada por contador.

Práctica sana de programación 3.6

Inicialice contadores y totales.

Advierta que el cálculo de promedios incluído en el programa produjo un resultado entero. De hecho, en este ejemplo, la suma de las calificaciones es 817, misma que al dividirse entre 10 debería resultar en 81.7, es decir, un número con un punto decimal. Veremos cómo enfrentar estos números (conocidos como números de punto flotante) en la sección siguiente.

3.9 Cómo formular algoritmos con refinamiento descendente paso a paso: Estudio del caso 2 (repetición controlada por centinela)

Generalicemos el problema de promedios de clase: considere el problema siguiente:

Desarrolle un programa de promedios de clase que pueda procesar un número arbitrario de calificaciones, cada vez que se ejecute el programa.

En el primer ejemplo de promedio de clase, se sabía por anticipado el número de calificaciones (10). En este ejemplo, no se da ninguna indicación de cuantas calificaciones se tomarán. El programa debe ser capaz de procesar un número arbitrario de calificaciones. ¿Cómo podrá el programa determinar cuándo parar la captura de calificaciones? ¿Cuándo sabrá que debe calcular e imprimir el promedio de clase?

Una forma de resolver este problema es utilizar un valor especial llamado un *valor centinela* (también conocido como *valor señal,* un *valor substituto,* o un *valor bandera*) que indicará "fin de la captura de datos". El usuario escribirá calificaciones hasta que haya capturado todas las calificaciones legítimas. Entonces escribirá un valor centinela, a fin de indicar que ha sido introducida la última calificación. La repetición controlada por centinela a menudo se llama *repetición indefinida,* porque antes de que se empiece a ejecutar el ciclo el número de repetición no es conocido.

Claramente, el valor centinela deberá ser seleccionado de tal forma que no se confunda con algún valor de entrada aceptable. Dado que normalmente las calificaciones de un examen son enteros no negativos, para este problema, -1 resulta un valor centinela aceptable. Entonces, una ejecución del programa de promedios de clase pudiera procesar un flujo de entradas como 95, 96, 75, 74, 89, y -1. El programa a continuación calcularía e imprimiría el promedio de clase para las calificaciones 95, 96, 75, 74, y 89 (-1 es el valor centinela y, por lo tanto, no debe entrar en el cálculo de promedio).

Error común de programación 3.6

Seleccionar un valor centinela que también pudiera resultar un valor legítimo de datos.

Enfocamos el programa de promedios de clase con una técnica conocida como *refinación descendente paso a paso,* técnica que resulta esencial al desarrollo de los programas bien estructurados. Empezamos con una representación en seudocódigo de lo más *general*:

Determinar el promedio de clase correspondiente al examen.

Lo más general es un enunciado que representa la función general del programa. Como tal, lo más general es en efecto, una representación completa de todo el programa. Desafortunadamente, lo más general (como en este caso) rara vez transmite una cantidad suficiente de detalle a partir del cual se pueda escribir un programa en C.

Por lo cual empezamos con el proceso de refinación. Dividimos lo más general en una serie de tareas más pequeñas y las enlistamos en el orden en el cual deberán ser ejecutadas. Esto da como resultado el siguiente primer refinamiento:

Inicializar variables
Captura, suma y cuenta de las calificaciones del examen
Calcular e imprimir el promedio de clase

Aquí, sólo se ha utilizado la estructura de secuencia —los pasos enlistados deberán ser ejecutados en orden, uno después del otro.

Observación de ingeniería de software 3.3

Cada refinamiento, al igual que la misma parte más general, debe ser una especificación completa del algoritmo; cambiando sólo el nivel de detalle.

Para seguir al siguiente nivel de refinamiento, es decir, el *segundo nivel de refinamiento,* nos comprometeremos a variables específicas. Necesitamos un total acumulado de los números, una cuenta de cuántos números han sido procesados, una variable para recibir el valor de cada calificación conforme es introducida, y una variable que contenga el promedio calculado. El enunciado en seudocódigo

Inicializar variables

pudiera ser refinado como sigue:

Inicializar total a cero
Inicializar contador a cero

Note que sólo el total y el contador deben ser inicializados; las variables promedio y calificación (para el promedio calculado y la captura del usuario, respectivamente) no necesitan ser inicializados, porque sus valores se escribirán mediante el proceso de lectura destructiva, analizado en el capítulo 2. El enunciado en seudocódigo

Entrada, suma y conteo de las calificaciones del examen

requiere de una estructura de repetición (es decir, de un ciclo), que introduzca sucesivamente cada calificación. Dado que no sabemos por anticipado cuantas calificaciones serán procesadas, utilizaremos la repetición controlada por centinela. El usuario escribirá grados legítimos uno por uno. Una vez que haya escrito el último grado legítimo, el usuario escribirá el valor centinela. Después de haber capturado cada calificación el programa hará una prueba buscando este valor, y dará por terminado el ciclo, cuando el centinela haya sido escrito. La refinación del enunciado precedente en seudocódigo, resulta entonces

Escriba la primera calificación
(While) en tanto el usuario no haya introducido todavía el centinela
    Añada esta calificación al total acumulado
    Añada uno al contador de calificaciones
    Introduzca la siguiente calificación (posiblemente el centinela)

Note que en seudocódigo, para formar el cuerpo de la estructura *while,* no utilizamos llaves alrededor de un conjunto de enunciados. Simplemente hacemos una sangría de todos estos enunciados bajo *while,* para mostrar que todos ellos corresponden al *while.* Repetimos, el seudocódigo es sólo una ayuda informal de desarrollo de programas.

El enunciado en seudocódigo

Calcular e imprimir el promedio de clase

puede ser refinado como sigue:

(If) si el contador no es igual a cero
    Haga que el promedio sea igual al total dividido por el contador
    Imprima el promedio
(else) de lo contrario
    Imprima "No se han escrito calificaciones"

Note que estamos teniendo cuidado aquí de probar la posibilidad de división entre cero un *error fatal* que, si no es detectado causaría que el programa fallara (a menudo llamado "*colapso del programa*"). El segundo refinamiento completo se muestra en la figura 3.7.


```
Initialize total to zero
initialize counter to zero

Input the first grade
While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)

If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
else
    Print "No grades were entered"
```

Fig. 3.7 Algoritmo en seudocódigo, que utiliza repetición controlada por centinela para resolver el problema de promedio de clase.

Error común de programación 3.7

Cualquier intento de dividir entre cero causará un error fatal.

Práctica sana de programación 3.7

Al ejecutar una división por una expresión cuyo valor pudiera ser cero, pruebe de forma explícita este caso y manéjelo apropiadamente en su programa (como sería impresión de un mensaje de error), en vez de permitir que ocurra un error fatal.

En las figuras 3.5 y 3.7, incluimos dentro del seudocódigo algunas líneas totalmente en blanco para mayor legibilidad. De hecho, las líneas en blanco separan estos programas en sus diferentes fases.

Observación de ingeniería de software 3.4

Muchos programas pueden ser divididos lógicamente en tres fases: una fase de inicialización, que inicializa las variables del programa; una fase de proceso, que captura valores de datos y ajusta las variables de programa correspondientemente; y una fase de terminación, que calcula e imprime los resultados finales.

El algoritmo en seudocódigo de la figura 3.7 resuelve el problema más general de promedio de clase. Este algoritmo fue desarrollado después de sólo dos niveles de refinamiento. A veces se requieren de más niveles.

Observación de ingeniería de software 3.5

El programador termina el proceso de refinamiento descendente paso a paso cuando el algoritmo en seudocódigo queda especificado con suficiente detalle para que el programador pueda convertirlo de seudocódigo a C. A partir de ahí la implantación del programa C es por lo regular sencilla.

En la figura 3.8 aparecen el programa C y una ejecución de muestra. A pesar de que sólo se han escrito calificaciones en enteros, el cálculo de promedio probable producirá un número decimal con un punto decimal. El tipo `int` no puede representar tal número. Para manejar números con puntos decimales el programa introduce el tipo de datos `float` (también conocidos como *números de punto flotante*) así como introduce un operador especial conocido como un *operador*

```
/* Class average program with
   sentinel-controlled repetition */
#include <stdio.h>

main()
{
   float average;              /* new data type */
   int counter, grade, total;

   /* initialization phase */
   total = 0;
   counter = 0;

   /* processing phase */
   printf("Enter grade, -1 to end: ");
   scanf("%d", &grade);

   while (grade != -1) {
      total = total + grade;
      counter = counter + 1;
      printf("Enter grade, -1 to end: ");
      scanf("%d", &grade);
   }

   /* termination phase */
   if (counter != 0) {
      average = (float) total / counter;
      printf("Class average is %.2f", average);
   }
   else
      printf("No grades were entered\n");

   return 0;   /* indicate program ended successfully */
}
```

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

Fig. 3.8 Programa en C y ejecución de muestra para el programa de promedio de clase mediante repetición controlada por centinela.

"cast" para manejar el cálculo de promedio. Estas características se explican en detalle después de presentar el programa.

Note el enunciado compuesto en el ciclo **while** de la figura 3.8. De nuevo, para que se ejecuten los cuatro enunciados dentro del ciclo son necesarias las llaves. Sin las llaves, los últimos

tres enunciados en el cuerpo del ciclo quedarían fuera del mismo, haciendo que la computadora interpretase este código en forma incorrecta, como sigue:

```
While(grade != -1)
   total = total + grade;
counter = counter + 1;
printf("Enter grade, -1 to end: ");
     scanf("%f", &grade);
```

Esto causaría un ciclo infinito, en tanto el usuario no introdujera -1 para la primera calificación.

Práctica sana de programación 3.8

En un ciclo controlado por centinela, las indicaciones solicitando la introducción de datos, deberían recordar de forma explícita al usuario cual es el valor centinela.

Los promedios no siempre resultan en valores enteros. A menudo, un promedio es un valor como 7.2 o -93.5, conteniendo una parte fraccionaria. Estos valores se conocen como números de punto flotante, y se representan por el tipo de datos **float**. La variable **average** se declara del tipo **float**, a fin de capturar el resultado fraccionario de nuestro cálculo. Sin embargo, el resultado del cálculo **total / counter** es un entero, porque tanto **total** como **counter** son ambas variables enteras. La división de dos enteros resulta en una *división de enteros,* en la cual se pierde la parte fraccionaria del cálculo (es decir, *se trunca*). Dado que este cálculo se ejecuta primero, se pierde la parte fraccionaria, antes de que el resultado sea asignado a **average**. A fin de producir un cálculo de punto flotante con valores enteros, debemos crear valores temporales que sean números de punto flotante para el cálculo. C proporciona un *operador cast unario* para esta tarea. El enunciado

```
average = (float) total / counter;
```

Incluye el operador cast (**float**) que crea una copia temporal de punto flotante de su propio operando, **total**. El uso de un operador cast de esta forma se conoce como *conversión explícita*. El valor almacenado en **total** sigue siendo un entero. El cálculo ahora consiste en un valor de punto flotante (la versión temporal **float** de **total**) dividida por el valor entero almacenado en **counter**. El compilador de C sólo sabe como evaluar expresiones en donde los tipos de datos de los operandos sean idénticos. A fin de asegurarse que los operandos sean del mismo tipo, el compilador lleva a cabo una operación denominada *promoción* (también conocida como *conversión implícita*) sobre los operandos seleccionados. Por ejemplo, en una expresión que contenga los tipos de datos **int** y **float**, el estándar ANSI especifica que se hacen copias de los operandos **int** y se *promueven* a **float**. En nuestro ejemplo, después de hacer una copia de **counter** y promoverla a **float**, se lleva a cabo el cálculo y el resultado de la división de punto flotante se asigna a **average**. El estándar ANSI incluye un conjunto de reglas para la promoción de operandos de diferentes tipos. En el capítulo 5 se presenta un análisis de todos los tipos estándar de datos y su orden de promoción.

Los operadores cast están disponibles para cualquier tipo de datos. El operador cast se forma colocando paréntesis alrededor del nombre de un tipo de datos. El operador cast es un *operador unario*, es decir, un operador que útiliza un operando. En el capítulo 2, estudiamos los operadores aritméticos binarios. C también acepta versiones unarias de los operadores más (**+**) y menos (**-**), de tal forma que el programador puede escribir expresiones como **-7** o **+5**. Los operadores cast se asocian de derecha a izquierda, y tienen la misma precedencia que los otros operadores unarios,

como el unario + y el unario -. Esta precedencia es de un nivel superior al de los *operadores multiplicativos* *, /, y %, y de un nivel menor que el de los paréntesis.

El programa de la figura 3.8 utiliza un especificador de conversión para **printf** igual a **%.2f** para la impresión del valor de **average**. La **f** indica que deberá ser impreso un valor de punto flotante. El **.2** es la *precisión* con la cual dicho valor se deberá desplegar. Indica que el valor será desplegado con 2 decimales a la derecha del punto decimal. Si se utiliza el especificador de conversión **%f** (sin especificar precisión), se utilizará la *precisión preestablecida* por omisión de 6 exactamente igual a si se utilizara un especificador de conversión **%.6f**. Cuando los valores de punto flotante se imprimen con precisión, el valor impreso se *redondea* al número indicado de posiciones decimales. El valor en memoria se conserva sin alteración. Cuando son ejecutados los enunciados siguientes, se imprimen los valores 3.45 y 3.4

```
printf("%.2f\n", 3.446);   /* prints 3.45 */
printf("%.1f\n", 3.446);   /* prints 3.4  */
```

Error común de programación 3.8

Es incorrecto utilizar precisión en una especificación de conversión en la cadena de control de formato de un enunciado **scanf**. *Las precisiones se utilizan sólo en la especificación de conversión de* **printf**.

Error común de programación 3.9

Utilizar números de punto flotante, de tal forma de que se suponga que están representados con precisión, puede llevar a resultados incorrectos. Los números de punto flotante son representados sólo en forma aproximada por la mayor parte de las computadoras.

Práctica sana de programación 3.9

No compare valores de punto flotante buscando igualdad.

A pesar del hecho de que los números de punto flotante no son siempre "100% precisos", tienen muchas aplicaciones. Por ejemplo, cuando hablamos de una temperatura corporal "normal" de 98.6°F(37°C), no es necesario ser preciso hasta un gran número de dígitos. Cuando vemos la temperatura en un termómetro y la leemos como 98.6, pudiera en realidad ser 98.5999473210643. El punto aquí es que el expresar este número como 98.6 es suficiente para la mayor parte de las aplicaciones. Diremos más sobre este tema posteriormente.

Otra forma en que se desarrollan los números de punto flotante es mediante la división. Cuando dividimos 10 entre 3, el resultado es 3.3333333...con la secuencia de 3 repitiéndose en forma infinita. La computadora asignará cierta cantidad de espacio para contener un valor como éste, por lo que claramente el valor de punto flotante almacenado sólo puede ser una aproximación.

3.10 Cómo formular algoritmos con refinamiento descendente paso a paso: Estudio de caso 3 (estructuras de control anidadas)

Trabajemos otro problema completo. Volveremos a formular el algoritmo utilizando seudocódigo y refinamiento descendente paso a paso, y escribiremos el programa en C correspondiente. Hemos visto que las estructuras de control pueden ser apiladas una encima de otra (en secuencia), de la misma forma que un niño apila bloques. En este estudio de caso veremos la única otra forma estructurada que en C pueden conectarse estructuras de control, es decir mediante el *anidar* una estructura de control dentro de otra.

Considere el siguiente enunciado de problema:

Una universidad ofrece un curso que prepara alumnos para el examen estatal de licenciatura para corredores de bienes raíces. El año pasado, varios de los alumnos que terminaron este curso hicieron el examen de licenciatura. Naturalmente, la universidad desea saber qué tan bien salieron sus alumnos en el examen. Se le ha pedido a usted que escriba un programa para resumir los resultados. Se le ha dado una lista de estos diez alumnos. A continuación de cada nombre se ha escrito un 1 si el alumno pasó el examen y un 2 si no lo pasó.

Su programa deberá analizar los resultados del examen, como sigue:

1. Introducir cada resultado de prueba (es decir, 1 o 2). Desplegar en pantalla el mensaje "introducir resultado" cada vez que el programa solicite otro resultado de prueba.
2. Contar el número de resultados de prueba de cada tipo.
3. Desplegar un resumen de los resultados de prueba, indicando el número de alumnos que pasaron y el número de alumnos que reprobaron.
4. Si más de 8 alumnos pasaron el examen, imprima el mensaje "aumente la colegiatura".

Después de leer de forma cuidadosa el enunciado del problema, hacemos las siguientes observaciones:

1. El programa debe procesar 10 resultados de prueba. Se utilizará un ciclo controlado por contador.
2. Cada resultado de prueba es un número ya sea 1 ó 2. Cada vez que el programa lea un resultado de prueba, el programa debe determinar si el número es 1 ó 2. Probaremos buscando un 1 en nuestro algoritmo. Si el número no es un 1, supondremos que se trata de 2. (Un ejercicio al final de este capítulo analiza las consecuencias de esta suposición).
3. Se utilizarán dos contadores uno para contar el número de estudiantes o alumnos que pasaron el examen y uno para contar el número de alumnos que reprobaron el examen.
4. Después que el programa haya procesado todos los resultados, debe decidir si más de 8 alumnos pasaron el examen.

Procedamos con refinamiento descendente paso a paso. Empezamos con una representación general en seudocódigo:

Analice los resultados de examen y decida si debe aumentarse la colegiatura

De nuevo, es importante enfatizar que lo general es una representación completa del programa, pero es probable que se requerirán varios refinamientos, antes de que el seudocódigo pueda ser convertido naturalmente en un programa C. Nuestro primer refinamiento es:

Inicializar variables
Introduzca las diez calificaciones de examen, cuente los aprobados y los reprobados
Imprima un resumen de los resultados de examen, decida si debe aumentarse la colegiatura

Aquí, también, aunque tenemos una representación completa de la totalidad del programa, se requiere de aún más refinamiento. Debemos ahora fijar ciertas variables específicas. Se requieren de contadores para registrar los aprobados y los reprobados, un contador se utilizará para controlar el proceso del ciclo, y se necesita una variable para almacenar la entrada del usuario. El enunciado en seudocódigo.

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

Inicializar variables

puede ser refinado como sigue:

Inicializar aprobados a cero
Inicializar reprobados a cero
Inicializar alumnos a uno

Note que sólo los contadores y los totales se inicializan . El enunciado en seudocódigo

Introduzca las diez calificaciones de examen, cuente los aprobados y los reprobados

requiere de un ciclo que introduzca en lo sucesivo el resultado de cada examen. Aquí se sabe por anticipado que existirán precisamente diez resultados de exámenes, por lo que es apropiado utilizar un ciclo controlado por contador. Dentro del ciclo (es decir, *anidado* dentro del ciclo) una estructura de doble selección determinará si cada resultado de examen es un aprobado o un reprobado, e incrementará de forma correspondiente los contadores apropiados. Por lo tanto, el refinamiento del enunciado precedente en seudocódigo es

While En tanto el contador de alumnos sea menor o igual a diez
Introduzca el resultado del siguiente examen
If Si el alumno aprobó
Añada uno a aprobados
else De lo contrario
Añada uno a reprobados
Añada uno a contador de alumnos

Note la utilización de líneas en blanco para destacar la estructura de control *if/else*, a fin de mejorar la legibilidad del programa. El enunciado en seudocódigo

Imprima un resumen de los resultados de examen y decida si debe aumentarse la colegiatura

puede ser refinado como sigue:

Imprima el número de aprobados
Imprima el número de reprobados
Si ocho o más estudiantes aprobaron
Imprima "Aumentar colegiatura"

La segunda refinación completa aparece en la figura 3.9. Note que también se utilizan líneas en blanco para destacar la estructura *while* mejorando la legibilidad del programa.

Este seudocódigo está lo suficiente refinado para su conversión a C. El programa C y dos ejecuciones de muestra aparecen en la figura 3.10. Note que se ha aprovechado una característica de C, que permite que la inicialización sea incorporada en las declaraciones. Esta inicialización ocurre en tiempo de compilación.

Sugerencia de rendimiento 3.1

La inicialización de variables en el mismo momento que son declaradas reducen el tiempo de ejecución de un programa.

Observación de ingeniería de software 3.6

La experiencia ha mostrado que la parte más difícil para resolver un programa en una computadora es desarrollar el algoritmo de su solución. Una vez que se haya especificado el algoritmo correcto, el proceso de producir un programa C operacional por lo regular resulta simple.

Initialize passes to zero
Initialize failures to zero
Initialize student to one

While student counter is less than or equal to ten
Input the next exam result

If the student passed
Add one to passes
else
Add one to failures

Add one to student counter

Print the number of passes
Print the number of failures
If eight or more students passed
Print "Raise tuition"

Fig. 3.9 Seudocódigo para el problema de resultados del examen.

Observación de ingeniería de software 3.7

Muchos programadores escriben programas sin jamás utilizar herramientas de desarrollo de programas, como es el seudocódigo. Sienten que su meta final es resolver el problema sobre una computadora, y que la escritura de seudocódigo sólo retarda la producción de los resultados finales.

3.11 Operadores de asignación

C dispone de varios operadores de asignación para la abreviatura de las expresiones de asignación. Por ejemplo, el enunciado

```
c = c + 3;
```

puede ser abreviado utilizando el *operador de asignación* `+=` como

```
c += 3;
```

El operador `+=` añade el valor de la expresión, a la derecha del operador, al valor de la variable a la izquierda del operador, y almacena el resultado en la variable a la izquierda del operador. Cualquier enunciado de la forma

variable = variable operador expresión;

donde *operador* es uno de los operadores binarios `+`, `-`, `*`, `/` o `%` (u otros que discutiremos en el capítulo 10), pueden ser escritos de la forma

variable operador= expresión;

```
/* Analysis of examination results */
#include <stdio.h>

main()
{
   /* initializing variables in declarations */
   int passes = 0, failures = 0, student = 1, result;

   /* process 10 students; counter-controlled loop */
   while (student <= 10) {
      printf("Enter result (1=pass,2=fail): ");
      scanf("%d", &result);

      if (result == 1)          /* if/else nested in while */
         passes = passes + 1;
      else
         failures = failures + 1;

      student = student + 1;
   }

   printf("Passed %d\n", passes);
   printf("Failed %d\n", failures);

   if (passes > 8)
      printf("Raise tuition\n");

   return 0;    /* successful termination */
}
```

```
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=paSS,2=fail): 1
Enter Result (1=pass,2=fail): 2
Passed 6
Failed 4
```

Fig. 3.10 Programa en C y ejecuciones de muestra para el problema de resultados de examen (parte 1 de 2).

Por lo tanto la asignación `c += 3` añade `3` a `c`. En la figura 3.11 aparecen los operadores de asignación aritméticos, con expresiones de muestra utilizando estos operadores y con explicaciones.

Sugerencia de rendimiento 3.2

Una expresión con un operador de asignación (como en `c += 3`) se compila más aprisa que la expresión equivalente expandida (`c = c + 3`) porque en la primera expresión `c` se evalúa únicamente una vez, en tanto que en la segunda se evalúa dos veces.


```
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Passed 9
Failed 1
Raise tuition
```

Fig. 3.10 Programa C y ejecuciones de muestra para el problema de resultados de examen (parte 2 de 2).

Operador de asignación	Expresión de muestra	Explicación	Asignación
Assume : `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to `c`
`-=`	`d -= 4`	`d = d - 4`	1 to `d`
`*=`	`e *= 5`	`e = e * 5`	20 to `e`
`/=`	`f /= 3`	`f = f / 3`	2 to `f`
`%=`	`g %= 9`	`g = g % 9`	3 to `g`

Fig. 3.11 Operadores de asignación aritméticos.

Sugerencia de rendimiento 3.3

Muchas de las sugerencias de rendimiento que mencionamos en este texto dan como resultado mejoras nominales, por lo que el lector pudiera estar tentado a ignorarlos. El hecho es que el efecto acumulativo de todas estas mejoras de rendimiento pueden hacer que un programa ejecute en forma significativamente más rápida. Además, se obtiene una mejora significativa cuando una mejora, supuestamente sólo nominal, es introducida en un ciclo que pudiera repetirse un gran número de veces.

3.12 Operadores incrementales y decrementales

C también tiene el *operador incremental* unario, `++`, y el *operador decremental* unario `--`, que se resumen en la figura 3.12. Si una variable `c` es incrementada en 1, el operador incremental `++` puede ser utilizado en vez de las expresiones `c = c + 1` o bien `c += 1`. Si los operadores incrementales o decrementales son colocados antes de una variable, se conocen como los *operadores de preincremento* o de *predecremento*, respectivamente. Si los operadores incremen-

Operador	Expresión de muestra	Explicación
++	++a	Se incrementa **a** en 1 y a continuación se utiliza el nuevo valor de **a** en la expresión en la cual resida **a**.
++	a++	Utiliza el valor actual de **a** en la expresión en la cual reside **a**, y después se incrementa **a** en 1
--	--b	Se decrementa **b** en 1 y a continuación se utiliza el nuevo valor de **b** en la expresión en la cual reside **b**.
--	b--	Se utiliza el valor actual de **b** en la expresión en la cual reside **b**, y después se decrementa a **b** en 1.

Fig. 3.12 Los operadores incrementales y decrementales.

tales o decrementales se colocan después de una variable, se conocen como los *operadores de postincremento* o de *postdecremento*, respectivamente. El preincrementar (o predecrementar) una variable hace que la variable primero se incremente (o decremente) en 1, y a continuación el nuevo valor de la variable se utilizará en la expresión en la cual aparece. Si se postincrementa (o postdecrementa) la variable hace que el valor actual de la variable se utilice en la expresión en la cual aparece, y a continuación el valor de la variable se incrementará (o decrementará) en 1.

El programa de la figura 3.13 demuestra la diferencia entre las versiones de preincremento y de postincremento del operador ++. Postincrementar la variable c hará que se incremente, después de su uso en el enunciado **printf**. Preincrementar la variable c hará que se incremente, antes de su uso en el enunciado **printf**.

El programa despliega el valor de c antes y después de haber utilizado el operador ++. El operador decremental (--) funciona en forma similar.

Práctica sana de programación 3.10

Los operadores unarios deben ser colocados de forma directa al lado de sus operandos, sin espacios intermedios.

Los tres enunciados de asignación de la figura 3.10

```
passes = passes + 1;
failures = failures + 1;
student = student + 1;
```

pueden ser escritos de manera más concisamente con operadores de asignación, como

```
passes += 1;
failures += 1;
student += 1;
```

con operadores de preincremento como sigue

```
++passes;
++failures;
++student;
```



```
/* Preincrementing and postincrementing */
#include <stdio.h>

main()
{
   int c;

   c = 5;
   printf("%d\n", c);
   printf("%d\n", c++);        /* postincrement */
   printf("%d\n\n", c);

   c = 5;
   printf("%d\n", c);
   printf("%d\n", ++c);        /* preincrement */
   printf("%d\n", c);

   return 0;   /* successful termination */
}
```

```
5
5
6

5
6
6
```

Fig. 3.13 Muestra de la diferencia entre preincrementar y postincrementar.

o con operadores postincrementales, como sigue

```
passes++;
failures++;
student++;
```

Es importante hacer notar que al incrementar o decrementar una variable en un enunciado por sí mismo, las formas de preincremento y postincremento tienen el mismo efecto. Es sólo cuando la variable aparece en el contexto de una expresión más grande que el preincrementar y postincrementar tienen efectos distintos (y similar para predecrementar y postdecrementar).

Sólo se puede utilizar un nombre simple de variable como operando de un operador incremental o decremental.

Error común de programación 3.10

Intentar utilizar el operador incremental o decremental en una expresión distinta a la de un nombre simple de variable, es decir escribiendo ++(x+1) es un error de sintaxis.

Práctica sana de programación 3.11

Por lo regular el estándar ANSI no especifica el orden en el cual serán evaluados los operandos de un operador (aunque veremos excepciones a lo anterior en el caso de unos cuantos operadores en el capítulo 4). Por lo tanto, el programador deberá de evitar utilizar enunciados con operadores incrementales o decrementales, en los cuales una variable particular sujeta a incremento o decremento aparezca más de una vez.

La tabla en la figura 3.14 muestra la precedencia y asociatividad de los operadores presentados hasta este punto. Los operadores se muestran de arriba a abajo en orden decreciente de precedencia. La segunda columna describe la asociatividad de los operadores en cada uno de los niveles de precedencia. Note que el operador condicional (`?:`), los operadores unarios e incremental (`++`), decremental (`--`), más (`+`), menos (`-`) y los cast, así como los operadores de asignación `=`, `+=`, `-=`, `*=`, `/=` y `%=` se asocian de derecha a izquierda. La tercera columna identifica los diferentes grupos de operadores. Todos los demás operadores de la figura 3.14 se asocian de izquierda a derecha.

Resumen

- La solución a cualquier problema de cómputo involucra la ejecución de una serie de acciones en un orden específico. Un procedimiento para resolver un problema en términos de las acciones a ejecutarse y del orden en el cual estas acciones deben de ejecutarse se conoce como un algoritmo.
- La especificación en un programa de cómputo del orden en el cual los enunciados deben ser ejecutados, se conoce como control de programa.
- El seudocódigo es un lenguaje artificial e informal, que auxilia a los programadores en el desarrollo de algoritmos. Es similar al inglés coloquial. De hecho los programas en seudocódigo no se pueden ejecutar en las computadoras. Más bien, el seudocódigo tan sólo ayuda al programador a "pensar" un programa, antes de intentar su escritura en un lenguaje de programación, como es C.
- El seudocódigo está formado sólo por caracteres, por lo que los programadores pueden escribir los programas en seudocódigo en la computadora, editarlos y guardarlos.

Operadores	Asociatividad	Tipo
`()`	de izquierda a derecha	paréntesis
`++ -- + - (tipo)`	de derecha a izquierda	unario
`* / %`	de izquierda a derecha	multiplicativo
`+ -`	de izquierda a derecha	aditivo
`< <= > >=`	de izquierda a derecha	relacional
`== !=`	de izquierda a derecha	igualdad
`?:`	derecha a izquierda	condicional
`= += -= *= /= %=`	derecha a izquierda	de asignación

Fig. 3.14 Precedencia de los operadores utilizados hasta ahora en el texto.

- El seudocódigo comprende solamente enunciados ejecutables. Las declaraciones son mensajes al compilador, indicándole los atributos de variables, y diciéndole que reserve espacio para las mismas.
- Una estructura de selección se utiliza para elegir entre cursos alternativos de acción.
- La estructura de selección **`if`** ejecuta una acción indicada, sólo cuando la condición es verdadera.
- La estructura de selección **`if/else`** define acciones diferentes a ejecutarse cuando la condición es verdadera, y cuando la condición es falsa.
- Una estructura de selección anidada **`if/else`** puede probar muchos casos diferentes. Si más de una condición es verdadera, sólo serán ejecutados los enunciados después de la primera condición verdadera.
- Siempre que deba ejecutarse más de un enunciado, donde por lo regular un enunciado se espera, estos enunciados deberán ser colocados entre llaves, para formar un enunciado compuesto. Un enunciado compuesto puede ser colocado en cualquier sitio donde pueda colocarse un enunciado simple.
- Un enunciado vacío, que indica que no debe tomarse ninguna acción, se identifica colocando un punto y coma (`;`) donde debería estar el enunciado.
- Una estructura de repetición define que una acción deberá ser repetida, en tanto cierta condición siga siendo verdadera.
- El formato para la estructura de repetición **`while`** es

```
while (condición)
   enunciado
```

- El enunciado (o enunciado compuesto o bloque), contenido en la estructura de repetición **`while`** constituye el cuerpo del ciclo.
- Por lo regular, alguna acción especificada dentro del cuerpo de un **`while`** eventualmente debe hacer que la condición se convierta en falsa. De lo contrario, el ciclo no terminará nunca —un error conocido como ciclo infinito.
- Los ciclos controlados por contador utilizan una variable como contador, para determinar cuando debe terminar el ciclo.
- Un total es una variable que acumula la suma de una serie de números. Por lo regular, los totales deberán ser inicializados a cero, antes de ejecutar un programa.
- Un diagrama de flujo es una representación gráfica de un algoritmo. Los diagramas de flujo se dibujan utilizando ciertos símbolos especiales como son óvalos, rectángulos, diamantes y pequeños círculos conectados por flechas, llamadas líneas de flujo. Los símbolos indican acciones a ejecutarse. Las líneas de flujo indican el orden en el cual se deberán ejecutar las acciones.
- El símbolo oval, también conocido como símbolo de terminación, indica el principio y el final de todo algoritmo.
- El símbolo rectángulo, también conocido como símbolo de acción, indica cualquier tipo de cálculo, o de operación de entrada/salida. Los símbolos rectángulo corresponden a las acciones ejecutadas por enunciados de asignación, o por operaciones de entrada/salida, llevadas a cabo casi siempre por funciones estándar de biblioteca como **`printf`** y **`scanf`**.

- El símbolo diamante, también conocido como símbolo de decisión, indica que se debe tomar una decisión. El símbolo de decisión contiene una expresión, que puede ser o verdadera o falsa. Dos líneas de flujo parten de este símbolo. Una de estas líneas de flujo indica la dirección a tomar cuando la condición es verdadera; la otra indica la dirección a tomar cuando la condición es falsa.
- Un valor que contiene una parte fraccionaria es conocido como un número de punto flotante, y se representa por el tipo de datos **`float`**.
- La división de dos enteros resulta en una división de enteros, en el cual se perderá (es decir, quedará truncada) cualquier parte fraccionaria resultante del cálculo.
- C dispone de un operador unario cast (**`float`**) para crear una copia temporal de punto flotante de su operando. El uso de esta forma de un operador cast, se conoce como conversión explícita. Los operadores cast están disponibles para cualquier tipo de datos.
- El compilador de C sólo sabe evaluar expresiones en las cuales son idénticos los tipos de datos de los operandos. Para asegurarse que los operandos sean del mismo tipo, el compilador ejecuta una operación conocida como promoción (también llamada conversión implícita), sobre los operandos seleccionados. El estándar o norma ANSI especifica que se hagan copias de los operandos **`int`** y se promuevan a **`float`**. El estándar ANSI proporciona un conjunto de reglas para la promoción de operandos de diferentes tipos.
- Los valores de punto flotante son sacados en un enunciado **`printf`** con un número específico de dígitos después del punto decimal, mediante el uso de una precisión en el especificador de conversión **`%f`**. El valor **`3.456`**, salido con el especificador de conversión **`%.2f`** aparecerá desplegado como **`3.46`**. Si se utiliza el especificador de conversión **`%f`** (sin especificar precisión), será utilizada la precisión preestablecida por omisión de 6.
- C contiene varios operadores de asignación, que ayudan a abreviar ciertos tipos comunes de expresiones aritméticas de asignación. Estos operadores son: **`+=`**, **`-=`**, **`*=`**, **`/=`**, y **`%=`**. En general, cualquier enunciado de la forma

  variable = *variable operador expresión*;

  donde **`operador`** es uno de los operadores **`+`**, **`-`**, **`*`**, **`/`**, o **`%`**, puede ser escrito de la forma

  variable operador = *expresión*;

- C proporciona el operador incremental **`++`**, y el operador decremental **`--`**, para incrementar o decrementar una variable en 1. Estos operadores pueden ser prefijos o postfijos a una variable. Si el operador es prefijo a la variable, esta será incrementada o decrementada primero por 1, y a continuación utilizada en su expresión. Si el operador es postfijo a la variable, esta será utilizada en su expresión, y a continuación incrementada o decrementada en 1.

Terminología

acción
símbolo de acción
algoritmo
operadores aritméticos de asignación: **`+=`**, **`-=`**, **`*=`**, **`/=`** y **`%=`**
símbolo de flecha
bloque
cuerpo de un cliclo
"colapso"
operador cast
enunciado compuesto
operador condicional (**`?:`**)
estructura de control
contador
repetición controlada por contador
"choque"
decisión
símbolo de decisión
operador decremental (**`--`**)
precisión preestablecida o por omisión
repetición definida
símbolo diamante
división entre cero
estructura de doble selección
valor sustituto
enunciado vacío (**`;`**)
"entrada de fin de datos"
símbolo de terminación
conversión explícita
error fatal
primera refinación
valor bandera
`float`
número de punto flotante
diagrama de flujo
símbolo de diagrama de flujo
línea de flujo
valor "basura"
eliminación **`goto`**
enunciado **`goto`**
estructura de selección **`if/else`**
estructura de selección **`if`**
conversión implícita
operador incremental (**`++`**)
repetición indefinida
ciclo infinito
inicialización
fase de inicialización
división de enteros
error lógico
ciclar
estructura de múltiple selección
operadores multiplicativos
estructuras de control anidadas
estructuras **`if/else`** anidadas
error no fatal
orden de acciones
símbolo oval
operador postdecremental
operador postincremental
precisión
operador predecremental
operador preincremental
fase de procesamiento
control de programa
promoción
seudocódigo
símbolo rectángulo
repetición
estructuras de repetición
redondeo
segunda refinación
selección
estructura de selección
valor centinela
ejecución secuencial
estructura en secuencia
valor señal
estructuras de control de una entrada/una salida
estructura de una selección
estructuras de control apiladas
pasos
refinación paso a paso
programación estructurada
error de sintaxis
condición de terminación
fase de terminación
símbolo de terminación
operador ternario
general
refinación descendente paso a paso
total
transferencia de control
truncamiento
estructura de repetición **`while`**
caracteres de espacio en blanco

Errores comunes de programación

3.1 Olvidar una o ambas de las llaves que delimitan un enunciado compuesto.
3.2 Colocar en una estructura **`if`** un punto y coma despues de la condición, llevará a un error lógico en estructuras **`if`** de una sola selección y a un error de sintaxis en estructuras **`if`** de doble selección.

3.3 No incluir en el cuerpo de una estructura `while` una acción que haga que la condición existente en `while` eventualmente se convierta en falsa. Por lo regular, esta estructura de repetición no terminará jamás un error conocido como "ciclo infinito".
3.4 Escribir la palabra reservada `while` con una `W` como `While` (recuerde que C es un lenguaje que es sensible a la caja tipográfica). Todas las palabras reservadas de C, como `while`, `if` y `else`, contienen sólo letras minúsculas.
3.5 Si no se inicializa un contador o un total, los resultados de su programa es probable que estarán incorrectos. Esto es un ejemplo de un error lógico.
3.6 Seleccionar un valor centinela que también pudiera resultar un valor legítimo de datos.
3.7 Cualquier intento de dividir entre cero causará un error fatal.
3.8 Es incorrecto utilizar precisión en una especificación de conversión en la cadena de control de formato de un enunciado `scanf`. Las precisiones se utilizan sólo en la especificación de conversión de `printf`.
3.9 Utilizar números de punto flotante, de tal forma de que se suponga que están representados con precisión, puede llevar a resultados incorrectos. Los números de punto flotante son representados sólo en forma aproximada por la mayor parte de las computadoras.
3.10 Intentar utilizar el operador incremental o decremental en una expresión distinta a la de un nombre simple de variable, es decir, escribiendo `++(x+1)` es un error de sintaxis.

Prácticas sanas de programación

3.1 La aplicación consistente de reglas convencionales responsables para las sangrías mejora en forma importante la legibilidad de los programas. Sugerimos un tabulador de tamaño fijo de aproximadamente 1/4 de pulgada o de tres espacios por cada sangría.
3.2 A menudo se utiliza el seudocódigo durante el proceso de diseño para "pensar en voz alta" un programa. Posteriormente el programa en seudocódigo se convierte a C.
3.3 Haga sangrías en ambos cuerpos de los enunciados de una estructura `if/else`.
3.4 Si existen varios niveles de sangría, cada nivel deberá tener sangría con una cantidad igual de espacio.
3.5 Algunos programadores prefieren escribir las llaves de principio y de terminación de los enunciados compuestos antes de escribir en el interior de dichas llaves los enunciados individuales. Esto les ayuda a evitar la omisión de una o ambas de las llaves.
3.6 Inicialice contadores y totales.
3.7 Al ejecutar una división por una expresión cuyo valor pudiera ser cero, pruebe de forma explícita este caso y manéjelo de manera apropiada en su programa (como sería impresión de un mensaje de error), en vez de permitir que ocurra un error fatal.
3.8 En un ciclo controlado por centinela, las indicaciones solicitando la introducción de datos, deberían recordar de forma explícita al usuario cual es el valor centinela.
3.9 No compare valores de punto flotante buscando igualdad.
3.10 Los operadores unarios deben ser colocados en directo al lado de sus operandos, sin espacios intermedios.
3.11 Por lo regular el estándar ANSI no especifica el orden en el cual serán evaluados los operandos de un operador (aunque veremos excepciones a lo anterior en el caso de unos cuantos operadores en el capítulo 4). Por lo tanto, el programador deberá de evitar utilizar enunciados con operadores incrementales o decrementales, en los cuales una variable particular sujeta a incremento o decremento aparezca más de una vez.

Sugerencias de rendimiento

3.1 La inicialización de variables en el mismo momento que son declaradas reducen el tiempo de ejecución de un programa.

3.2 Una expresión con un operador de asignación (como en `c += 3`) se compila más aprisa que la expresión equivalente expandida (`c = c + 3`) porque en la primera expresión *c* se evalúa una vez, en tanto que en la segunda se evalúa dos veces.
3.3 Muchos de las sugerencias de rendimiento que mencionamos en este texto dan como resultado mejoras nominales, por lo que el lector pudiera estar tentado a ignorarlos. El hecho es que el efecto acumulativo de todas estas mejoras de rendimiento pueden hacer que un programa ejecute en forma significativamente más rápida. Además, se obtiene una mejora significativa cuando una mejora, supuestamente solo nominal, es introducida en un ciclo que pudiera repetirse un gran número de veces.

Observaciones de ingeniería de software

3.1 Un enunciado compuesto puede ser colocado en cualquier parte de un programa, donde pueda ser colocado un enunciado sencillo.
3.2 Igual que un enunciado compuesto puede ser colocado en cualquier lugar donde pudiera colocarse un enunciado sencillo, también es posible que no haya ningún enunciado de ningún tipo, es decir, un enunciado vacío. El enunciado vacío se representa colocando un punto y coma (;) donde por lo regular debería estar el enunciado.
3.3 Cada refinamiento, al igual que la misma parte más general, debe ser una especificación completa del algoritmo; cambiando sólo el nivel de detalle.
3.4 Muchos programas pueden ser divididos de forma lógica en tres fases: una fase de inicialización, que inicializa las variables del programa; una fase de proceso, que captura valores de datos y ajusta las variables de programa correspondiente; y una fase de terminación, que calcula e imprime los resultados finales.
3.5 El programador termina el proceso de refinamiento descendente paso a paso cuando el algoritmo en seudocódigo queda especificado con suficiente detalle para que el programador pueda convertirlo de seudocódigo a C. A partir de ahí la implantación del programa C es sencilla.
3.6 La experiencia ha mostrado que la parte más difícil para resolver un programa en una computadora es desarrollar el algoritmo de su solución. Una vez que se haya especificado el algoritmo correcto, el proceso de producir un programa C operacional resulta simple.
3.7 Muchos programadores escriben programas sin jamás utilizar herramientas de desarrollo de programas, como es el seudocódigo. Sienten que su meta final es resolver el problema sobre una computadora, y que la escritura de seudocódigo sólo retarda la producción de los resultados finales.

Ejercicios de autoevaluación

3.1 Responda a cada una de las preguntas siguientes.
a) Un procedimiento para resolver un problema en términos de las acciones a ejecutarse y del orden en el cual deberán dichas acciones ejecutarse, se conoce como un _____.
b) Especificar el orden de ejecución de los enunciados por parte de la computadora, se llama _____.
c) Todos los programas pueden ser escritos en función de tres estructuras de control: _____, _____, y _____.
d) La estructura de selección _____ se utiliza para ejecutar una acción cuando una condición es verdadera, y otra acción cuando la condición es falsa.
e) Varios enunciados agrupados juntos en llaves (`{` y `}`) se conocen como un _____.
f) La estructura de repetición _____ especifica que un enunciado o grupo de enunciados debe ser ejecutado de forma repetidamente, en tanto cierta condición se mantenga verdadera.
g) La repetición de un conjunto de instrucciones un número específico de veces, se conoce como una repetición _____.
h) Cuando no se conoce por anticipado cuantas veces debe repetirse un conjunto de enunciados, se puede utilizar un valor _____ para terminar la repetición.

3.2 Escriba cuatro enunciados diferentes en C, que cada uno de ellos añada uno a la variable entera **x**.

3.3 Escriba un enunciado en C para llevar a cabo cada uno de los siguientes:
a) Asignar la suma de **x** y **y** a **z**, y después del cálculo incrementar el valor de **x** en 1.
b) Multiplicar la variable **product** por 2, utilizando el operador ***=**.
c) Multiplicar la variable **product** por 2, utilizando los operadores **=** y *****.
d) Verificar si el valor de la variable **count** es mayor que 10. Si es así, imprima "**Count is greater than 10**".
e) Decremente la variable **x** en 1 y a continuación réstela de la variable **total**.
f) Añada la variable **x** a la variable **total**, y a continuación decremente **x** en 1.
g) Calcule el residuo, después que **q** se haya dividido por **divisor**, y asigne el resultado a **q**. Escriba este enunciado de dos formas distintas.
h) Imprima el valor **123.4567** con **2** dígitos de precisión ¿Cuál será el valor impreso?
i) Imprima el valor de punto flotante **3.14159** con tres dígitos a la derecha del punto decimal. ¿Cuál será el valor impreso?.

3.4 Escriba un enunciado C, que ejecute cada una de las tareas siguientes:
a) Declare las variables **sum** y **x** del tipo **int**.
b) Inicialice la variable **x** a 1.
c) Inicialice la variable **sum** a **0**.
d) Añada la variable **x** a la variable **sum** y asigne el resultado a la variable **sum**.
e) Imprima "**The sum is: **" seguido por el valor de la variable **sum**.

3.5 Combine los enunciados que escribió en el Ejercicio 3.4 en un programa que calcule la suma de los enteros de 1 a 10. Utilice la estructura **while** para ciclar a través del cálculo y de los enunciados de incremento. El ciclo deberá terminar cuando el valor de **x** se convierta en 11.

3.6 Determinar los valores de cada variable, después de que se haya ejecutado el cálculo. Suponga que cuando empieza cada enunciado, todas las variables tienen el valor 5.
a)
```
product *= x++;
```
b)
```
result = ++x + x;
```

3.7 Escriba enunciados simples en C que:
a) Introduzcan la variable entera **x** mediante **scanf**.
b) Introduzcan la variable entera **y** mediante **scanf**.
c) Inicialicen la variable entera **i** a **1**.
d) Inicialicen la variable entera **power** a **1**.
e) Multipliquen la variable **power** por **x** y asignen el resultado a **power**.
f) Incrementen la variable **y** a **1**.
g) Prueben **y** para ver si es menor o igual a **x**.
h) Saquen la variable entera **power** mediante **printf**.

3.8 Escriba un programa en C que utilice los enunciados del Ejercicio 3.7 para calcular **x** elevada a la potencia **y**. El programa deberá contener una estructura de control de repetición **while**.

3.9 Identifique y corrija los errores en cada uno de los siguientes:
a)
```
while (c  <=  5) {
      product  *= c;
      ++c;
```
b)
```
scanf("%.4F", &value);
```
c)
```
if (gender == 1)
      printf("Woman\n");
else;
      printf("Man\n");
```


3.10 Indique qué es lo que no está bien en la siguiente estructura de repetición **while**:
```
while (z = 0)
   sum += z;
```

Respuestas a los ejercicios de autoevaluación

3.1 a) Algoritmo. b) Control de programa. c) Secuencia, selección, repetición. d) **if/else**. e) Enunciado compuesto. f) **while**. g) Controlado por contador. h) Centinela.

3.2
```
x = x + 1;
x += 1;
++x;
x++;
```

3.3 a)
```
z = x++ + y;
```
b)
```
product *= 2;
```
c)
```
product = product * 2;
```
d)
```
if (count > 10)
      printf("Count is greater than 10.\n");
```
e)
```
total -= --x;
```
f)
```
total += x--;
```
g)
```
q %= divisor;
q = q % divisor;
```
h)
```
printf("%.2f", 123.4567);
```
123.46 is displayed.
i)
```
printf("%.3f\n", 3.14159);
```
3.142 is displayed.

3.4 a)
```
int sum, x;
```
b)
```
x = 1;
```
c)
```
sum = 0;
```
d)
```
sum += x;
```
or
```
sum = sum + x;
```
e)
```
printf("The sum is: %d\n", sum);
```

3.5
```
/* Calculate the sum of the integers from 1 to 10 */
#include <stdio.h>

main()
{
   int sum, x;

   x = 1;
   sum = 0;
   while (x <= 10) {
      sum += x;
      ++x;
   }

   printf("The sum is: %d\n", sum);
}
```

3.6 a)
```
product = 25, x = 6;
```
b)
```
result = 12, x = 6;
```

3.7

a) `scanf("%d", &x);`
b) `scanf("%d", &y);`
c) `i = 1;`
d) `power = 1;`
e) `power *= x;`
f) `y++;`
g) `if(y <= x)`
h) `printf("%d", power);`

3.8

```
/* raise x to the y power */
#include <stdio.h>

main()
   int x,y,i,power;

   i = 1;
   power = 1;
   scanf("%d", &x);
   scanf("%d", &y);

   while (i <= y) {
      power *= x;
      ++i;
   }

   printf("%d", power);
   return 0;
}
```

3.9 a) Error: falta la llave derecha de cierre del cuerpo **`while`**.
Corrección: Añada la llave de cierre derecha, después del enunciado **`++c;`**.
b) Error: precisión utilizada en una especificación de conversión **`scanf`**.
Corrección: Eliminar **`.4`** de la especificación de conversión **`scanf`**.
c) Error: punto y coma después de la parte **`else`** de la estructura **`if/else`** da como resultado un error lógico.
El segundo **`printf`** se ejecutará siempre.
Corrección: eliminar el punto y coma después de **`else`**.

3.10 El valor de la variable **`z`** no se cambia nunca en la estructura **`while`**. Por lo tanto se ha creado un ciclo infinito. Para evitar el ciclo infinito, **`z`** debe ser decrementado, para que de forma eventual se convierta en **`0`**.

Ejercicios

3.11 Identifique y corrija los errores en cada uno de los siguientes (Nota: pudiera haber más de un error en cada segmento de código):

a)
```
if (age >= 65);
   printf("Age is greater than or equal to 65\n");
else
   printf("Age is less than 65\n");
```

b)
```
int x = 1, total;

while (x <= 10) {
   total += x;
   ++x;
}
```


c)
```
While (x <= 100)
   total += x;
   ++x;
```

d)
```
while (y > 0) {
   printf("%d\n", y);
   ++y;
}
```

3.12 Llene los espacios vacíos en cada uno de los siguientes:
a) La solución a cualquier problema involucra la ejecución de una serie de acciones en un _____ específico.
b) Un sinónimo de procedimiento es _____.
c) Una variable que acumula la suma de varios números es un _____.
d) El proceso de definir ciertas variables en valores específicos al principio de un programa, se conoce como _____.
e) Un valor especial utilizado para indicar "entrada de fin de datos" se conoce como un valor _____, _____, _____, o _____.
f) Un _____ es una representación gráfica de un algoritmo.
g) En un diagrama de flujo, el orden en el cual deben ser ejecutados los pasos se indican por los símbolos _____.
h) El símbolo de terminación indica el _____, así como el _____ de todo algoritmo.
i) Los símbolos rectángulo corresponden a cálculos que por lo regular se ejecutan por enunciados _____ y por operaciones de entrada/salida _____ que se llevan normalmente a cabo mediante llamadas a las _____ y _____ de las funciones estándar de biblioteca.
j) El elemento escrito dentro de un símbolo decisión se conoce como una _____.

3.13 ¿Qué es lo que imprime el siguiente programa?

```
#include <stdio.h>

main()
{
   int x = 1, total;

   while (x <= 10) {
      y = x * x;
      printf("%d\n", y);
      total += y;
   }

   printf("Total is %d\n", total);
   return 0;
}
```

3.14 Escriba un enunciado en seudocódigo que indique cada uno de los siguientes:
a) Despliegue el mensaje "**`Enter two numbers`**".
b) Asigne la suma de las variables **`x`**, **`y`** y **`z`** a la variable **`p`**.
c) La condición siguiente debe de ser probada en una estructura de selección **`if/else`**: el valor actual de la variable **`m`** es mayor que dos veces el valor actual de la variable **`v`**.
d) Obtener del teclado valores para las variables **`s`**, **`r`** y **`t`**.

3.15 Formule un algoritmo en seudocódigo para cada uno de los siguientes:
a) Obtenga dos números del teclado, calcule la suma de los números y despliegue el resultado.
b) Obtenga dos números del teclado, y determine y despliegue cuál (si alguno) es el mayor de los dos números.

c) Obtenga una serie de números positivos del teclado, y determine y despliegue la suma de los números. Suponga que el usuario escribe el valor centinela `-1` para indicar "entrada de fin de datos".

3.16 Indique cuál de los siguientes es verdadero y cual es falso. Si el enunciado es falso, explique por qué.

a) La experiencia ha demostrado que la parte más difícil de la resolución de un problema en una computadora es el producir un programa C funcional.
b) Un valor centinela debe ser un valor que no pueda ser confundido con un valor de datos legítimo.
c) Las líneas de flujo indican las acciones a ejecutarse.
d) Las condiciones escritas dentro de los símbolos de decisión siempre contienen operadores aritméticos (es decir `+`, `-`, `*`, `/` y `%`).
e) En la refinación descendente paso a paso, cada refinación es una representación completa del algoritmo.

Para los ejercicios 3.17 a 3.21, lleve a cabo cada uno de estos pasos:

1. Lea el enunciado del problema.
2. Formule el algoritmo utilizando refinación descendente paso a paso.
3. Escriba un programa en C.
4. Pruebe, depure y ejecute el programa en C.

3.17 En razón del alto precio de la gasolina, los conductores están preocupados con el kilometraje que obtienen de sus automóviles. Un conductor ha llevado registro de varios tanques de gasolina, anotando las millas manejadas y los galones utilizados en cada uno de los tanques. Desarrolle un programa en C que introduzca las millas manejadas y los galones utilizados para cada tanque. El programa debe calcular y desplegar las millas por galón obtenidas de cada tanque. Después de procesar toda la información de entrada, el programa deberá calcular e imprimir las millas por galón combinadas, obtenida de todos los tanques.

```
Enter the gallons used (-1 to end): 12.8
Enter the miles driven: 287
The miles / gallon for this tank was 22.421875

Enter the gallons used (-1 to end): 10.3
Enter the miles driven: 200
The miles / gallon for this tank was 19.417475

Enter the gallons used (-1 to end): 5
Enter the miles driven: 120
The miles / gallon for this tank was 24.000000

Enter the gallons used (-1 to end): -1
The overall average miles/gallon was 21.601423
```

3.18 Desarrolle un programa en C que determine si un cliente de una tienda departamental ha excedido el límite de crédito en una cuenta de cargo. Para cada uno de los clientes, están disponibles los siguientes hechos:

1. Número de la cuenta.
2. Saldo al principio del mes.
3. Total de todos los elementos cargados por el cliente este mes.
4. Total de todos los créditos aplicados este mes a la cuenta de este cliente.
5. Límite permitido de crédito.

El programa deberá introducir cada uno de estos hechos, calcular el nuevo saldo (= saldo inicial + cargos - créditos), y determinar si el nuevo saldo excede el límite de crédito del cliente. Para aquellos clientes cuyo límite de crédito esté excedido, el programa deberá desplegar el número de cuenta del cliente, el límite de crédito, el nuevo saldo y el mensaje "límite de crédito excedido".

```
Enter account number (-1 to end): 100
Enter beginning balance: 5394.78
Enter total charges: 1000.00
Enter total credits: 500.00
Enter credit limit: 5500.00
Account:       100
Credit limit: 5500.00
Balance:      5894.78
Credit Limit Exceeded.

Enter account number (-1 to end): 200
Enter beginning balance: 1000.00
Enter total charges: 123.45
Enter total credits: 321.00
Enter credit limit: 1500.00

Enter account number (-1 to end): 300
Enter beginning balance: 500.00
Enter total charges: 274.73
Enter total credits: 100.00
Enter credit limit: 800.00

Enter account number (-1 to end): -1
```

3.19 Una gran empresa química paga a su personal de ventas en base a comisiones. El personal de ventas recibe $200 por semana más 9% de las ventas brutas de esa semana. Por ejemplo, una persona de ventas que vende $5000 de productos químicos en una semana, recibe $200 más 9 % de $5000, o sea un total de $650. Desarrolle un programa en C que introduzca las ventas brutas de cada vendedor correspondiente a la última semana, y calcule y despliegue las ganancias de dicho vendedor. Procese las cifras vendedor por vendedor.

```
Enter sales in dollars (-1 to end): 5000.00
Salary is: $650.00

Enter sales in dollars (-1 to end): 1234.56
Salary is: $311.11

Enter sales in dollars (-1 to end): 1088.89
Salary is: $298.00

Enter sales in dollars (-1 to end): -1
```

3.20 El interés simple de un préstamo se calcula mediante la fórmula

```
interest = principal * rate * days / 365
```

La fórmula anterior supone que `rate` es la tasa de interés anual y, por lo tanto, incluye la división entre 365 (días). Desarrolle un programa en C que introduzca `principal`, `rate` y `days` para varios prestamos diferentes, y que calcule y despliegue el interés simple para cada uno de ellos, mediante el uso de la fórmula anterior.

```
Enter loan principal (-1 to end): 1000.00
Enter interest rate: .1
Enter term of the loan in days: 365
The interest charge is $100.00

Enter loan principal (-1 to end): 1000.00
Enter interest rate: .08375
Enter term of the loan in days: 224
The interest charge is $51.40

Enter loan principal (-1 to end): 10000.00
Enter interest rate: .09
Enter term of the loan in days: 1460
The interest charge is $3600.00

Enter loan principal (-1 to end): -1
```

3.21 Desarrolle un programa en C que determine la nómina bruta para cada uno de varios empleados. La empresa paga "tiempo normal" para las primeras 40 horas trabajadas de cada empleado y paga "tiempo y medio" para todas las horas trabajadas en exceso de 40 horas. Se le proporciona una lista de los empleados de la empresa, el número de horas que cada empleado trabajó la última semana, y la tasa horaria de cada empleado. Su programa deberá introducir esta información para cada uno de los empleados, y determinar y desplegar la nómina bruta de cada uno de ellos.

```
Enter # of hours worked (-1 to end): 39
Enter hourly rate of the worker ($00.00): 10.00
Salary is $390.00

Enter # of hours worked (-1 to end): 40
Enter hourly rate of the worker ($00.00): 10.00
Salary is $400.00

Enter # of hours worked (-1 to end): 41
Enter hourly rate of the worker ($00.00): 10.00
Salary is $415.00

Enter # of hours worked (-1 to end): -1
```

3.22 Escriba un programa en C que demuestre la diferencia entre predecrementar y postdecrementar, utilizando el operador decremental `--`.

3.23 Escriba un programa en C que utilice ciclos para imprimir los números del 1 al 10, lado a lado en el mismo renglón, con tres espacios entre cada uno de ellos.

3.24 El proceso de encontrar el número más grande (es decir, el máximo de un grupo de números), es utilizado con frecuencia en aplicaciones de computación. Por ejemplo, un programa que determina el ganador de un concurso de ventas, introduciría el número de unidades vendidas por cada vendedor. El vendedor que hubiera vendido la mayor cantidad de unidades, ganaría el concurso. Escriba un programa primero en seudocódigo, y a continuación en C, que introduzca una serie de 10 números, y determine e imprima el mayor de los mismos. *Sugerencia:* su programa debería de utilizar tres variables como sigue:

`counter:` Un contador para contar hasta 10 (es decir, para controlar cuántos números han sido introducidos, y para determinar cuándo se han procesado todos los 10 números).
`number:` El número actual introducido al programa.
`largest:` El número más grande encontrado hasta ahora.

3.25 Escriba un programa en C, que utilice ciclos para imprimir la siguiente tabla de valores:

```
N       10*N      100*N     1000*N

1       10        100       1000
2       20        200       2000
3       30        300       3000
4       40        400       4000
5       50        500       5000
6       60        600       6000
7       70        700       7000
8       80        800       8000
9       90        900       9000
10      100       1000      10000
```

El carácter de tabulador, `\t`, puede ser utilizado en un enunciado `printf` para separar las columnas mediante tabuladores.

3.26 Escriba un programa en C que utilice ciclo para producir la siguiente tabla de valores:

```
A       A+2       A+4       A+6

3       5         7         9
6       8         10        12
9       11        13        15
12      14        16        18
15      17        19        21
```

3.27 Utilizando un método similar al del Ejercicio 3.26, encuentre los *dos* valores más grandes de los 10 números. Nota: Sólo puede introducir una vez cada número.

3.28 Modifique el programa de la figura 3.10 para validar sus entradas. En cualquier entrada, si el valor introducido es diferente que uno o dos, siga ciclando hasta que el usuario introduzca un valor correcto.

3.29 ¿Qué es lo que imprime el programa siguiente?

```
#include <stdio.h>

main()
{
   int count = 1;

   while (count <= 10) {
      printf("%s\n", count % 2 ? "****" : "++++++++");
      ++count;
   }

   return 0;
}
```

3.30 ¿Qué es lo que imprime el programa siguiente?

```
#include <stdio.h>

main()
{
   int row = 10, column;

   while (row >= 1) {
      column = 1;

      while (column <= 10) {
         printf("%s", row % 2 ? "<" : ">");
         ++column;
      }

      --row;
      printf("\n");
   }

   return 0;
}
```

3.31 *(Problema del else colgante)*. Determine la salida para cada uno de los siguientes, cuando **x** es **9** y **y** es **11**, y cuando **x** es **11** y **y** es **9**. Advierta que en un programa en C el compilador ignora la sangría. También, el compilador de C siempre asocia un **else** con el **if** anterior, a menos de que se le indique lo contrario mediante la colocación de las llaves **{}**. En vista de que a primera vista, el programador tal vez no estaría seguro de, cuál **if** coincide con qué **else**, esto se conoce como el problema del "else colgante". Hemos eliminado la sangría del código siguiente, para que el problema se haga más interesante. (*Sugerencia:* aplique las reglas de sangría que ha aprendido).

a)
```
if (x < 10)
if (y > 10)
printf("*****\n");
else
printf("#####\n");
printf("$$$$$\n");
```

b)
```
if (x < 10) {
if (y > 10)
printf("*****\n");
}
else {
printf("#####\n");
printf("$$$$$\n");
}
```

3.32 *(Otro problema de else colgante)*. Modifique el código que sigue para producir la salida mostrada. Utilice las técnicas apropiadas de sangrías. No puede hacer ningún cambio a excepción de inserción de llaves. En un programa C el compilador ignorará la sangría. Hemos eliminado las sangrías del código siguiente par hacer más atractivo el problema. Nota: pudiera ser posible que no se requiera de ninguna modificación.


```
if (y == 8)
if (x == 5)
printf("@@@@@\n");
else
printf("#####\n");
printf("$$$$$\n");
printf("&&&&&\n");
```

a) Suponiendo **x** = **5** y **y** = **8**, se produce la siguiente salida.

```
@@@@@
$$$$$
&&&&&
```

b) Suponiendo **x** = **5** y **y** = 8, se produce la siguiente salida.

```
@@@@@
```

c) Suponiendo **x** = **5** y **y** = **8**, se produce la siguiente salida.

```
@@@@@
&&&&&
```

d) Suponiendo **x** = **5** y **y** = **7**, se produce la siguiente salida. Nota: los tres enunciados **printf** últimos son todos parte de un enunciado compuesto.

```
#####
$$$$$
&&&&&
```

3.33 Escriba un programa que lea el lado de un cuadrado y a continuación lo imprima en forma de asteriscos. Su programa deberá poder funcionar para cuadrados de todos tamaños entre 1 y 20. Por ejemplo, si su programa lee un tamaño de 4, debería imprimir

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

3.34 Modifique el programa que escribió en el ejercicio 3.23, de tal forma que imprima un cuadrado hueco. Por ejemplo, si su programa lee un tamaño 5, deberá imprimir

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

3.35 Un palíndromo es un número o una frase de texto, que se lee igual hacia adelante y hacia atrás. Por ejemplo, cada uno de los siguientes enteros de cinco dígitos son palíndromos: 12321, 55555, 45554 y 11611. Escriba un programa que lea un entero de cinco dígitos y que determine si es o no un palíndromo. (*Sugerencia:* utilice los operadores de división y de módulo para separar los números en sus dígitos individuales).

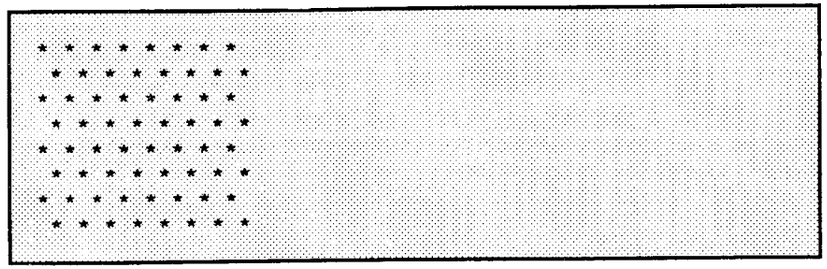
3.36 Introduzca un entero que contenga sólo 0s y 1s (es decir, un entero "binario") e imprima su equivalente decimal. (*Sugerencia:* utilice los operadores de módulo y de división para detectar los dígitos del número "binario" uno por uno, de derecha a izquierda. Al igual que en el sistema numérico decimal, donde el dígito más a la derecha tiene un valor posicional de 1, y el siguiente dígito a la izquierda tiene un valor posicional de 10, y a continuación de 100, y a continuación de 1000, etcétera, en un sistema numérico binario, el dígito más a la derecha tiene un valor posicional de 1, el siguiente dígito a la derecha tiene un valor posicional de 2, y a continuación de 4, y a continuación de 8, etcétera. Por lo tanto, el número decimal 234 puede ser interpretado como 4 * 1 + 3 * 10 + 2 * 100. El equivalente decimal del número 1101 binario es 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8 o bien, 1 + 0 +4 + 8, es decir 13).

3.37 De forma continua escuchamos decir que las computadoras son muy rápidas. ¿Cómo puede usted determinar la verdadera velocidad de operación de su máquina? Escriba un programa con un ciclo **while** que cuente de 1 hasta 3,000,000 de uno en uno. Cada vez que el contador llegue a un múltiplo de 1,000,000, imprima este número en la pantalla. Utilice su reloj para medir cuánto tiempo tarda cada repetición de 1,000,000 del ciclo.

3.38 Escriba un programa que imprima 100 asteriscos, uno por uno. Después de cada décimo asterisco, su programa deberá imprimir un carácter de nueva línea. (*Sugerencia:* cuente de 1 a 100. Utilice un operador de módulo para reconocer cada vez que el contador llegue a un múltiplo de 10.)

3.39 Escriba un programa que lea un entero, y determine e imprima cuántos dígitos de ese entero son 7s.

3.40 Escriba un programa que despliegue el siguiente patrón.



Su programa sólo puede usar tres enunciados `printf`, y uno de la forma

```
printf("* ");
```

otro de la forma

```
printf(" ");
```

y uno de la forma

```
printf("\n");
```

3.41 Escriba un programa que continúe imprimiendo múltiplos del entero 2, es decir, 2, 4, 8, 16, 32, 64, etcétera. Su ciclo no deberá de terminar (es decir, usted debe de crear un ciclo infinito). ¿Qué ocurre cuando ejecuta este programa?

3.42 Escriba un programa que lea el radio de un círculo (como valor `float`) y que calcule e imprima el diámetro, la circunferencia y el área. Utilice para el valor de 3.14159, π.

3.43 ¿Qué es lo que no está bien en el enunciado siguiente?. Vuelva a escribir el enunciado, para que ejecute lo que probablemente el programador intentaba hacer.

```
printf ("%d", ++(x + y));
```

3.44 Escriba un programa que lea tres valores `float` no cero, y que determine e imprima si pueden representar los lados de un triángulo.

3.45 Escriba un programa que lea tres enteros no ceros y que determine e imprima si pueden ser los lados de un triángulo rectángulo.

3.46 Una empresa desea transmitir datos mediante el teléfono, pero están preocupados de que sus teléfonos pudieran estar intervenidos. Todos sus datos se transmiten como enteros de cuatro dígitos. Le han solicitado a usted que escriba un programa que cifre sus datos, de tal forma de que puedan ser transmitidos con mayor segururidad. Su programa debe leer un entero de cuatro dígitos y cifrarlo como sigue: remplazar cada dígito por *(la suma del dígito más 7) módulo 10.* A continuación, intercambiar el primer dígito con el tercero, y el segundo con el cuarto. A continuación imprimir el entero cifrado. Escriba un programa por separado, que introduzca un entero de cuatro dígitos cifrado, y que lo descifre para formar el número original.

3.47 El factorial de un entero no negativo n se escribe como $n!$ (se dice "factorial de n") y se define como sigue:

$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$ (para valores de n mayores que o igual a 1)

y

$n! = 1$ (para $n = 0$).

Por ejemplo, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, lo que significa 120.

a) Escriba un programa que lea un entero no negativo, y que calcule e imprima su factorial.

b) Escriba un programa que estime el valor de la constante matemática e, utilizando la fórmula

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

c) Escriba un programa que calcule el valor de e^x, utilizando la fórmula

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

4

Control de programa

Objetivos

- Ser capaz de utilizar las estructuras de repetición **`for`** y **`do/while`**.
- Comprensión de la selección múltiple utilizando la estructura de selección **`switch`**.
- Ser capaz de utilizar los enunciados de control de programa **`break`** y **`continue`**.
- Ser capaz de utilizar los operadores lógicos.

¿Quién puede controlar su destino?
William Shakespeare
Othello

La llave que siempre se usa está brillante.
Benjamín Franklin

El ser humano es un animal que fabrica herramientas.
Benjamín Franklin

La inteligencia ...es la facultad de poder fabricar objetos artificiales, en especial herramientas para fabricar herramientas.
Henry Bergson.

Sinopsis

4.1 Introducción

Llegado a este punto, el lector debería ya estar cómodo escribiendo programas en C simples pero completos. En este capítulo, se estudia con mayor detalle la repetición, y se presentan estructuras adicionales de control de repetición, es decir, la estructura **`for`** y la estructura **`do/while`**. También se presenta la estructura de selección múltiple **`switch`**. Se analiza el enunciado **`break`**, para salir de inmediato y rápidamente de ciertas estructuras de control, así como el enunciado **`continue`**, para saltarse el resto del cuerpo de una estructura de repetición, continuando con la siguiente iteración del ciclo. El capítulo analiza operadores lógicos utilizados en la combinación de condiciones, y concluye con un resumen de los principios de la programación estructurada, tal y como se presentan en los capítulos 3 y 4.

4.2 Lo esencial de la repetición

La mayor parte de los programas incluyen repeticiones o *ciclos.* Un ciclo es un grupo de instrucciones que la computadora ejecuta en forma repetida, en tanto se conserve verdadera alguna *condición de continuación del ciclo*. Hemos analizado dos procedimientos de repetición:

1. Repetición controlada por contado
2. Repetición controlada por centinela

La repetición controlada por contador se denomina a veces *repetición definida,* porque con anticipación se sabe con exactitud cuántas veces se ejecutará el ciclo. La repetición controlada por centinela a veces se denomina *repetición indefinida,* porque no se sabe con anticipación cuantas veces el ciclo se ejecutará.

En la repetición controlada por contador, se utiliza una *variable de control* para contar el número de repeticiones. La variable de control es incrementada (normalmente en 1), cada vez que se ejecuta el grupo de instrucciones. Cuando el valor de la variable de control indica que se ha ejecutado el número correcto de repeticiones, se termina el ciclo y la computadora continúa ejecutando el enunciado siguiente al de la estructura de repetición.

Los valores centinela se utilizan para controlar la repetición cuando:

1. El número preciso de repeticiones, no es conocido con anticipación, y
2. El ciclo incluye enunciados que deben obtener datos cada vez que éste se ejecuta.

El valor centinela indica "fin de datos". El centinela es introducido una vez que al programa se le han proporcionado todos los elementos normales de datos. Los centinelas deben ser diferentes a los elementos normales de datos.

4.3 Repetición controlada por contador

La repetición controlada por contador requiere:

1. El *nombre* de una variable de control (o contador del ciclo).
2. El *valor inicial* de la variable de control.
3. El *incremento* (o *decremento*) con el cual, cada vez que se termine un ciclo, la variable de control será modificada.
4. La condición que compruebe la existencia del *valor final* de la variable de control (es decir, si se debe o no seguir con el ciclo).

Considere el programa sencillo mostrado en la figura 4.1, que imprime los números de 1 a 10. La declaración

```
int counter = 1
```

le da *nombre* a la variable de control (**`counter`**), la declara como un entero, reserva espacio para la misma, y ajusta su *valor inicial* a **`1`**. Esta declaración no es un enunciado ejecutable.

La declaración e inicialización de **`counter`** también podía haber sido hecha mediante los enunciados

```
int counter;

counter = 1;
```

La declaración no es ejecutable, pero la asignación si lo es. Se utilizarán ambos métodos para inicialización de variables.

El enunciado

```
++counter;
```

```
/* Counter-controlled repetition */
#include <stdio.h>

main()
{
   int counter = 1;              /* initialization */

   while (counter <= 10) {       /* repetition condition */
      printf ("%d\n", counter);
      ++counter;                 /* increment */
   }

   return 0;
}
```

```
1
2
3
4
5
6
7
8
9
10
```

Fig. 4.1 Repetición controlada por contador.

incrementa en 1 el contador del ciclo, cada vez que este ciclo se ejecuta. La condición de continuación del ciclo existente en la estructura **while** prueba si el valor de la variable de control es menor o igual a **10** (el valor final en el cual la condición se hace verdadera). Note que el cuerpo de este **while** se ejecuta, aun cuando la variable de control llegue a **10**. El ciclo se terminará cuando la variable de control exceda de **10** (es decir, **counter** se convierte en **11**).

Los programadores de C por lo regular elaborarían la programación de la figura 4.1 de manera más concisa, inicializando **counter** a **0** y remplazando la estructura **while** con

```
While (++counter <= 10 )
   printf ("%d/n", counter);
```

Este código ahorra un enunciado porque el incremento se efectúa de forma directa en la condición **while**, antes de que la condición sea probada. También, este código elimina las llaves alrededor del cuerpo del **while**, porque el **while** ahora sólo contiene un enunciado. Para codificar de esta forma tan concisa y condensada, se requiere de algo de práctica.

Error común de programación 4.1

Dado que los valores en punto flotante pueden ser aproximados, el control de contador de ciclos con variables de punto flotante puede dar como resultado valores de contador no precisos y pruebas no exactas de terminación.

Práctica sana de programación 4.1

Controlar el contador de ciclos con valores enteros.

Práctica sana de programación 4.2

Hacer sangría en los enunciados en el cuerpo de cada estructura de control.

Práctica sana de programación 4.3

Colocar una línea en blanco antes y después de cada estructura de control principal, para que se destaque en el programa.

Práctica sana de programación 4.4

Demasiados niveles anidados pueden dificultar la comprensión de un programa. Como regla general, procure evitar el uso de más de tres niveles de sangrías.

Práctica sana de programación 4.5

La combinación de espaciamiento vertical antes y después de las estructuras de control y las sangrías de los cuerpos de las estructuras de control dentro de los encabezados de las estructuras de control, le da a los programas una apariencia bidimensional que mejora de manera significativa la legibilidad del programa.

4.4 La estructura de repetición for

La estructura de repetición **for** maneja de manera automática todos los detalles de la repetición controlada por contador. Para ilustrar los poderes de **for**, volvamos a escribir el programa de la figura 4.1. El resultado se muestra en la figura 4.2.

El programa funciona como sigue: cuando la estructura **for** inicia su ejecución, la variable de control **counter** se inicializa a **1**. A continuación, se verifica la condición de continuación de ciclo **counter <= 10**. Dado que el valor inicial de counter es **1**, se satisface esta condición, por lo que el enunciado **printf** imprime el valor de counter, es decir **1**.

```
/* Counter-controlled repetition with the for structure */
#include <stdio.h>

main()
{
   int counter;

   /* initialization, repetition condition, and increment */
   /* are all included in the for structure header        */
   for (counter = 1; counter <= 10; counter++)
      printf("%d\n", counter);

   return 0;
}
```

Fig. 4.2 Repetición controlada por contador con la estructura **for**.

A continuación la variable de control **counter** es incrementada por la expresión **counter++**, y el ciclo empieza otra vez, con la prueba de continuación de ciclo. Dado que la variable de control ahora es igual a **2**, el valor final aún no es excedido, y el programa por lo tanto ejecuta otra vez el enunciado **printf**. Este proceso continúa hasta que la variable de control **counter** es incrementada a su valor final de 11 lo que hará que falle la prueba de continuación de ciclo y termine la repetición. El programa continúa ejecutando el primer enunciado que encuentra después de la estructura **for** (en este caso, el enunciado **return** al final del programa).

La figura 4.3 analiza más de cerca la estructura **for** de la figura 4.2. Advierta que la estructura **for** "lo hace todo" especifica cada uno de los elementos necesarios para la repetición controlada por contador con una variable de control. Si en el cuerpo del **for** existe más de un enunciado, se requerirán de llaves para definir el cuerpo del ciclo.

Note que la figura 4.2 utiliza la condición de continuación de ciclo **counter <= 10**. Si el programador hubiera escrito en forma incorrecta **counter < 10**, entonces el ciclo sólo se hubiera ejecutado 9 veces. Este es un error común de lógica, conocido como un *error de diferencia por uno.*

Error común de programación 4.2

Usar un operador relacional incorrecto o usar un valor final incorrecto de un contador de ciclo, en la condición de una estructura **while** *o* **for**, *puede causar errores de diferencia por uno.*

Práctica sana de programación 4.6

Usar el valor final en la condición de una estructura **while** *o* **for** *y utilizar el operador relacional* **<=** *auxiliará a evitar errores de diferencia por uno. Para un ciclo que se utilice para imprimir los valores del 1 al 10, por ejemplo, la condición de confirmación de ciclo debería ser* **counter <= 10** *en vez de* **counter < 11** *o bien* **counter < 10**.

El formato general de la estructura **for** es

```
for (expresión1; expresión2; expresión3)
   enunciado
```

donde *expresión1* inicializa la variable de control de ciclo, *expresión2* es la condición de continuación del ciclo, y *expresión3* incrementa la variable de control. En la mayor parte de los

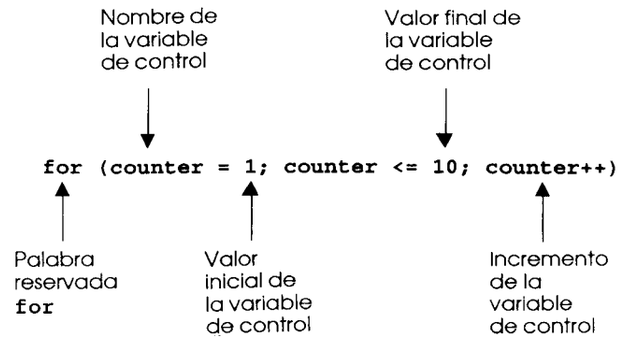


Fig. 4.3 Componentes de un encabezado típico **for**.

casos, la estructura **for** puede representarse mediante una estructura equivalente **while**, como sigue:

```
expresión1;
while (expresión2) {
   enunciado
   expresión3;
}
```

Existe una excepción para esta regla, misma que se analizará en la Sección 4.9.

A menudo, *expresión1* y *expresión3* son expresiones en listas separadas por coma. Las comas se utilizan aquí como *operadores coma,* garantizando que las listas de expresiones se evalúen de izquierda a derecha. El valor y el tipo de una lista de expresiones separada por coma, es el valor y el tipo de la expresión más a la derecha de dicha lista. El operador coma es utilizado prácticamente sólo en estructuras **for**. Su uso principal es permitir al programador utilizar múltiples expresiones de inicialización y/o múltiples incrementos. Por ejemplo, en una sola estructura **for** pudieran existir dos variables de control que deban ser inicializadas e incrementadas.

Práctica sana de programación 4.7

Coloque sólo expresiones que involucren las variables de control en las secciones de inicialización y de incremento de una estructura **for**. *Las manipulaciones de las demás variables deberían de aparecer, ya sea antes del ciclo (si se ejecutan una vez, como los enunciados de incialización) o dentro del cuerpo del ciclo (si se ejecutan una vez en cada repetición, como son los enunciados incrementales o decrementales).*

Las tres expresiones de la estructura **for** son opcionales. Si se omite *expresión2*, C supondrá que la condición es verdadera, creando por lo tanto un ciclo infinito. También se puede omitir la *expresión1*, si la variable de control se inicializa en alguna otra parte del programa. La *expresión3* podría también omitirse, si el incremento se calcula mediante enunciados en el cuerpo de la estructura **for**, o si no se requiere de ningún incremento. La expresión incremental en la estructura **for** actúa como un enunciado de C independiente, al final del cuerpo del **for**. Por lo tanto, las expresiones

```
counter = counter + 1
counter += 1
++counter
counter++
```

son equivalentes todas ellas, en la porción incremental de la estructura **for**. Muchos programadores C prefieren la forma **counter++**, porque el incremento ocurre después de que se haya ejecutado el cuerpo del ciclo. Por lo tanto, la forma postincremental parece más natural. Dado que la variable que se postincrementa o se preincrementa aquí no aparece en una expresión, ambas formas de incremento surten el mismo efecto. Los dos puntos y comas en la estructura **for** son requeridos.

Error común de programación 4.3

Usar comas en vez de puntos y coma en un encabezado **for**.

Error común de programación 4.4

Colocar un punto y coma de inmediato a la derecha de un encabezado **for** *hace del cuerpo de esta estructura* **for** *un enunciado vacío. Por lo regular esto es un error lógico*

4.5 La estructura for: notas y observaciones

1. Tanto la inicialización, como la condición de continuación de ciclo, y el incremento pueden contener expresiones aritméticas. Por ejemplo, suponga que `x = 2` y `y = 10`, el enunciado

   ```
   for (j = x;  j <= 4 * x * y; j += y / x)
   ```

   es equivalente al enunciado

   ```
   for (j = 2;  j <= 80;  j += 5)
   ```

2. El "incremento" puede ser negativo (en cuyo caso realmente se trata de un decremento, y el ciclo de hecho contará hacia atrás).
3. Si la condición de continuación de ciclo resulta falsa al inicio, la porción del cuerpo del ciclo no se ejecutará. En vez de ello, la ejecución seguirá adelante con el enunciado que siga a la estructura **for**.
4. La variable de control con frecuencia se imprime o se utiliza en cálculos en el cuerpo de un ciclo, pero esto no es necesario. Es común utilizar la variable de control para controlar la repetición, aunque jamás se mencione la misma dentro del cuerpo del ciclo.
5. El diagrama de flujo de la estructura **for** es muy similar al de la estructura **while**. Por ejemplo, el diagrama de flujo del enunciado **for**

   ```
   for (counter = 1; counter <= 10; counter++)
   printf("%d", counter);
   ```

   se muestra en la figura 4.4. Este diagrama de flujo deja claro que la incialización ocurre una vez y que el incremento ocurre después de que se ejecuta el enunciado del cuerpo. Advierta que (independiente de los pequeños círculos y flechas), el diagrama de flujo contiene sólo los símbolos de rectángulo y diamante. Imagine, otra vez, que el programador tiene acceso a un contenedor profundo de estructuras **for** vacías —tantas como necesite el programador para apilarlas y anidarlas con otras estructuras de control, para formar una implantación estructurada del flujo de control de un algoritmo. Y otra vez, los rectángulos y los diamantes se rellenarán con acciones y decisiones apropiadas al algoritmo.

Práctica sana de programación 4.8

Aunque el valor de la variable de control puede ser modificado en el cuerpo de un ciclo **for**, *ello podría llevarnos a errores sutiles. Lo mejor es no cambiarlo.*

4.6 Ejemplos utilizando la estructura for

Los siguientes ejemplos muestran métodos de variar en una estructura **for** la variable de control.

a) Varíe la variable de control de `1` a `100` en incrementos de `1`.

```
for (i = 1; i <= 100; i++)
```

b) Varíe la variable de control de `100` a `1` en incrementos de `-1` (decrementos de `1`).

```
for (i = 100; i >= 1; i--)
```

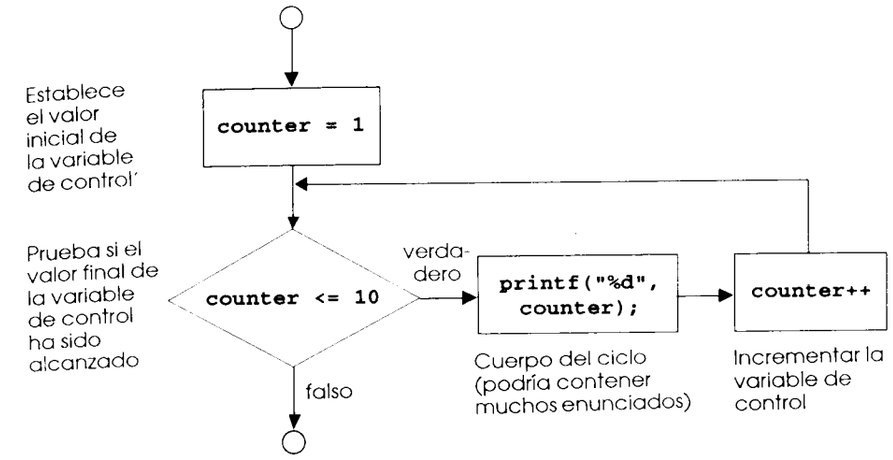



Fig. 4.4 Diagrama de flujo de una estructura **for** típica

c) Variar la variable de control de `7` a `77` en pasos de `7`.

```
for (i = 7; i <= 77; i += 7)
```

d) Variar la variable de control de `20` a `2` en pasos de `-2`

```
for (i = 20; i >= 2; i -= 2)
```

e) Variar la variable de control a lo largo de la siguiente secuencia de valores: **2, 5, 8, 11, 14, 17, 20.**

```
for (j = 2; j <= 20; j += 3)
```

f) Variar la variable de control de acuerdo a la siguiente secuencia de valores: **99, 88, 77, 66, 55, 44, 33, 22, 11, 0.**

```
for (j = 99; j >= 0; j -=11)
```

Los siguientes dos ejemplos proporcionan aplicaciones simples de la estructura **for**. El programa de la figura 4.5 utiliza la estructura **for** para sumar todos los enteros pares desde `2` hasta `100`.

Note que el cuerpo de la estructura **for** de la figura 4.5 podría de hecho combinarse en la parte más a la derecha del encabezado **for** mediante el uso del operador coma, como sigue:

```
for (number = 2; number <= 100; sum += number, number += 2)
   ;
```

La inicialización `sum = 0` también podría ser combinada en la sección de inicialización del **for**.

Práctica sana de programación 4.9

Aunque los enunciados que anteceden a un **for**, *y los enunciados del cuerpo de* **for**, *a menudo pueden ser combinados en un encabezado* **for**, *evite hacerlo, porque hace el programa más difícil de leer.*

```
/* Summation with for */
#include <stdio.h>

main()
{
   int sum = 0, number;

   for (number = 2; number <= 100; number += 2)
      sum += number;

   printf("Sum is %d\n", sum);

   return 0;
}
```

```
Sum is 2550
```

Fig. 4.5 Suma utilizando `for`.

Práctica sana de programación 4.10

Limite, si es posible, a una sola línea el tamaño de los encabezados de estructuras de control.

El ejemplo siguiente calcula el interés compuesto, utilizando la estructura **`for`**. Considere el siguiente enunciado de problema:

Una persona invierte $1000.00 en una cuenta de ahorros, que reditúa un interés del 5 %. Suponiendo que todo el interés se queda en depósito dentro de la cuenta, calcule e imprima la cantidad de dinero en la cuenta, al final de cada año, durante 10 años. Para la determinación de estas cantidades utilice la fórmula siguiente:

$$a = p(1 + r)^n$$

donde

p es la cantidad originada invertida (es decir, el principal)
r es la tasa anual de interés
n es el número de años
a es la cantidad en depósito al final del año *n*.

Este problema incluye un ciclo, que ejecuta el cálculo indicado para cada uno de los 10 años en los cuales el dinero se queda en depósito. La solución aparece en la figura 4.6.

La estructura **`for`** ejecuta 10 veces el cuerpo del ciclo, variando la variable de control de 1 a 10, en incrementos de 1. Aunque C no incluye un operador de exponenciación, podemos, sin embargo, utilizar para este fin la función estándar de biblioteca **`pow`**. La función **`pow(x, y)`** calcula el valor de **`x`**, elevado a la potencia **`y`**. Toma dos argumentos del tipo **`double`** y devuelve un valor **`double`**. El tipo **`double`** es un tipo de punto flotante parecido a **`float`**, pero una variable de tipo **`double`** puede almacenar un valor de una magnitud mucho mayor y con mayor precisión que **`float`**. Note que, siempre que una función matemática como es **`pow`** sea utilizada, deberá incluirse el archivo de cabecera **`math.h`**. De hecho, este programa funcionaría mal sin la inclusión


```
/* Calculating compound interest */
#include <stdio.h>
#include <math.h>

main()
{
   int year;
   double amount, principal = 1000.0, rate = .05;

   printf("%4s%21s\n", "Year", "Amount on deposit");

   for (year = 1; year <= 10; year++) {
      amount = principal * pow(1.0 + rate, year);
      printf("%4d%21.2f\n", year, amount);
   }

   return 0;
}
```

```
Year   Amount on deposit
1            1050.00
2            1102.50
3            1157.62
4            1215.51
5            1276.28
6            1340.10
7            1407.10
8            1477.46
9            1551.33
10           1628.89
```

Fig. 4.6 Cómo calcular interés compuesto utilizando `for`.

de **`math.h`**. La función **`pow`** requiere dos argumentos **`double`**. Note que **`year`** es un entero. El archivo **`math.h`** incluye información que le indica al compilador que convierta el valor de **`year`** a una representación temporal **`double`**, antes de llamar a la función. Esta información queda contenida en algo que se conoce como *función prototipo* **`pow`**. Las funciones prototipo son una nueva característica de importancia de ANSI C y se explican en el capítulo 5. En este capítulo 5 proporcionamos un resumen de la función **`pow`** y de otras funciones matemáticas de biblioteca.

Note que hemos declarado las variables **`amount`**, **`principal`** y **`rate`** como del tipo **`double`**. Hemos hecho lo anterior por razones de simplicidad, porque estamos manejando fracciones de dólares.

Práctica sana de programación 4.11

No utilice variables del tipo `float` o `double` para llevar a cabo cálculos monetarios. La falta de precisión de los números de punto flotante pueden causar errores, que resultarán en valores monetarios incorrectos. En los ejercicios, exploramos la utilización de enteros para la ejecución de cálculos monetarios.

A continuación una explicación sencilla de lo que podría salir mal al utilizar **float** o bien **double** para representar cantidades en dólares.

Dos cantidades en dólares **float** almacenadas en la máquina podrían ser 14.234 (que con **%.2f**, imprime como 14.23) y 18.673 (el cual con **%.2f**, imprime como 18.67). Cuando estas cantidades se suman, producen la suma 32.907, que con **%.2f**, imprime como 32.91. Por lo tanto su impresión aparecería como

```
  14.23
+ 18.67
-------
  32.91
```

la suma de los números individuales, según se ven impresos ¡debería ser 32.90! ¡Queda usted advertido!

El especificador de conversión **%21.2f** es utilizado en el programa para imprimir el valor de la variable **amount**. El **21** es el especificador de conversión que indica el ancho del campo en el cual el valor se imprimirá. Un ancho de campo de **21** define que el valor impreso aparecerá en **21** posiciones de impresión. El **2** especifica la precisión (es decir, el número de posiciones decimales). Si el número de caracteres desplegados es menor que el ancho del campo, entonces el valor quedará de forma automática *justificado a la derecha* dentro de ese campo. Esto es en particular útil para alinear valores de punto flotante que tengan la misma precisión. A fin de *justificar a la izquierda* un valor en un campo, coloque un - (signo menos) entre el % y el ancho del campo. Note que el signo menos, también puede ser utilizado para alinear a la izquierda a los enteros (como en **%-6d**) y a las cadenas de caracteres (como en **%-8s**). En el capítulo 9 analizaremos con todo detalle las poderosas capacidades de formato de **printf** y de **scanf**.

4.7 La estructura de selección múltiple switch

En el capítulo 3, analizamos la estructura de una selección **if**, y la estructura de doble selección **if/else**. En forma ocasional, un algoritmo contendrá una serie de decisiones, en las cuales una variable o expresión se probará por separado contra cada uno de los valores constantes enteros que puede asumir, y se tomarán diferentes acciones. Para esta forma de toma de decisiones se proporciona una estructura de selección múltiple **switch** C.

La estructura **switch** está formada de una serie de etiquetas **case**, y de un caso opcional **default**. El programa en la figura 4.7 utiliza **switch** para contar el número de cada distinta letra de calificación que los estudiantes alcanzaron en un examen.

En el programa, el usuario escribe las calificaciones, en letras, correspondientes a una clase. Dentro del encabezado **while**,

```
while ( ( grade = getchar() ) != EOF)
```

la asignación entre paréntesis **(grade = getchar())** se ejecuta en primer término. La función **getchar** (proveniente de la biblioteca estándar de entrada/salida) lee un carácter del teclado y almacena este carácter en la variable entera **grade**. Los caracteres se almacenan por lo regular en variables del tipo ***char***. Sin embargo, una característica importante de C, es que los caracteres pueden ser almacenados en cualquier tipo de dato entero, porque en la computadora son representados como enteros de un byte. Por lo tanto, podemos tratar a un carácter como si fuera ya sea o un entero, o un carácter, dependiendo de su uso. Por ejemplo, el enunciado

```
printf("The character (%c) has the value %d.\n", 'a', 'a');
```



```
/* Counting letter grades */
#include <stdio.h>

main()
{
   int grade;
   int aCount = 0, bCount = 0, cCount = 0,
       dCount = 0, fCount = 0;

   printf("Enter the letter grades.\n");
   printf("Enter the EOF character to end input.\n");

   while ( ( grade = getchar() ) != EOF) {

      switch (grade) {    /* switch nested in while */

         case 'A': case 'a':  /* grade was uppercase A */
            ++aCount;         /* or lowercase a */
            break;

         case 'B': case 'b':  /* grade was uppercase B */
            ++bCount;         /* or lowercase b */
            break;

         case 'C': case 'c':  /* grade was uppercase C */
            ++cCount;         /* or lowercase c */
            break;

         case 'D': case 'd':  /* grade was uppercase D */
            ++dCount;         /* or lowercase d */
            break;

         case 'F': case 'f':  /* grade was uppercase F */
            ++fCount;         /* or lowercase f */
            break;

         case '\n': case ' ': /* ignore these in input */
            break;

         default:        /* catch all other characters */
            printf("Incorrect letter grade entered.");
            printf(" Enter a new grade.\n");
            break;
      }
   }

   printf("\nTotals for each letter grade are:\n");
   printf("A: %d\n", aCount);
   printf("B: %d\n", bCount);
   printf("C: %d\n", cCount);
   printf("D: %d\n", dCount);
   printf("F: %d\n", fCount);

   return 0;
}
```

Fig. 4.7 Un ejemplo del uso de **switch** (parte 1 de 2).

```
Enter the letter grades.
Enter the EOF character to end input
A
B
C
C
A
D
F
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
B

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Fig. 4.7 Un ejemplo del uso de **switch** (parte 2 de 2).

utiliza los especificadores de conversión **%c** y **%d**, para imprimir el carácter **a** y su valor como entero, respectivamente. El resultado es

```
The character (a) has the value 97.
```

El entero 97 es la representación numérica del carácter, dentro de la computadora. Muchas computadoras hoy día utilizan el *conjunto de caracteres ASCII (American Standard Code for Information Interchange),* en el cual el 97 representa la letra minúscula **'a'**. En el Apéndice D se presenta una lista de los caracteres ASCII y sus valores decimales. Los caracteres pueden ser leídos utilizando **scanf** mediante el uso del especificador de conversión %c.

De hecho todos los enunciados de asignación en conjunto tienen un valor. Este es precisamente el valor que se asigna a la variable del lado izquierdo del signo de =. El valor de la asignación **grade = getchar()** es el carácter que es regresado por **getchar** y asignado a la variable **grade**.

El hecho que los enunciados de asignación tengan valores puede ser útil para inicializar varias variables con un mismo valor. Por ejemplo,

```
a = b = c = 0;
```

primero evalúa la asignación **c = 0** (porque el operador = se asocia de derecha a izquierda). A la variable **b**, a continuación, se le asigna el valor de la asignación **c = 0** (que es 0). Entonces, a la variable **a** se le asigna el valor de la asignación **b = (c = 0)** (lo que también es 0). En el programa, el valor de la asignación **grade = getchar()** se compara con el valor de **EOF**, (un

símbolo cuya siglas significan "fin de archivo"). Utilizamos **EOF** (que tiene por lo regular el valor de -1) como valor centinela. El usuario escribe una combinación de tecleos, dependiendo del sistema, que signifiquen "fin de archivo", es decir, "ya no tengo más datos a introducir". **EOF** es una constante simbólica entera, definida en el archivo de cabecera **<stdio.h>** (en el capítulo 6 veremos cómo son definidas las constantes simbólicas). Si el valor asignado a **grade** es igual a **EOF**, el programa se termina. En este programa hemos decidido representar caracteres como int, porque **EOF** tiene un valor entero (otra vez, lo normal es -1).

Sugerencia de portabilidad 4.1

Las combinaciones de teclas para la introducción de **EOF** *(fin de archivo) son dependientes del sistema.*

Sugerencia de portabilidad 4.2

Hacer la prueba buscando la constante simbólica **EOF**, *en vez de buscar -1, hace más portátiles a los programas. El estándar ANSI indica que* **EOF** *es un valor entero negativo (pero no necesariamente -1). Por lo tanto,* **EOF** *pudiera tener diferentes valores en diferentes sistemas*

En sistemas UNIX y en muchos otros, el indicador **EOF** se introduce escribiendo la secuencia

<return> <ctrl-d>

Esta notación significa que se oprima la tecla de entrar y de forma simultánea se presione tanto la tecla **ctrl** como la tecla **d**. En otros sistemas, como en el VAX VMS de Digital Equipment Corporation, o en el MS-DOS de Microsoft Corporation, el indicador **EOF** puede ser introducido escribiendo

<ctrl-z>

El usuario introduce las calificaciones con el teclado. Cuando se oprime la tecla de retorno (o de entrar) los caracteres son leídos por la función **getchar**, un carácter a la vez. Si el carácter introducido no es igual a **EOF**, se introduce en la estructura **switch**. La palabra reservada **switch** es seguida por el nombre de variable **grade** entre paréntesis. Esto se conoce como la *expresión de control.* El valor de esta expresión es comparado con cada una de las *etiquetas case*. Suponga que el usuario ha escrito la letra **C** como calificación. La **C** automáticamente se compara con cada uno de los **case** dentro de **switch**. Si ocurre una coincidencia (**case 'C':**), se ejecutarán los enunciados correspondientes a dicho **case**. En el caso de la letra **C**, **cCount** se incrementará en **1**, y de inmediato mediante el enunciado **break** se sale de la estructura **switch**.

El enunciado **break** causa que el control de programa continúe con el primer enunciado que sigue después de la estructura **switch**. Se utiliza el enunciado **break**, porque de lo contrario los **case**s en un enunciado **switch** se ejecutarían juntos. Si en alguna parte de una estructura **switch** no se utiliza **break**, entonces, cada vez que ocurre una coincidencia en la estructura, se ejecutarían todos los enunciados de los **case**s remanentes. (Esta característica rara vez es de utilidad, aunque resulta perfecta para programar la canción iterativa "¡The twelve Days of Christmas!"). Si no existe coincidencia, el caso **default** es ejecutado y se imprime un mensaje de error.

Cada **case** puede tener una o más acciones. La estructura **switch** es diferente de todas las demás estructuras, en el sentido de que no se requieren llaves alrededor de varias acciones en un **case** de un **switch**. La estructura general de selección múltiple **switch** (que utilice a un **break** en cada **case**) tiene un diagrama de flujo como el de la figura 4.8.

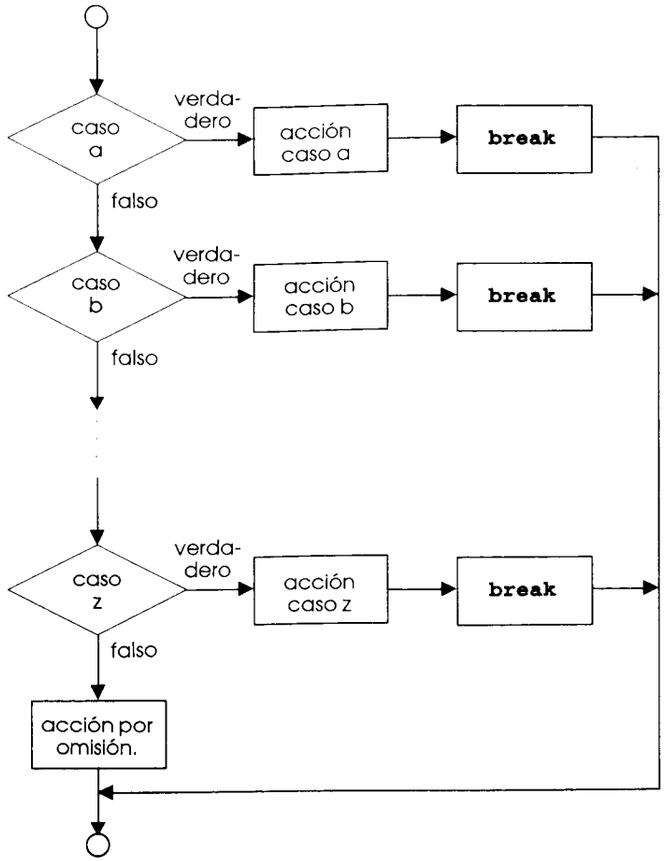


Fig. 4.8 La estructura de selección múltiple **switch**.

El diagrama de flujo deja claro que cada enunciado **break**, al final de un **case**, hace que el control dé salida de inmediato a la estructura **switch**. Otra vez, advierta que (independientemente de los pequeños círculos y flechas) el diagrama de flujo sólo contiene símbolos rectángulo y diamante. Imagínese, otra vez, que el programador tiene acceso a un contenedor profundo de estructuras **switch** vacías —tantas como pudiera necesitar el programador para apilarlas y anidarlas junto con otras estructuras de control, para formar una implantación estructurada del flujo de control de un algoritmo. Y de nuevo, los rectángulos y los diamantes a continuación serán llenados con acciones y decisiones apropiadas al algoritmo.

Error común de programación 4.5

Olvidar en una estructura **switch** *un enunciado* **break**, *cuando se requiere de uno.*

Práctica sana de programación 4.12

Proveer un caso **default** *en enunciados* **switch**. *En un* **switch** *serán ignorados los casos no aprobados de forma explícita. El caso* **default** *ayuda a evitar lo anterior, enfocando al programador sobre la necesidad de procesar condiciones excepcionales. Existen situaciones en las cuales no se requiere de procesamiento* **default**.

Práctica sana de programación 4.13

Aunque las cláusulas **case** *y la cláusula de caso* **default** *en una estructura* **switch** *pueden ocurrir en cualquier orden, se considera una práctica sana de programación colocar al final la cláusula* **default**.

Práctica sana de programación 4.14

En una estructura **switch**, *cuando la cláusula* **default** *se enlista al final, el enunciado* **break** *no es requerido. Pero algunos programadores incluyen este* **break**, *para fines de claridad y simetría con otros* **cases**.

En la estructura **switch** de la figura 4.7, las líneas

```
case '\n': case ' ':
   break;
```

hacen que el programa se salte los caracteres de nueva línea y de espacios vacíos. La lectura de caracteres uno a la vez puede causar ciertos problemas. Para que el programa lea los caracteres, deben ser enviados a la computadora oprimiendo la *tecla de entrar* del teclado. Esto hace que se coloque un carácter de nueva línea en la entrada, después del carácter que deseamos procesar. A menudo, este carácter de nueva línea debe ser procesado en forma especial, para que el programa funcione de manera correcta. Al incluir los casos precedentes en nuestra estructura **switch**, evitamos que en el caso **default** se imprima el mensaje de error, cada vez que se encuentra en la entrada un carácter de nueva línea o de espacio.

Error común de programación 4.6

No procesar los caracteres de nueva línea en la entrada al leer los caracteres uno a la vez, puede ser causa de errores lógicos.

Práctica sana de programación 4.15

Recuerde proporcionar capacidades de proceso para los caracteres de nueva línea en la entrada, cuando se procesen caracteres uno a la vez.

Note que varias etiquetas de caso enlistadas juntas (como **case 'D':case 'd':** en la figura 4.7) significan que el mismo conjunto de acciones ocurrirá para cualquiera de estos casos.

Al utilizar la estructura **switch**, recuerde que puede ser usada sólo para probar una *expresión integral constante*, es decir, cualquier combinación de constantes de carácter y de constantes enteros que tengan un valor constante entero. Una constante de carácter se representa como un carácter específico, entre comillas sencillas como es **'A'**. Los caracteres deben ser encerrados dentro de comillas sencillas para que sean reconocidos como constantes de carácter. Las constantes enteras son solo valores enteros. En nuestro ejemplo hemos utilizado constantes de carácter. Recuerde que los caracteres de hecho son valores pequeños enteros.

UNIVERSIDAD DE LA REPÚBLICA FACULTAD DE INGENIERÍA

Los lenguajes portátiles, como C, deben tener tamaños flexibles de tipos de datos. Diferentes aplicaciones pudieran necesitar enteros de tamaños diferentes. C proporciona varios tipos de datos para representar enteros. El rango de los valores de enteros para cada tipo depende del hardware particular de cada computadora. Además de los tipos **int** y **char**, C proporciona los tipos **short** (que es una abreviatura de **short int**) y **long** (que es una abreviatura de **long int**). El estándar ANSI especifica que el rango mínimo de valores para enteros **short** es ±32767. Para la gran mayoría de los cálculos enteros, los enteros **long** son suficientes. El estándar define que el rango mínimo de valores para los enteros **long** es ±2147483647. En la mayoría de las computadoras, los **int** son equivalentes ya sea a **short** o a **long**. El estándar indica que el rango de valores para un **int**, es por lo menos igual al rango de los enteros **short** y no mayor que el rango de los enteros **long**. El tipo de dato **char** puede ser utilizado para representar enteros en el rango de ±127, o cualquiera de los caracteres del conjunto de caracteres de la computadora.

Sugerencia de portabilidad 4.3

Dado que de un sistema a otro **int** *varía en tamaño, utilice enteros* **long**, *si espera procesar enteros fuera del rango ±32767, y desearía ser capaz de ejecutar el programa en varios sistemas de cómputo diferentes.*

Sugerencia de rendimiento 4.1

En situaciones orientadas a rendimiento, donde la memoria está escasa o se requiere de velocidad, pudiera ser deseable utilizar tamaños de enteros más pequeños.

4.8 La estructura de repetición do/while

Las estructura de repetición **do/while** es similar a la estructura **while**. En la estructura **while**, la condición de continuación de ciclo se prueba al principio del ciclo, antes de ejecutarse el cuerpo del mismo. La estructura **do/while** prueba la condición de continuación del ciclo, *después* de ejecutar el cuerpo del ciclo y, por lo tanto, el cuerpo del ciclo se ejecutará por lo menos una vez. Cuando termina **do/while**, la ejecución continuará con el enunciado que aparezca después de la cláusula **while**. Note que en la estructura **do/while** no es necesario utilizar llaves, si en el cuerpo existe sólo un enunciado. Sin embargo, por lo regular las llaves se utilizan, para evitar confusión entre las estructuras **while** y **do/while**. Por ejemplo,

```
while (condición)
```

se considera normalmente como el encabezado de una estructura **while**. Un **do/while** sin llaves rodeando el cuerpo de un solo enunciado, aparece como

```
do
   enunciado
while (condición);
```

lo que podría ser motivo de confusión. La última línea **while** (*condición*) **;** pudiera ser interpretada de forma equivocada por el lector como una estructura **while** conteniendo un enunciado vacío. Por lo tanto, el **do/while** con un enunciado, a menudo se escribe como sigue, a fin de evitar confusión:


```
do {
enunciado
} while (condición);
```

Práctica sana de programación 4.16

Algunos programadores incluyen siempre llaves en una estructura **do/while**, *aún si las llaves no son necesarias. Esto ayuda a eliminar ambigüedad entre la estructura* **do/while** *con un enunciado, y la estructura* **while**.

Error común de programación 4.7

Se generarán ciclos infinitos en una estructura **while**, **for** *o bien* **do/while** *cuando la condición de continuación de ciclo nunca se convierte en falsa. A fin de evitar lo anterior, asegúrese de que no exista un punto y coma inmediatamente después del encabezado de una estructura* **while** *o* **for**. *En un ciclo controlado por contador, asegúrese que la variable de control es incrementada (o decrementada) en el cuerpo del ciclo. En un ciclo controlado por centinela, asegúrese que el valor centinela es eventualmente introducido.*

El programa de la figura 4.9 utiliza una estructura **do/while** para imprimir los números del 1 al 10. Note que la variable de control **counter** es preincrementada en la prueba de continuación de ciclo. Note también la utilización de llaves para encerrar el cuerpo de un solo enunciado del **do/while**.

La estructura del **do/while** se diagrama en la figura 4.10. Este diagrama de flujo demuestra que la condición de continuación de ciclo no se ejecutará sino hasta después de que la acción se lleve a cabo por lo menos una vez. De nuevo, note (que además de los pequeños círculos y flechas) el diagrama de flujo sólo contiene un símbolo de rectángulo y uno de diamante. Imagine, repetimos, que el programador tiene acceso a un contenedor profundo de estructuras **do/while** vacías tantas como pudiera necesitar el programador para apilarlas y anidarlas junto con otras estructuras de control, para formar una implantación estructurada del flujo de control de un algoritmo. Y otra vez, los rectángulos y diamantes serán entonces llenados con acciones y decisiones apropiadas a dicho algoritmo.

```
/* Using the do/while repetition structure */
#include <stdio.h>

main()
{
   int counter = 1;

   do {
      printf("%d  ", counter);
   } while (++counter <= 10);

   return 0;
}
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 4.9 Cómo usar la estructura **do/while**.

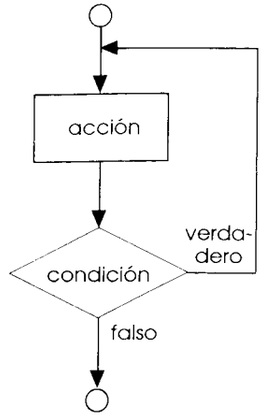


Fig. 4.10 La estructura de repetición do/while.

4.9 Los enunciados break y continue

Los enunciados **break** y **continue** se utilizan para modificar el flujo de control. El enunciado **break**, cuando se utiliza o ejecuta en una estructura **while, for, do/while,** o **switch**, causa la salida inmediata de dicha estructura. La ejecución del programa continúa con el primer enunciado después de la estructura. Los usos comunes del enunciado **break** son para escapar en forma prematura de un ciclo, o para saltar el resto de una estructura **switch** (como en la figura 4.7). La figura 4.11 demuestra el enunciado **break** en una estructura de repetición **for.** Cuando la estructura **if** detecta que **x** se ha convertido en **5**, se ejecuta **break**. Esto da por terminado el enunciado **for,** y el programa continúa con el **printf** existente después de **for**. El ciclo se ejecuta por completo cuatro veces.

El enunciado **continue**, cuando se ejecuta en una estructura **while, for, do/while,** salta los enunciados restantes del cuerpo de dicha estructura, y ejecuta la siguiente iteración del ciclo. En estructuras **while** y **do/while,** la prueba de continuación de ciclo se valúa de inmediato de que se haya ejecutado el enunciado **continue**. En la estructura **for**, se ejecuta la expresión incremental, y a continuación se valúa la prueba de continuación de ciclo. Anteriormente indicamos que la estructura **while** podría ser utilizada, en la mayor parte de los casos, para representar la estructura **for**. Una excepción ocurre cuando la expresión incremental en la estructura **while** sigue a un enunciado **continue**. En este caso, el incremento no se lleva a cabo antes de la prueba de la condición de continuación de repetición, y no se ejecuta el **while** de la misma forma que en el caso del **for**. En la figura 4.2 se utiliza el enunciado **continue** en una estructura **for,** para saltar el enunciado **printf** de la estructura, y empezar con la siguiente iteración del ciclo.

Práctica sana de programación 4.17

Algunos programadores son de la opinión que **break** *y* **continue** *violan las normas de la programación estructurada. Dado que los efectos de estos enunciados pueden ser obtenidos mediante técnicas de programación estructuradas que aprenderemos pronto, estos programadores no utilizan* **break** *ni* **continue**.


```
/* Using the break statement in a for structure */
#include <stdio.h>

main()
{
   int x;

   for (x = 1; x <= 10; x++) {

      if (x == 5)
         break;     /* break loop only if x == 5 */

      printf("%d ", x);
   }

   printf("\nBroke out of loop at x == %d\n", x);
   return 0;
}
```

```
1 2 3 4
Broke out of loop at x == 5
```

Fig. 4.11 Cómo utilizar el enunciado break en una estructura for.

```
/* Using the continue statement in a for structure */
#include <stdio.h>

main()
{
   int x;

   for (x = 1; x <= 10; x++) {

      if (x == 5)
         continue;  /* skip remaining code in loop only
                       if x == 5 */

      printf("%d ", x);
   }

   printf("\nUsed continue to skip printing the value 5\n");
   return 0;
}
```

```
1 2 3 4 5 6 7 8 9 10
Used continue to skip printing the value 5
```

Fig. 4.12 Cómo utilizar el enunciado continue en una estructura for.

Sugerencia de rendimiento 4.2

Los enunciados `break` *y* `continue`*, cuando son usados de forma adecuada, se ejecutan más aprisa que las técnicas estructuradas correspondientes que pronto aprenderemos.*

Observación de ingeniería de software 4.1

Existe cierta contraposición entre alcanzar ingeniería de software de calidad y conseguir software de mejor rendimiento. A menudo una de estas metas se consigue a expensas de la otra.

4.10 Operadores lógicos

Hasta ahora hemos estudiado sólo *condiciones simples*, como son `counter < = 10`, `total >1000` y `number != sentinelValue`. Hemos expresado estas condiciones en términos de los operadores relacionales >, <, >=, y <=, y de los operadores de igualdad == y !=. Cada decisión probaba precisamente una sola condición. Si quisiéramos probar varias condiciones en el proceso de llevar a cabo una decisión, tendríamos que ejecutar estas pruebas en enunciados por separado, o en estructuras anidadas `if`, o bien `if/else`.

C proporciona *operadores lógicos,* que pueden ser utilizados para formar condiciones más complejas, al combinar condiciones simples. Los operadores lógicos son `&&` (*el AND lógico*), `||` (*OR lógico*), y `!` (*NOT lógico* también conocido como *negación lógica*). Estudiaremos ejemplos de cada uno de los anteriores.

Suponga que deseamos asegurarnos, en algún momento en un programa, que dos condiciones son *ambas* verdaderas, antes de que seleccionemos una cierta trayectoria de ejecución. En este caso podemos utilizar el operador lógico `&&`, como sigue:

```
if (gender == 1 && age = 65)
   ++seniorFemales;
```

Este enunciado `if` contiene dos condiciones simples. La condición `gender == 1` pudiera ser evaluada, por ejemplo, a fin de determinar si una persona es mujer. La condición `age >= 65` se evalúa para determinar si la persona es un ciudadano senior. Las dos condiciones simples se evalúan primero, porque las precedencias de == y de >= ambas son más altas que la precedencia de `&&`. A continuación, el enunciado `if` considera la condición combinada.

```
gender == 1 && age >= 65
```

Esta condición es verdadera si y sólo si ambas condiciones simples son verdaderas. Por último, si esta condición combinada es de hecho verdadera, entonces el contador de `seniorFemales` se incrementará en `1`. Si cualquiera o ambas de las condiciones simples son falsas, entonces el programa salta la incrementación y prosigue al enunciado que siga al `if`.

La tabla de la figura 4.13 resume el operador `&&`. La tabla muestra las cuatro posibles combinaciones de valores cero (falso) y no cero (verdadero) correspondientes a la expresión1 y a la expresión2. Estos tipos de tablas a menudo se conocen como *tablas de la verdad.* C evaluará todas las expresiones que incluyan operadores relacionales, operadores de igualdad, y operadores lógicos en 0 o en 1. Aunque C defina como 1 un valor verdadero, aceptará como verdadero *cualquier* valor no cero.

Consideremos ahora el operador `||` (OR lógico). Suponga que deseamos asegurarnos, en determinado momento de un programa, que alguna *o* ambas de dos condiciones son verdaderas,

expresión1	expresión2	expresión1 && expresión2
0	0	0
0	no cero	0
no cero	0	0
no cero	no cero	1

Fig. 4.13 Tabla de la verdad para el operador `&&` (AND lógico).

antes de que seleccionemos un cierto camino de ejecución. En este caso utilizaremos el operador `||`, como en el siguiente segmento de programa:

```
if (semesterAverage >= 90 || finalExam >= 90)
   printf("Student grade is A\n");
```

Este enunciado también contiene dos condiciones simples. La condición `semesterAverage >= 90` se evalúa para determinar si el estudiante merece un "A" en el curso, debido a un rendimiento sólido a todo lo largo del semestre. La condición `finalExam >= 90` se evalúa para determinar si el estudiante merece una calificación de "A" en el curso, debido a un rendimiento extraordinario durante el examen final. El enunciado `if`, a continuación analiza la condición combinada

```
semesterAverage >= 90 || finalExam >= 90
```

y califica al estudiante con una "A", si cualquiera o ambas de las condiciones simples es verdadera. Note que el mensaje "`Student grade is A`" no se imprimirá salvo que ambas condiciones simples resulten falsas (cero). La figura 4.14 es una tabla de la verdad correspondiente al operador OR lógico (`||`).

El operador `&&` tiene una precedencia más alta que `||`. Ambos operadores se asocian de izquierda a derecha. Una expresión que contenga `&&` o `||` sólo se evalúa en tanto no se conozca la verdad o la falsedad. Entonces, la evaluación de la condición

```
gender == 1 && age >= 65
```

se detendrá, si `gender` no es igual a `1` (es decir, que toda la expresión es falsa), y sólo continuará si `gender` es igual a `1` (es decir, toda la expresión podría aún ser verdadera si `age >= 65`).

expresión1	expresión2	expresión1 || expresión2
0	0	0
0	no cero	1
no cero	0	1
no cero	no cero	1

Fig. 4.14 Tabla de la verdad para el operador OR lógico (`||`).

Sugerencia de rendimiento 4.3

En expresiones que utilicen el operador `&&`, haga que la condición que más probabilidades tenga de ser falsa sea la que aparezca más a la izquierda. En expresiones que utilicen al operador `||`, haga que la condición que más probabilidades tenga de ser verdadera, sea la condición más a la izquierda. Esto puede reducir el tiempo de ejecución del programa.

C proporciona el signo de `!` (negación lógica) para permitir a un programador el "invertir" el significado de una condición. A diferencia de los operadores `&&` y `||`, que combinan dos condiciones (y que por lo tanto son operadores binarios), el operador de negación lógica tiene como operando una condición (y, por lo tanto, es un operador unario). El operador de negación lógico se coloca antes de una condición, cuando estemos interesados en escoger un camino de ejecución si resulta falsa la condición original (antes del operador de negación lógico), como en el siguiente segmento de programa:

```
if (!(grade == sentinelValue))
printf("The next grade is %f\n", grade);
```

Se requiere el paréntesis que rodea la condición `grade == sentinelValue`, porque el operador de negación lógico tiene una precedencia más alta que el operador de igualdad. La figura 4.15 es una tabla de la verdad correspondiente al operador lógico de negación.

En la mayor parte de los casos, expresando la condición en forma distinta y utilizando un operador relacional apropiado el programador puede evitar el uso de la negación lógica. Por ejemplo, el enunciado precedente pudiera verse también escrito como sigue:

```
if (grade != sentinelValue)
printf("The next grade is %f\n", grade);
```

La tabla en la figura 4.16 muestra la precedencia y asociatividad de los operadores de C estudiados hasta este momento. Los operadores se muestran de arriba abajo en orden decreciente de precedencia.

4.11 Confusión entre los operadores de igualdad (==) y de asignación (=)

Existe un tipo de error que los programadores de C, independiente de su experiencia, tienden a hacer con tanta frecuencia que sentimos que merece una sección por separado. Este error es el intercambio accidental de los operadores == (igualdad) y = (asignación). Lo que hace tan dañinos estos intercambios es el hecho de que, de forma ordinaria, no causan errores de sintaxis. En vez de ello, por lo regular los enunciados que contienen estos errores compilan en forma correcta, y los programas se ejecutan hasta su terminación, generando probablemente resultados incorrectos, debido a errores lógicos en tiempo de ejecución.

expresión	!expresión
0	1
nonzero	0

Fig. 4.15 Tabla de la verdad para el operador ! (negación lógica).

Operadores	Asociatividad	Tipo
`()`	de izquierda a derecha	paréntesis
`++ -- + - ! (tipo)`	de derecha a izquierda	unario
`* / %`	de izquierda a derecha	multiplicativo
`+ -`	de izquierda a derecha	aditivo
`< <= > >=`	de izquierda a derecha	relacional
`== !=`	de izquierda a derecha	igualdad
`&&`	de izquierda a derecha	AND lógico
`||`	de izquierda a derecha	OR lógico
`?:`	de derecha a izquierda	condicional
`= += -= *= /= %=`	de derecha a izquierda	asignación
`,`	de izquierda a derecha	coma

Fig. 4.16 Precedencia y asociatividad de operadores.

Existen dos aspectos de C que causan estos problemas. Uno es que cualquier expresión en C que produzca un valor, puede ser utilizada en la porción decisiva de cualquier estructura de control. Si el valor es 0, se trata como falso, y si el valor no es 0 se trata como verdadero. El segundo es que las asignaciones en C producen un valor, es decir el valor asignado a la variable en el lado izquierdo del operador de asignación. Por ejemplo, suponga que tratamos de escribir

```
if (payCode == 4)
printf("You get a bonus!");
```

pero accidentalmente escribimos

```
if (payCode = 4)
printf("You get a bonus!");
```

El primer enunciado `if` concede de forma correcta una bonificación a la persona cuyo código de nómina es igual a 4. El segundo enunciado `if` —aquel que tiene el error— evalúa la expresión de asignación en la condición `if`. Esta expresión es una asignación simple, cuyo valor es la constante 4. Dado que cualquier valor no cero se interpreta como "verdadero", la condición en este enunciado `if` será siempre verdadera, y la persona recibirá siempre una bonificación ¡independiente de cuál sea el verdadero código de nómina!

Error común de programación 4.8

Utilizar el operador `==` para asignación, o bien utilizar el operador `=` para igualdad.

Los programadores por lo regular escriben condiciones como `x == 7` con el nombre de la variable a la izquierda y la constante a la derecha. Al invertir éstas, de forma tal que la constante esté a la izquierda y el nombre de la variable a la derecha, como en `7 == x`, el programador que de forma accidental remplaza el operador `==` con el `=` quedará protegido por el compilador. El compilador

tratará lo anterior como un error de sintaxis, porque en el lado izquierdo de un enunciado de asignación sólo puede colocarse un nombre de variable. Como mínimo, esto evitará el daño potencial de un error lógico en tiempo de ejecución.

Los nombres de variables se dice que son *lvalues* (por "left values") porque pueden ser utilizados en el lado izquierdo de un operador de asignación. Las constantes se dice que son *rvalues* (por "right values") porque sólo pueden ser utilizadas en el lado derecho de un operador de asignación. Note que los lvalues también pueden ser utilizados como rvalues, pero no al revés.

Práctica sana de programación 4.18

Cuando una expresión de igualdad tiene una variable y una constante como en `x == 1`*, algunos programadores prefieren escribir la expresión con la constante a la izquierda y el nombre variable a la derecha, como protección contra un error lógico, que ocurriría cuando el programador remplaza de forma accidental el operador* `==` *por signo de* `=`*.*

El otro lado de la moneda puede resultar igual de desagradable. Suponga que el programador desea asignar un valor a una variable con un enunciado simple como

```
x = 1;
```

pero en vez de ello escribe

```
x == 1;
```

Aquí, tampoco, hay error de sintaxis. En vez de ello, el compilador simplemente evalúa la expresión condicional. Si `x` es igual a `1`, la condición es verdadera y la expresión devolverá el valor 1. Si `x` no es igual a `1`, la condición es falsa y la expresión regresará el valor 0. Independientemente del valor que se regrese, no ha habido un valor de asignación, por lo que el valor simplemente se perderá, y el valor de `x` se conservará inalterado, causando probablemente un error lógico en tiempo de ejecución. ¡Desafortunadamente, en este problema no tenemos un truco a la mano disponible para auxiliarle!

4.12 Resumen de la programación estructurada

Al igual que los arquitectos diseñan edificios, empleando la sabiduría colectiva de su profesión, así deberían los programadores diseñar los programas. Nuestro campo es más joven que el de la arquitectura, y nuestra sabiduría colectiva es considerablemente menor. Hemos aprendido mucho en apenas cinco décadas. Pero quizás de mayor importancia, hemos aprendido que la programación estructurada produce programas que son más fáciles (que los programas no estructurados) de entender y, por lo tanto, más fáciles de probar, depurar, modificar e inclusive de comprobar su corrección en un sentido matemático.

Los capítulos 3 y 4 se han concentrado en las estructuras de control de C. Se ha presentado cada estructura, su diagrama de flujo y su análisis, por separado, junto con ejemplos. Ahora, resumimos los resultados de los capítulos 3 y 4 e introducimos un conjunto sencillo de reglas para la formación y las propiedades de los programas estructurados.

La figura 4.17 resume las estructuras de control analizadas en los capítulos 3 y 4. Los pequeños círculos se utilizan en la figura para indicar un solo punto de entrada y un solo punto de salida de cada estructura. El conectar de forma arbitraria símbolos individuales de diagramas de flujo puede llevar a programas no estructurados. Por lo tanto, la profesión de la programación ha decidido combinar los símbolos de diagramación de flujo para formar un conjunto limitado de estructuras de control, y elaborar sólo programas estructurados mediante la combinación adecuada de

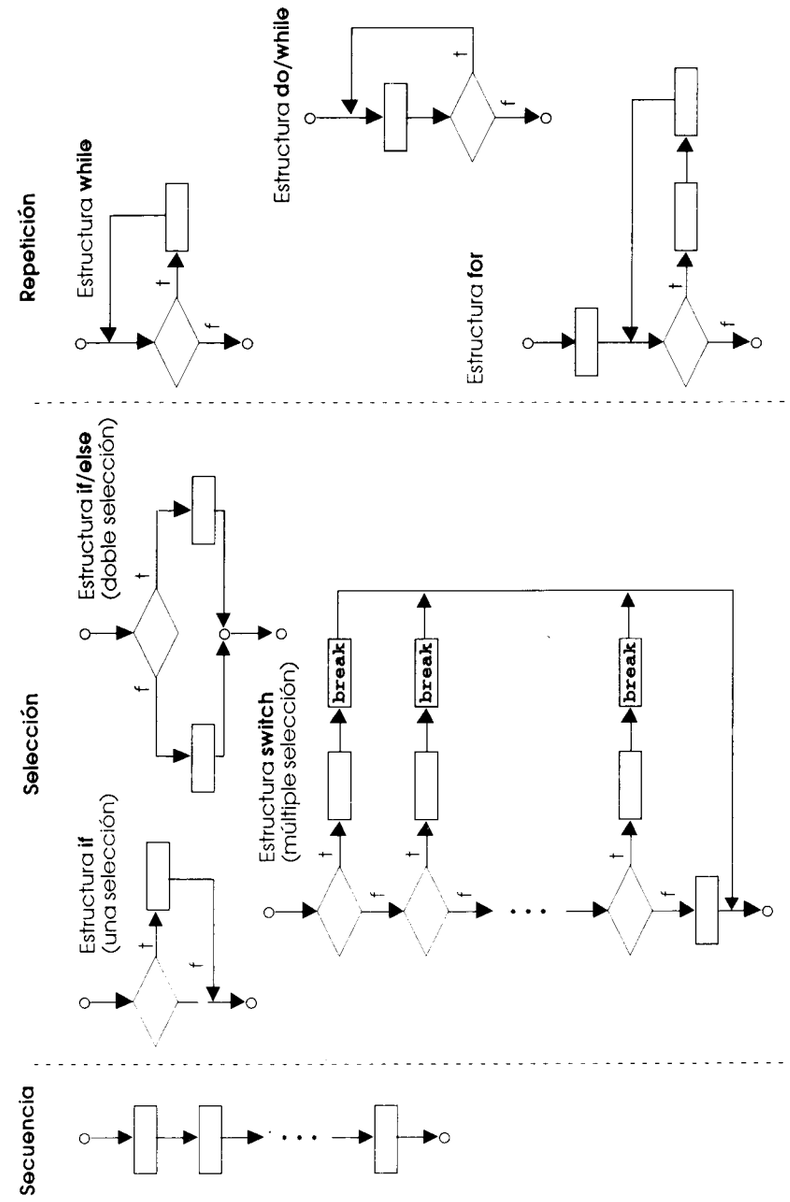


Fig. 4.17 Estructuras de una entrada/una salida de secuencia, de selección y de repetición de C.

estructuras de control en sólo dos formas simples. Por simplicidad, sólo se utilizan estructuras de control de una entrada/una salida sólo hay una forma de entrar y una de salir en cada estructura

de control. Es sencilla la conexión en secuencia de las estructuras de control para formar programas estructurales el punto de salida de una estructura de control se conecta de forma directa al punto de entrada de la siguiente estructura de control, es decir dentro de un programa las estructuras de control son simplemente colocadas una después de la otra; le hemos llamado a esto "apilar estructuras de control". Las reglas para formar programas estructurados, también permite anidar las estructuras de control.

En la figura 4.18 se muestran las reglas para la formación de programas adecuadamente estructurados. Las reglas suponen que el símbolo rectángulo de diagramación de flujo puede ser utilizado para indicar cualquier acción, incluyendo entrada/salida.

La aplicación de las reglas de la figura 4.18 siempre resulta en un diagrama de flujo estructurado con una apariencia nítida y de bloques constructivos. Por ejemplo, la aplicación repetida de la regla 2 al diagrama de flujo más simple, resulta en un diagrama de flujo estructurado, que contiene muchos rectángulos en secuencia (figura 4.20). Note que la regla 2 genera una pila de estructuras de control; por lo tanto llamaremos a la regla 2 *regla de apilamiento.*

La regla 3 se conoce como la *regla de anidamiento.* La aplicación repetida de la regla 3 al diagrama de flujo más simple resulta en un diagrama de flujo con estructuras de control nítida anidadas. Por ejemplo, en la figura 4.21, el rectángulo en el diagrama de flujo más simple, primero se remplaza con una estructura de doble selección (`if/else`). A continuación se vuelve a aplicar la regla tres a ambos rectángulos de la estructura de doble selección, remplazando cada uno de estos rectángulos por estructuras de doble selección. Los recuadros punteados alrededor de cada una de las estructuras de doble selección, representan el rectángulo que fue remplazado.

Reglas para la formación de programas estructurados

1) Empiece con el "diagrama de flujo más simple" (figura 4.19)
2) Cualquier rectángulo (acción) puede ser remplazado por dos rectángulos (acciones) en secuencia.
3) Cualquier rectángulo (acción) puede ser remplazado por cualquier estructura de control (secuencia, `if`, `if/else`, `switch`, `while`, `do/while`, o bien `for`).
4) Las reglas 2 y 3 pueden ser aplicadas tan frecuentemente como se desee y en cualquier orden.

Fig. 4.18 Reglas para la formación de programas estructurados.

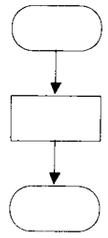


Fig. 4.19 El diagrama de flujo más sencillo.

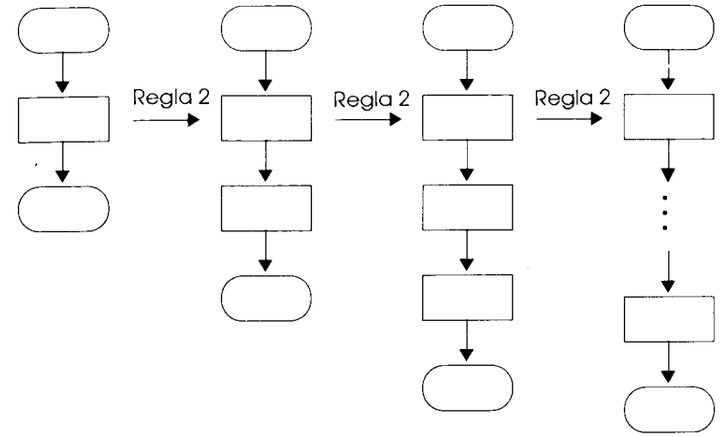


Fig. 4.20 Aplicación repetida de la regla 2 de la figura 4.18 al diagrama de flujo más sencillo.

La regla 4 genera estructuras más grandes, más complejas y más profundamente anidadas. Los diagramas de flujo que resultan de aplicar las reglas de la figura 4.18, constituyen el conjunto de todos los diagramas posibles de flujos estructurados y, por lo tanto, también el conjunto de todos los posibles programas estructurados.

Es debido a la eliminación del enunciado `goto` que estos bloques constructivos nunca se traslapan el uno sobre el otro. La belleza del enfoque estructurado, es que utilizamos un pequeño número de piezas simples de una entrada/una salida, y las ensamblamos en dos formas sencillas. En la figura 4.22 se muestran los tipos de bloques constructivos apilados, que resultan de la aplicación de la regla 2, y los tipos de bloques constructivos anidados que resultan de la aplicación de la regla 3. La figura también muestra el tipo de bloques constructivos superpuestos que no pueden aparecer en diagramas de flujo estructurados (debido a la eliminación del enunciado `goto`).

Si se siguen las reglas de la figura 4.18, no es posible crear un diagrama de flujo no estructurado (como el de la figura 4.23). Si no está seguro de cómo está estructurado un diagrama de flujo particular, aplique las reglas de la figura 4.18 a la inversa, para tratar de reducir dicho diagrama de flujo al diagrama de flujo más simple. Si el diagrama de flujo es reducible al más simple, el diagrama de flujo original está estructurado; de lo contrario no lo está.

La programación estructurada fomenta la simplicidad. Bohm y Jacopini nos han dado como resultado que sólo se requieren tres formas de control:

- Secuencia
- Selección
- Repetición

La secuencia es un asunto trivial. La selección se pone en acción en una de tres formas:

- La estructura `if` (una selección)
- La estructura `if/else` (doble selección)
- La estructura `switch` (múltiple selección)

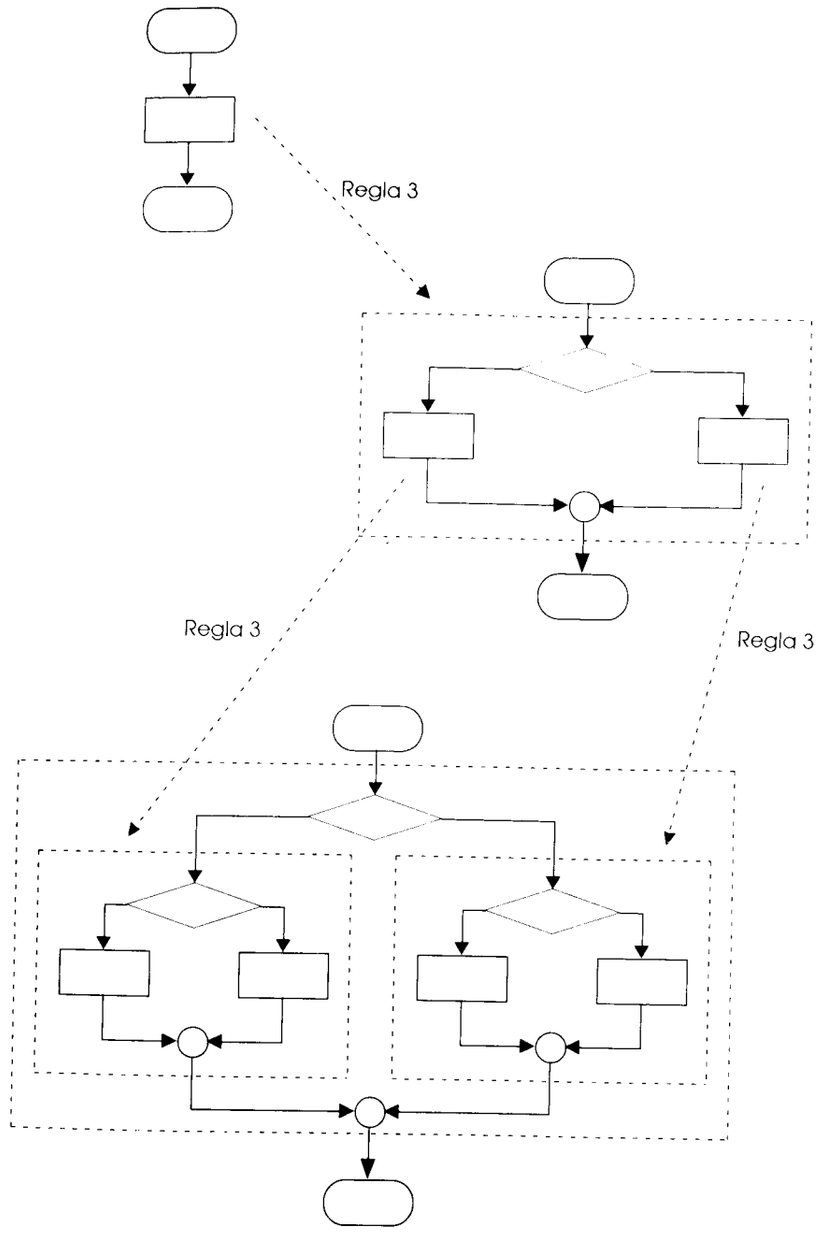


Fig. 4.21 Aplicación de la regla 3 de la figura 4.18 al diagrama de flujo más simple.

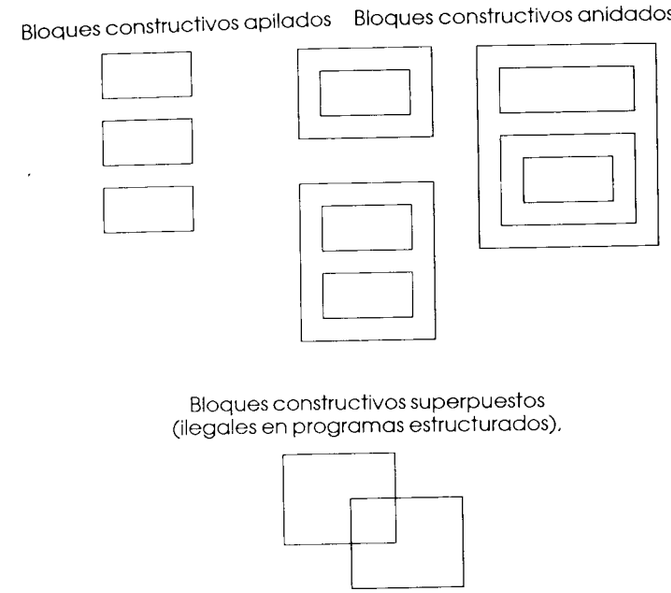


Fig. 4.22 Bloques constructivos apilados, bloques constructivos anidados, y bloques constructivos superpuestos.

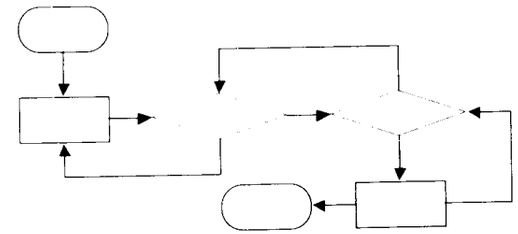


Fig. 4.23 Un diagrama de flujo no estructurado.

De hecho, es fácil comprobar que la estructura simple **`if`** es suficiente para proporcionar cualquier forma de selección —todo lo que se puede hacer con la estructura **`if/else`** y con la estructura **`switch`**, puede ser llevado a cabo con la estructura **`if`**.

La repetición se ejecuta en una de tres formas:

- La estructura **`while`**.
- La estructura **`do/while`**.
- La estructura **`for`**

Es fácil también demostrar que la estructura **while** es suficiente para proporcionar cualquier forma de repetición. Cualquier cosa que pueda ser realizada con la estructura **do/while** y la estructura **for** también puede ser realizada con la estructura **while**.

Combinando estos resultados se llega a que cualquier forma de control, que de alguna manera se requiera en un programa C, puede ser expresado en términos de tres formas de control:

- Secuencia
- Estructura **if** (selección)
- Estructura **while** (repetición)

Y estas estructuras de control sólo pueden ser combinadas de dos formas —apilamiento y anidamiento. De hecho, en realidad la programación estructurada fomenta la simplicidad.

En los capítulos 3 y 4 analizamos cómo conformar programas a partir de estructuras de control, que contengan acciones y decisiones. En el capítulo 5, presentamos otra unidad estructural de programa, llamada la *función*. Aprenderemos a generar programas grandes, mediante la combinación de funciones, mismas que a su vez están compuestas de estructuras de control. También analizaremos cómo el uso de las funciones promueve la reutilización del software.

Resumen

- Un ciclo es un grupo de instrucciones que la computadora ejecuta de forma repetida hasta que queda satisfecha cierta condición de terminación. Dos formas de repetición son la repetición controlada por contador y la repetición controlada por centinela.
- Se utiliza un contador de ciclo para contar el número de veces que un grupo de instrucciones deberá ser repetido. Este contador es incrementado (generalmente en 1), cada vez que es ejecutado el grupo de instrucciones.
- Los valores centinelas en general son utilizados para controlar la repetición, cuando no se conoce con anticipación el número preciso de repeticiones y el ciclo incluye enunciados que deben obtener datos cada vez que éste se ejecuta.
- Se captura un valor centinela después de que todos los elementos válidos de datos hayan sido proporcionados al programa. Los centinelas deben ser seleccionados con cuidado, de forma tal de que no exista posibilidad de confundirlos con elementos válidos de datos.
- La estructura de repetición **for** maneja en forma automática todos los detalles de la repetición controlada por contador. El formato general de la estructura **for** es

```
for (expresión1; expresión2; expresión3)
   enunciado
```

- donde *expresión1* inicia la variable de control del ciclo, *expresión2* es la condición de continuación del ciclo, y la *expresión3* incrementa la variable de control.
- La estructura de repetición **do/while** es similar a la estructura de repetición **while**, pero la estructura **do/while** prueba la condición de continuación del ciclo al final del ciclo, por lo que el cuerpo del ciclo por lo menos se ejecutará una vez. El formato para el enunciado **do/while** es


```
do
   enunciado
while (condición);
```

- El enunciado **break**, cuando se ejecuta en una de la estructuras de repetición (**for**, **while** y **do/while**), causa la salida inmediata de la estructura. La ejecución continúa con el siguiente primer enunciado después del ciclo.
- El enunciado **continue**, cuando se ejecuta en una de las estructuras de repetición (**for**, **while** y **do/while**), salta todos los enunciados siguientes del cuerpo de la estructura, y continúa con la siguiente iteración del ciclo.
- El enunciado **switch** maneja una serie de decisiones en las cuales una variable o expresión particular se prueba para cada uno de los valores que puede asumir, y se toman diferentes acciones. En un enunciado **switch**, cada **case** puede hacer que se ejecuten muchos enunciados. En la mayor parte de los programas, resulta necesario incluir un enunciado **break** después de los enunciados para cada **case**, de lo contrario el programa ejecutará los enunciados en cada **case**, hasta que un enunciado **break** se encuentre o hasta que se alcance el final del enunciado **switch**. Enlistando las etiquetas **case** juntas antes de los enunciados, varios **case** pueden ejecutar los mismos enunciados. La estructura **switch** sólo puede probar expresiones integrales constantes.
- La función **getchar** regresa un carácter del teclado (la entrada estándar) como un entero.
- En sistemas UNIX y en muchos otros, el carácter **EOF** se introduce escribiendo la secuencia

  <return> <ctrl-d>

  en VMS y en DOS, el carácter **EOF** se introduce escribiendo

  <ctrl-z>

- Los operadores lógicos pueden s ser utilizados para formar condiciones complejas, combinando condiciones. Los operadores lógicos son **&&**, **||**, y **!**, significan AND lógico, OR lógico y NOT lógico, (negación) respectivamente.
- Un valor verdadero es cualquier valor no cero.
- Un valor falso es 0 (cero).

Terminología

conjunto de carácter ASCII
cuerpo de un ciclo
break
etiqueta **case**
char
continue
variable de control
repetición controlada por contador
<ctrl-z>
Decremento
caso **default** en **switch**
Repetición definida
double
Estructura de repetición **do/while**
fin de archivo
EOF
ancho de campo
valor final de variable de control

estructura de repetición `for`
función `getchar`
incremento de variable de control
repetición indefinida
ciclo infinito
valor inicial de variable de control
justificación a la izquierda
AND lógico (`&&`)
negación lógica (`!`)
operadores lógicos
OR lógico (`||`)
`long`
condición de continuación de ciclo
variable de control de ciclo
contador de ciclo
lvalue ("left value")
signo menos para justificación a la izquierda
selección múltiple
estructuras de control anidadas
regla de anidación
error de diferencia por uno
función `pow`
estructuras de repetición
<return><ctrl-d>
justificación a la derecha
rvalue ("valor derecho")
`short`
condición simple
estructuras de control de una entrada/ una salida
regla de apilamiento
estructura de selección `switch`
tabla de la verdad
operador unario
estructura de repetición `while`

Errores comunes de programación

4.1 Dado que los valores en punto flotante pueden ser aproximados, el control de contador de ciclos con variables de punto flotante puede dar como resultado valores de contador no precisos y pruebas no exactas de terminación.

4.2 Usar un operador relacional incorrecto o usar un valor final incorrecto de un contador de ciclo, en la condición de una estructura `while` o `for`, puede causar errores de diferencia por uno.

4.3 Usar comas en vez de puntos y coma en un encabezado `for`.

4.4 Colocar un punto y coma inmediatamente a la derecha de un encabezado `for` hace del cuerpo de esta estructura `for` un enunciado vacío. Por lo regular esto es un error lógico

4.5 Olvidar en una estructura `switch` un enunciado `break`, cuando se requiere de uno.

4.6 No procesar los caracteres de nueva línea en la entrada al leer los caracteres uno a la vez, puede ser causa de errores lógicos.

4.7 Se generarán ciclos infinitos en una estructura `while`, `for` o bien `do/while` cuando la condición de continuación de ciclo nunca se convierte en falsa. A fin de evitar lo anterior, asegúrese de que no exista un punto y coma después del encabezado de una estructura `while` o `for`. En un ciclo controlado por contador, asegúrese que la variable de control es incrementada (o decrementada) en el cuerpo del ciclo. En un ciclo controlado por centinela, asegúrese que el valor centinela es eventual introducido en algún momento.

4.8 Utilizar el operador == para asignación, o bien utilizar el operador = para igualdad.

Prácticas sanas de programación

4.1 Controlar el contador de ciclos con valores enteros.

4.2 Hacer sangría en los enunciados en el cuerpo de cada estructura de control.

4.3 Colocar una línea en blanco antes y después de cada estructura de control principal, para que se destaque en el programa.

4.4 Demasiados niveles anidados pueden dificultar la comprensión de un programa. Como regla general, procure evitar el uso de más de tres niveles de sangrías.

4.5 La combinación de espaciamiento vertical antes y después de las estructuras de control y las sangrías de los cuerpos de las estructuras de control dentro de los encabezados de las estructuras de control, le da a los programas una apariencia bidimensional que mejora de manera significativa la legibilidad del programa.

4.6 Usar el valor final en la condición de una estructura `while` o `for` y utilizar el operador relacional <= auxiliará a evitar errores de diferencia por uno. Para un ciclo que se utilice para imprimir los valores del 1 al 10, por ejemplo, la condición de confirmación de ciclo debería ser `counter <= 10` en vez de `counter < 11` o bien counter < 10.

4.7 Coloque sólo expresiones que involucren las variables de control en las secciones de inicialización y de incremento de una estructura `for`. Las manipulaciones de las demás variables deberían de aparecer, ya sea antes del ciclo (si se ejecutan una vez, como los enunciados de incialización) o dentro del cuerpo del ciclo (si se ejecutan una vez en cada repetición, como son los enunciados incrementales o decrementales).

4.8 Aunque el valor de la variable de control puede ser modificado en el cuerpo de un ciclo `for`, ello podría llevarnos a errores sutiles. Lo mejor es no cambiarlo.

4.9 Aunque los enunciados que anteceden a un `for`, y los enunciados del cuerpo de un `for`, a menudo pueden ser combinados en un encabezado `for`, evite hacerlo, porque hace el programa más difícil de leer.

4.10 Limite, si es posible, a una sola línea el tamaño de los encabezados de estructuras de control.

4.11 No utilice variables del tipo `float` o `double` para llevar a cabo cálculos monetarios. La falta de precisión de los números de punto flotante pueden causar errores, que resultarán en valores monetarios incorrectos. En los ejercicios, exploramos la utilización de enteros para la ejecución de cálculos monetarios.

4.12 Proveer un caso `default` en enunciados `switch`. En un `switch` serán ignorados los casos no probados en forma explicita. El caso `default` ayuda a evitar lo anterior, enfocando al programador sobre la necesidad de procesar condiciones excepcionales. Existen situaciones en las cuales no se requiere de procesamiento `default`.

4.13 Aunque las cláusulas `case` y la cláusula de caso `default` en una estructura `switch` pueden ocurrir en cualquier orden, se considera una práctica sana de programación colocar al final la cláusula `default`.

4.14 En una estructura `switch`, cuando la cláusula `default` se enlista al final, el enunciado `break` no es requerido. Pero algunos programadores incluyen este `break`, para fines de claridad y simetría con otros `case`s.

4.15 Recuerde proporcionar capacidades de proceso para los caracteres de nueva línea en la entrada, cuando se procesen caracteres uno a la vez.

4.16 Algunos programadores incluyen siempre llaves en una estructura `do/while`, aún si las llaves no son necesarias. Esto ayuda a eliminar ambigüedad entre la estructura `do/while` con un enunciado, y la estructura `while`.

4.17 Algunos programadores son de la opinión que `break` y `continue` violan las normas de la programación estructurada. Dado que los efectos de estos enunciados pueden ser obtenidos mediante técnicas de programación estructuradas que aprenderemos pronto, estos programadores no utilizan `break` ni `continue`.

4.18 Cuando una expresión de igualdad tiene una variable y una constante como en `x == 1`, algunos programadores prefieren escribir la expresión con la constante a la izquierda y el nombre variable a la derecha, como protección contra un error lógico, que ocurriría cuando el programador remplaza de forma accidental el operador == por signo de =.

Sugerencias de rendimiento

4.1 En situaciones orientadas a rendimiento, donde la memoria está escasa o se requiere de velocidad, pudiera ser deseable utilizar tamaños de enteros más pequeños.

4.2 Los enunciados `break` y `continue`, cuando se útilizan de forma adecuada, se ejecutan con mayor rapidez que las técnicas estructuradas correspondientes que pronto aprenderemos.

4.3 En expresiones que utilicen el operador &&, haga que la condición que más probabilidades tenga de ser falsa sea la que aparezca más a la izquierda. En expresiones que utilicen al operador ||, haga

UNIVERSIDAD DE LAS AMERICAS
FACULTAD DE INGENIERIA

que la condición que más probabilidades tenga de ser verdadera, sea la condición más a la izquierda. Esto puede reducir el tiempo de ejecución del programa.

Sugerencias de portabilidad

4.1 Las combinaciones de teclas para la introducción de **EOF** (fin de archivo) son sistema dependientes.

4.2 Hacer la prueba buscando la constante simbólica **EOF**, en vez de buscar -1, hace más portátiles a los programas. El estándar ANSI indica que **EOF** es un valor entero negativo (pero no necesariamente -1). Por lo tanto, **EOF** pudiera tener diferentes valores en diferentes sistemas

4.3 Dado que de un sistema a otro **int** varía en tamaño, utilice enteros **long**, si espera procesar enteros fuera del rango ±32767, y desearía ser capaz de ejecutar el programa en varios sistemas de cómputo diferentes.

Observación de ingeniería de software

4.1 Existe cierta contraposición entre alcanzar ingeniería de software de calidad y conseguir software de mejor rendimiento. A menudo una de estas metas se consigue a expensas de la otra.

Ejercicios de autoevaluación

4.1 Llene los espacios vacíos en cada uno de los enunciados siguientes.

a) La repetición controlada por contador también se conoce como repetición _____, porque se sabe por anticipado cuántas veces se ejecutará el ciclo.

b) La repetición controlada por centinela también se conoce como _____, porque no se sabe con anticipación cuántas veces se ejecutará el ciclo.

c) En la repetición controlada por contador, se utiliza un _____ para contar el número de veces que deben repetirse un grupo de instrucciones.

d) El enunciado _____, cuando se ejecuta en una estructura de repetición, hace que se ejecute de inmediato la siguiente iteracción del ciclo.

e) El enunciado _____, cuando se ejecuta en una estructura de repetición, o en un **switch**, hace que se salga de inmediato de la estructura.

f) La _____, se utiliza para probar una variable o expresión particular para cada uno de los valores enteros constantes que puede asumir.

4.2 Indique si los siguientes son verdaderos o falsos. Si la respuesta es falsa, explique por qué.

a) El caso **default** se requiere en la estructura de selección **switch**.

b) El enunciado **break** se requiere en el caso **default** de una estructura de selección **switch**.

c) La expresión **(x > y&&a < b)** es verdadera ya sea que **x > y** es verdadero o **a > b** es verdadero.

d) Una expresión que contenga el operador **||** es verdadera si alguno o ambos de sus operandos son verdaderos.

4.3 Escriba un enunciado en C o un conjunto de enunciados en C que ejecuten cada una de las tareas siguientes:

a) Sume los enteros impares entre 1 y 99 mediante una estructura **for**. Suponga que las variables enteras **sum** y **count** han sido declaradas.

b) Imprima el valor **333.546372** en un ancho de campo con **15** caracteres con precisiones de **1**, **2**, **3**, **4**, y **5**. Justifique la salida a la izquierda. ¿Cuáles son los valores que se imprimen?

c) Calcule el valor de **2.5** elevado a la potencia de **3** utilizando la función **pow**. Imprima el resultado con una precisión de **2** en un ancho de campo de **10** posiciones. ¿Cuál es el valor que se imprime?

d) Imprima los enteros de 1 a 20 utilizando el ciclo **while** y la variable contador **x**. Suponga que la variable **x** ha sido declarada, pero no inicializada. Imprima sólo 5 enteros por línea. Sugerencia: Use el cálculo **x % 5**. Cuando el valor de lo anterior es 0, imprima un carácter de nueva línea, de lo contrario imprima un carácter de tabulador.

e) Repita el Ejercicio 4.3 (d) utilizando una estructura **for**.

4.4 Encuentre el error en cada uno de los segmentos de código siguientes y explique cómo corregirlo.

a)
```
x = 1;
while (x <= 10);
   x++;
}
```

b)
```
for (y = .1; y != 1.0; y += .1)
   printf("%f\n", y);
```

c)
```
switch (n) {
   case 1:
      printf("The number is 1\n");
   case 2:
      printf("The number is 2\n");
      break;
   default;
      printf("The number is not 1 or 2\n");
      break;
}
```

d) El siguiente código debería imprimir los valores del 1 al 10.
```
n = 1;
while (n < 10)
   printf("%d ", n++);
```

Respuestas a los ejercicios de autoevaluación

4.1 a) definido. b) indefinido. c) variable de control o contador. d) **continue**. e) **break**. f) estructura de selección **switch**.

4.2 a) Falso. El caso **default** es opcional. Si no se requiere una acción por omisión, entonces no es necesario un caso **default**.

b) Falso. El enunciado **break** se utiliza para salir de la estructura **switch**. El enunciado **break** no es requerido cuando el caso **default** es el último caso.

c) Falso. Al utilizar el operador **&&** ambas expresiones relacionadas deben ser verdaderas, para que toda la expresión resulte verdadera.

d) Verdadero.

4.3 a)
```
sum = 0;
for (count = 1; count <= 99; count +=2)
sum += count;
```

b)
```
printf("%-15.1f\n", 333.546372);  /* prints 333.5     */
printf("%-15.2f\n", 333.546372);  /* prints 333.55    */
printf("%-15.3f\n", 333.546372);  /* prints 333.546   */
printf("%-15.4f\n", 333.546372);  /* prints 333.5464  */
printf("%-15.5f\n", 333.546372);  /* prints 333.54637 */
```

c)
```
printf("%10.2f\n", pow(2.5, 3));  /* prints 15.63     */
```

d)
```
x = 1;
while (x <= 20) {
   printf("%d", x);
```

```
      if (x % 5 == 0)
         printf("\n");
      else
         printf("\t");
      x++;
   }
```

o bien

```
x = 1;
while (x <= 20)
   if (x % 5 == 0)
      printf("%d\n", x++);
   else
      printf("%d\t", x++);
```

o bien

```
x = 0;
while (++x  <= 20)
   if (x % 5 == 0)
      printf("%d\n", x);
   else
      printf("%d\t", x);
```

e)
```
for (x = 1; x <= 20; x++) {
   printf("%d", x);
   if (x % 5 == 0)
      printf("\n");
   else
      printf("\t");
}
```

o bien

```
for (x = 1; x <= 20; x++)
   if (x % 5 == 0)
      printf("%d\n", x);
   else
      printf("%d\t", x);
```

4.4 a) Error: el punto y coma después del encabezado `while` genera un ciclo infinito.
Corrección: Remplace el punto y coma por una llave izquierda `{`, o bien elimine tanto el `;` como la llave derecha `}`.

b) Error: usar un número de punto flotante para control de una estructura de repetición `for`.
Corrección: utilice un entero, y ejecute el cálculo adecuado a fin de obtener los valores que desea.

```
for (y = 1; y != 10; y++)
   printf("%f\n", (float) y / 10);
```

c) Error: el enunciado `break` faltante en los enunciados para el primer `case`.
Corrección: añada un enunciado `break` al final de los enunciados para el primer `case`. Advierta que esto no es necesariamente un error, si el programador desea que `case 2:` se ejecute siempre que se ejecute `case 1:`.

d) Error: Operador relacional incorrecto, utilizado en la condición de continuación de repetición `while`. Corrección: Utilice `<=` en vez de `<` que.

Ejercicios

4.5 Encuentre el error en cada uno de los siguientes (Nota: pudiera haber más de un solo error).

a)
```
For (x = 100, x < = 1, x++)
   printf("%d\n", x);
```

b) El código siguiente debería imprimir si el entero dado es impar o par:
```
switch (value % 2) {
   case 0:
      printf("Even integer\n");
   case 1:
      printf("Odd integer\n");
}
```

c) El siguiente código debe introducir un entero y un carácter, y a continuación imprimirlos. Suponga que el usuario escribe como entrada `100 A`.
```
scanf("%d", &intVal);
charVal = getchar();
printf("Integer: %d\nCharacter: %c\n", intVal, charVal);
```

d)
```
for (x = .000001; x <= .0001; x += .000001)
   printf("%.7f\n", x);
```

e) El código siguiente deberá dar como salida los enteros impares de 999 hasta 1:
```
for (x = 999; x >= 1; x += 2)
   printf("%d\n", x);
```

f) El código siguiente debe dar como salida los enteros pares desde 2 hasta 100:
```
counter = 2;

Do {
   if (counter % 2 == 0)
      printf("%d\n", counter);

   counter += 2;
      } While (counter  100);
```

g) El código siguiente deberá sumar los enteros de 100 a 150 (suponga que `total` está inicializado a 0):
```
for (x = 100; x <= 150; x++);
   total += x;
```

4.6 Diga qué valores de la variable de control `x` se imprimen para cada uno de los enunciados `for` siguientes:

a)
```
for (x = 2; x <= 13; x += 2)
   printf("%d\n", x);
```

b)
```
for (x = 5; x <= 22; x += 7)
   printf("%d\n", x);
```

c)
```
for (x = 3; x <= 15; x += 3)
   printf("%d\n", x);
```

d)
```
for (x = 1; x <= 5; x += 7)
   printf("%d\n", x);
```

e)
```
for (x = 12; x >= 2; x -= 3)
printf("%d\n", x);
```

4.7 Escriba enunciados `for` que impriman las siguientes secuencias de valores:

a) 1, 2, 3, 4, 5, 6, 7
b) 3, 8, 13, 18, 23
c) 20, 14, 8, 2, -4, -10
d) 19, 27, 35, 43, 51

4.8 ¿Qué es lo que ejecuta el siguiente programa?

```
#include <stdio.h>

main()
{
   int i, j, x, y;

   printf("Enter integers in the range 1-20: ");
   scanf("%d%d," &x, &y);

   for (i = 1; i <= y; i++) {

      for (j = 1; j <= x; j++)
         printf("@");

      printf("\n");
   }

   return 0;
}
```

4.9 Escriba un programa que sume una secuencia de enteros. Suponga que el primer entero leído con `scanf` especifica el número de valores que faltan de introducir. Su programa deberá leer un valor cada vez que `scanf` sea ejecutado. Una secuencia de entrada típica pudiera ser

```
5 100 200 300 400 500
```

donde `5` indica que los `5` valores subsiguientes deberán de ser sumados.

4.10 Escriba un programa que calcule e imprima el promedio de varios enteros. Suponga que el último valor leído mediante `scanf` es el centinela `9999`. Una secuencia típica de entrada pudiera ser

```
10 8 11 7 9 9999
```

indicando que debe calcularse el promedio de todos los valores que preceden a `9999`.

4.11 Escriba un programa que localice el más pequeño de varios enteros. Suponga que el primer valor leído especifica el número de valores que restan.

4.12 Escriba un programa que calcule e imprima la suma de los enteros pares del 2 al 30.

4.13 Escriba un programa que calcule e imprima el producto de los enteros impares del 1 al 15.

4.14 La función *factorial* se utiliza con frecuencia en problemas de probabilidades. El factorial de un entero positivo *n* (escrito *n!* y dicho como "factorial de n") es igual al producto de los enteros positivos del 1 hasta el *n*. Escriba un programa que evalúe los factoriales de los enteros del 1 al 5. Imprima el resultado en forma tabular. ¿Qué podría impedirle que se calculara el factorial de 20?.

4.15 Modifique el programa de interés compuesto de la Sección 4.6 para repetir sus pasos para tasas de interés de 5 %, 6 %, 7 %, 8 %, 9 % y 10 por ciento. Utilice un ciclo `for` para variar la tasa de interés.

4.16 Escriba un programa que imprima los siguientes patrones por separado, uno debajo del siguiente. Utilice ciclos `for` para generar los patrones. Todos los asteriscos (`*`), deberán ser impresos por un solo

enunciado `printf` de la forma `printf("*");` (esto hace que los asteriscos se impriman uno al lado del otro). *Sugerencia*: los últimos dos patrones requieren que cada línea empiece con un número correcto de espacios en blanco.

```
(A)          (B)          (C)          (D)

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

4.17 La cobranza se hace cada vez más difícil durante periodos de recesión, por lo que las empresas pudieran restringir sus límites de crédito para impedir que sus cuentas por cobrar (el dinero que se les debe) crezca demasiado. En respuesta a una recesión prolongada, una compañía ha reducido a la mitad los límites de crédito de sus clientes. Por lo tanto, si un cliente particular tenía un límite de crédito de $2000, el límite de crédito de este cliente es ahora de $1000. Si un cliente tenía un límite de crédito de $5000, el límite de crédito de este cliente es ahora de $2500. Escriba una programa que analice el estado de crédito de tres clientes de esta empresa. Para cada cliente se le dará:

1. El número de cuenta del cliente.
2. El límite de crédito del cliente, antes de la recesión.
3. EL saldo actual del cliente (es decir, la cantidad que el cliente le debe a la empresa).

Su programa deberá calcular e imprimir el nuevo límite de crédito para cada cliente, y deberá determinar (e imprimir) qué clientes tienen saldos actuales que exceden sus nuevos límites de crédito.

4.18 Una aplicación interesante de las computadoras, es dibujar gráficos y gráficas de barra (a veces llamadas "histogramas"). Escriba un programa que lea cinco números (cada uno de ellos entre 1 y 30). Para cada uno de los números leídos, su programa deberá imprimir una línea, conteniendo dicho número en asteriscos adyacentes. Por ejemplo, si su programa lee el número 7, deberá imprimir `*******`.

4.19 Una empresa de ventas por correo vende cinco productos distintos, cuyos precios de menudeo se muestran en la tabla siguiente:

Número de producto	Precio al menudeo
1	$ 2.98
2	4.50
3	9.98
4	4.49
5	6.87

Escriba un programa que lea una serie de pares de números, como sigue:

1. Número del producto.
2. Cantidad vendida en un día.

Su programa deberá utilizar un enunciado `switch` para ayudar a determinar el precio de menudeo de cada producto. Su programa deberá calcular y desplegar el valor total de menudeo, de todos los productos vendidos la semana pasada.

4.20 Complete las tablas de la verdad siguiente, llenando cada uno de los espacios en blanco con un 0 o un 1.

Condición1	Condición2	Condición1 && Condición2
0	0	0
0	no cero	0
no cero	0	____
no cero	no cero	____

Condición1	Condición2	Condición1 || Condición2
0	0	0
0	no cero	1
no cero	0	____
no cero	no cero	____

Condición1	!Condición1
0	1
no cero	____

4.21 Vuelva a escribir el programa de la figura 4.2, de tal forma que la inicialización de la variable `counter` sea hecha en la declaración, en vez de en la estructura `for`.

4.22 Modifique el programa de la figura 4.7, de tal forma que calcule la calificación promedio de la clase.

4.23 Modifique el programa de la figura 4.6, de tal forma que sólo utilice enteros para calcular el interés compuesto. (*Sugerencia*: trate todas las cantidades monetarias como números enteros de centavos. A continuación "divida" el resultado en su porción dólares y en su porción en centavos, mediante el uso de las operaciones de división y de módulo respectivamente. Inserte un punto).

4.24 Suponga que `i = 1, j = 2, k = 3, m = 2`. ¿Qué es lo que imprime cada uno de los enunciados siguientes?.

a) `printf("%d", i == 1);`
b) `printf("%d", j == 3);`

c) `printf("%d", i >= 1 && j > 4);`
d) `printf("%d", m <= 99 && k < m);`
e) `printf("%d", j >= i || k == m);`
f) `printf("%d", k + m < j || 3 - j >= k);`
g) `printf("%d", !m);`
h) `printf("%d", !(j - m));`
i) `printf("%d", !(k < m));`
j) `printf("%d", !(j > k));`

4.25 Imprima la tabla de equivalentes decimales, binarios, octales y hexadecimales. Si no está familiarizado con estos sistemas numéricos, lea primero el Apéndice E, si desea tratar de resolver este ejercicio.

4.26 Calcule el valor de π a partir de la serie infinita

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \ldots$$

Imprima una tabla que muestre el valor de π aproximado a un término de esta serie, a dos, a tres, etcétera. ¿Cuántos términos de esta serie tendrá que utilizar antes de que empiece a tener 3.14?, ¿3.141?, ¿3.1415? ¿3.14159?

4.27 (*Ternas pitagóricas*). Un triángulo rectángulo puede tener lados que sean todos enteros. El conjunto de tres valores enteros para los lados de un triángulo rectángulo se conoce como una terna pitagórica. Estos tres lados deben de satisfacer la relación de que la suma de los cuadrados de dos de los lados es igual al cuadrado de la hipotenusa. Encuentre todos las ternas pitagóricas para lado1, lado2 e hipotenusa, todos ellos no mayores de 500. Utilice un ciclo `for` de triple anidamiento, que pruebe todas las posibilidades. Este es un ejemplo de computación de "fuerza bruta". Para muchas personas no es de forma estética agradable. Pero existen muchas razones por las cuales estas técnicas son de importancia. Primero, con la potencia de las computadoras incrementándose a una velocidad tan fenomenal, soluciones que con la tecnología de sólo hace unos años habrían tomado años e inclusive siglos de tiempo de computadora para producir, pueden ser hoy producidas en horas, minutos o inclusive segundos. ¡Chips microprocesadores recientes pueden procesar más de 100 millones de instrucciones por segundo! Y en los años 90's probablemente aparecerán chips de microprocesador de billones de instrucciones por segundo. Segundo, como aprenderá en cursos de ciencias de computación más avanzadas, hay gran número de problemas interesantes para los cuales no existe un enfoque algorítmico conocido salvo el uso de simple fuerza bruta. En este libro investigamos muchos tipos de metodologías para la resolución de problemas. Consideraremos muchos enfoques de fuerza bruta para la solución de varios problemas interesantes.

4.28 Una empresa paga a sus empleados como gerentes (que reciben un salario semanal fijo), trabajadores horarios (quienes reciben un salario horario fijo por las primeras 40 horas de trabajo, y "tiempo y medio", es decir, 1.5 veces su sueldo horario, para las horas extras trabajadas), trabajadores a comisión (quienes reciben $250 más 5.7% de sus ventas semanales brutas), o trabajadores a destajo (quienes reciben una cantidad fija de dinero por cada una de las piezas que produce cada trabajador a destajo de esta empresa que trabaja sólo un tipo de piezas). Escriba un programa para calcular la nómina semanal de cada empleado. Usted no sabe por anticipado el número de empleados. Cada tipo de empleado tiene su propio código de nómina: los gerentes tienen un código de nómina 1, los trabajadores horario tienen un código 2, los trabajadores a comisión el código 3 y los trabajadores a destajo el código 4. Utilice un `switch` para calcular la nómina de cada empleado, basado en el código de nómina de dicho empleado. Dentro del `switch`, solicite al usuario (es decir, al oficinista de nóminas) que escriba los hechos apropiados que requiere su programa, para calcular la paga o nómina de cada empleado, basado en el código de nómina de cada uno de ellos.

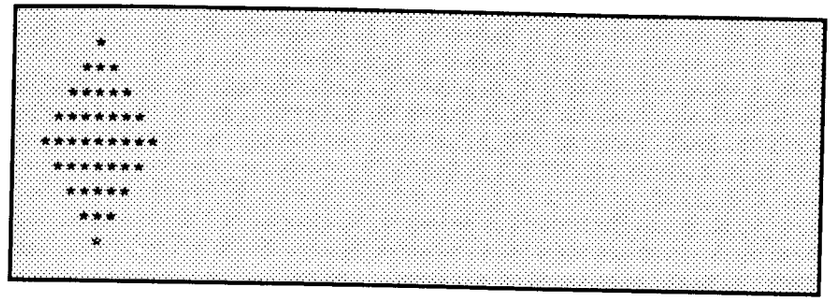
4.29 (*Leyes de De Morgan*). En este capítulo, hemos analizado los operadores lógicos `&&`, `||`, y `!`. Las leyes de De Morgan a veces pueden hacer que sea más conveniente para nosotros expresar una expresión lógica. Estas leyes dicen que la expresión `!(`*condición1* `&&` *condición2*`)` es lógicamente equivalente a la expresión `(!`*condición1* `||` `!`*condición2*`)`. También, la expresión `!(`*condición1* `||` `!`*condición2*`)` es

lógicamente equivalente a la expresión (!*condición1* && !*condición2*). Utilice las leyes de De Morgan para escribir expresiones equivalentes para cada uno de los siguientes, y a continuación escriba un programa para mostrar que tanto la expresión original como la nueva, en cada caso, son equivalentes:

a) `!(x < 5) && !(y >= 7)`
b) `!(a == b) || !(g != 5)`
c) `!((x <= 8) && (Y > 4))`
d) `!((i > 4( || (j <= 6))`

4.30 Vuelva a escribir el programa de la figura 4.7, remplazando el enunciado **`switch`** con un enunciado **`if/else`** anidado; tenga cuidado de manejar el caso **`default`** de forma adecuada. A continuación vuelva a escribir esta nueva versión, remplazando el enunciado **`if/else`** anidado con un serie de enunciados **`if`**; aquí, también, tenga cuidado cómo manejar correctamente el caso **`default`** (esto es más difícil que en el caso de la versión **`if/else`** anidada). Este ejercicio demuestra que **`switch`** es una conveniencia y que cualquier enunciado **`switch`** puede ser escrito utilizando sólo enunciados de una sola selección.

4.31 Escriba un programa que imprima la forma en diamante siguiente. Puede usted utilizar enunciados **`printf`** que impriman ya sea un asterisco (*), o un espacio en blanco. Maximice su utilización de repeticiones (utilizando estructuras **`for`** anidadas), y minimice el número de enunciados **`printf`**.



4.32 Modifique el programa que escribió en el Ejercicio 4.31 para leer un número impar del rango 1 al 19, a fin de especificar el número de renglones del diamante. Su programa a continuación deberá desplegar un diamante del tamaño apropiado.

4.33 Si está familiarizado con los números romanos, escriba un programa que imprima una tabla de todos los equivalentes de números romanos con números decimales en el rango del 1 al 100.

4.34 Escriba un programa que imprima una tabla de los equivalentes binarios, octal y hexadecimal de los números decimales en el rango 1 hasta 256. Si no está familiarizado con estos sistemas numéricos, lea primero el Apéndice E, si desea tratar de intentar hacer este ejercicio.

4.35 Describa el proceso que utilizaría para remplazar el ciclo **`do/while`** con un ciclo equivalente **`while`**. ¿Qué problema ocurre cuando usted trata de remplazar el ciclo **`while`** con un ciclo equivalente **`do/while`**? Suponga que se le ha dicho que debe eliminar un ciclo **`while`** y remplazarlo con un **`do/while`**. ¿Qué estructura de control adicional necesitaría utilizar, y cómo la utilizaría, para asegurarse que el programa resultante se comporta igual que el original?.

4.36 Escriba un programa que introduzca el año en el rango 1994 hasta 1999 y utilice una repetición de ciclo **`for`** para producir un calendario condensado e impreso de forma nítida. Tenga cuidado con los años bisiestos.

4.37 Una crítica del enunciado **`break`** y del enunciado **`continue`** es que cada uno de ellos no es estructurado. De hecho los enunciados **`break`** y **`continue`** siempre pueden ser remplazados por enunciados estructurados, aunque el hacerlo puede resultar un poco torpe. Describa en general cómo eliminaría cualquier enunciado **`break`** de un ciclo de un programa, y lo remplazaría con algún equivalente estructurado

(*Sugerencia*: el enunciado **`break`** deja un ciclo desde dentro del cuerpo del ciclo). La otra forma de salir es haciendo fallar la prueba de continuación de ciclo. Considere utilizar en la prueba de continuación de ciclo, una segunda prueba que indique (salida temprana debida a una condición "de salida"). Utilice la técnica que desarrolló aquí para eliminar el enunciado 'break' del programa de la figura 4.11.

4.38 ¿Qué es lo que hace el siguiente segmento de programa?.

```
for (i = 1; i <= 5; i++) {
    for (j = 1; j <= 3; j++) {
        for (k = 1, k <= 4; k++)
            printf("*");
            Printf("\n");

}
        printf("\n");

}
```

4.39 Describa en general cómo eliminaría cualquier enunciado **`continue`** de un ciclo de un programa y remplazar dicho enunciado con algún equivalente estructurado. Use las técnicas que desarrolló aquí para eliminar el enunciado **`continue`** del programa de la figura 4.12.

4.40 Describa en general cómo eliminaría los enunciados **`break`** de una estructura **`switch`** y los remplazaría con equivalentes estructurados. Utilice la técnica (quizás algo torpe) que desarrolló aquí para eliminar los enunciados **`break`** del programa de la figura 4.7.

5

Funciones

Objetivos

- Comprender cómo construir programas en forma modular partiendo de pequeñas partes conocidas como funciones.
- Presentar las funciones matemáticas comunes disponibles en la biblioteca estándar de C.
- Ser capaz de crear funciones nuevas.
- Comprender los mecanismos utilizados para pasar información de función a función.
- Introducir técnicas de simulación utilizando generación de números aleatorios.
- Comprender cómo escribir y utilizar funciones que se llamen a sí mismas.

La forma sigue siempre a la función.
Louis Henri Sullivan

E pluribus unum. (uno formado de muchos).
Virgilo

¡Oh! llama al ayer, pídele al tiempo que vuelva.
William Shakespeare
Richard II

Llámame Ismael.
Herman Melville
Moby Dick

Cuando me llames así, sonríe.
Owen Wister

Sinopsis

5.1 Introducción

La mayor parte de los programas de cómputo que resuelven problemas de la vida real, son mucho mayores que los programas presentados en los primeros pocos capítulos. La experiencia ha mostrado que la mejor forma de desarrollar y mantener un programa grande es construirlo a partir de piezas menores o *módulos*, siendo cada una de ellas más fácil de manipular que el programa original. Esta técnica se conoce como *divide y vencerás*. Este capítulo describe aquellas características del lenguaje C que facilitan el diseño, implantación, operación y mantenimiento de programas grandes.

5.2 Módulos de programa en C

En C los módulos se llaman *funciones*. Por lo general en C, los programas se escriben combinando nuevas funciones que el programador escribe, con funciones "preempacadas", disponibles en la *biblioteca estándar de C*. En este capítulo analizarémos ambos tipos de funciones. La biblioteca estándar de C contiene una amplia colección de funciones para llevar a cabo cálculos matemáticos

comunes, manipulaciones con cadenas, manipulaciones con caracteres, entrada/salida, y muchas otras operaciones útiles. Esto facilita la tarea del programador, porque estas funciones proporcionan muchas de las capacidades que los programadores requieren.

Práctica sana de programación 5.1

Familiarícese con la amplia colección de funciones de la biblioteca estándar ANSI C.

Observación de ingeniería de software 5.1

Evite reinventar la rueda. Siempre que sea posible, utilice funciones estándar de biblioteca ANSI C, en vez de escribir nuevas funciones. Esto reduce el tiempo de desarrollo del programa.

Sugerencia de portabilidad 5.1

Utilizar funciones de la biblioteca estándar ANSI C auxilia a que los programas sean más portátiles.

Aunque técnicamente las funciones estándar de biblioteca no forman parte del lenguaje C, de forma invariable son proporcionadas junto con los sistemas ANSI C. Las funciones **`printf`**, **`scanf`**, y **`pow`**, que hemos utilizado en capítulos anteriores, son funciones estándar de biblioteca.

El programador puede escribir funciones para definir tareas específicas, que puedan ser utilizadas en muchos puntos de un programa. Estas a veces se conocen como *funciones definidas por el programador*. Los enunciados que definen la función se escriben sólo una vez, quedando los enunciados ocultos de otras funciones.

Las funciones se *invocan* mediante una *llamada de función*. La llamada de función especifica el nombre de la misma y proporciona información (en forma de *argumentos*), que la función llamada necesita a fin de llevar a cabo su tarea designada. Una analogía común de lo anterior es la administración de tipo jerárquico. Un jefe (la *función que llama* o *el llamador*) le pide al trabajador (la *función llamada*) que ejecute una tarea y cuando ésta haya sido efectuada, informe. Por ejemplo, una función jefe, que desea mostrar información en pantalla, llama la función trabajador **`printf`** para que ejecute esa tarea, y a continuación **`printf`** despliega la información y cuando ha terminado su tarea informa, o *regresa* a la función llamadora. La función jefe no sabe cómo la función trabajador ejecuta sus tareas designadas. El trabajador pudiera tener que llamar a otras funciones trabajador, y el jefe no se dará cuenta de ello. Pronto veremos cómo este "ocultamiento" de los detalles de puesta en práctica promueve una buena ingeniería de software. En la figura 5.1 se muestra la función **`main`**, comunicándose con varias funciones trabajador, de una forma jerárquica. Advierta que **`worker1`** funciona como función jefe hacia **`worker4`** y **`worker5`**. Las relaciones entre las funciones pudieran ser distintas en estructura jerárquica a la que se muestra en esta figura.

5.3 Funciones matemáticas de biblioteca

Las funciones matemáticas de biblioteca le permiten al programador ejecutar ciertos cálculos matemáticos comunes. Para iniciarnos en el concepto de las funciones utilizamos aquí varias funciones matemáticas de biblioteca. Más adelante en el libro, analizaremos muchas de las demás funciones de la biblioteca estándar C. Una lista completa de las funciones estándar de biblioteca de C aparece en el Apéndice B.

Las funciones se utilizan normalmente en un programa, escribiendo el nombre de la función, seguido por un paréntesis izquierdo y a continuación por el *argumento* (o una lista de argumentos separados por comas), de la función seguida por un paréntesis derecho. Por ejemplo, un programador que desea calcular e imprimir la raíz cuadrada de **`900.0`** pudiera escribir

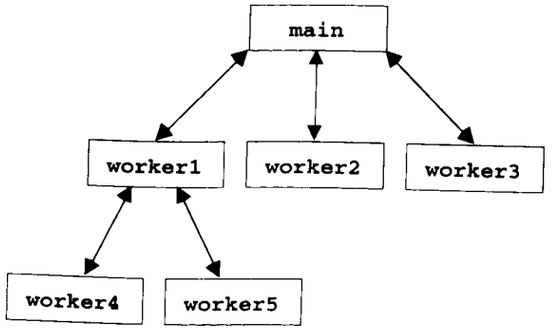


Fig. 5.1 Relación jerárquica función jefe/función trabajadora.

```
printf("%.2f", sqrt(900.0));
```

Cuando este enunciado se ejecuta, es llamada la función matemática de biblioteca **sqrt**, a fin de que calcule la raíz cuadrada del número contenido entre paréntesis (**900.0**). El número **900.0** es el argumento de la función **sqrt**. El enunciado anterior imprimiría **30.00**. La función **sqrt** toma un argumento del tipo **double** y regresa un resultado del tipo **double**. Todas las funciones de la biblioteca de matemáticas devuelven el tipo de datos **double**.

Práctica sana de programación 5.2

Incluya el archivo de cabecera de matemáticas utilizando la directriz de preprocesador `#include <math.h>` cuando esté utilizando funciones de la biblioteca de matemáticas.

Error común de programación 5.1

Olvidar incluir el archivo de cabecera matemático, al usar las funciones matemáticas de biblioteca, puede causar resultados extraños.

Los argumentos de las funciones pueden ser constantes, variables o expresiones. Si **c1 = 13.0**, **d = 3.0**, y **f = 4.0**, entonces el enunciado

```
printf("%.2f", sqrt(c1 + d * f));
```

calcula e imprime la raíz cuadrada de **13.0 + 3.0 * 4.0 = 25.0**, es decir **5.00**.

Algunas funciones matemáticas de biblioteca de C se resumen en la figura 5.2. En la figura, las variables **x** y **y** son del tipo **double**.

5.4 Funciones

Las funciones permiten al programador modularizar un programa. Todas las variables declaradas en las definiciones de función son *variables locales* —son conocidas sólo en la función en la cual están definidas. La mayor parte de las funciones tienen una lista de *parámetros*. Los parámetros proporcionan la forma de comunicar información entre funciones. Los parámetros de una función también son variables locales.

Función	Descripción	Ejemplo
`sqrt(x)`	raíz cuadrada de *x*	`sqrt(900.0)` es 30.0 `sqrt(9.0)` es 3.0
`exp(x)`	función exponencial e^x	`exp(1.0)` es 2.718282 `exp(2.0)` es 7.389056
`log(x)`	logaritmo natural de *x* (de base *e*)	`log(2.718282)` es 1.0 `log(7.389056)` es 2.0
`Log10(x)`	logaritmo de *x* (de base 10)	`log10(1.0)` es 0.0 `log10(10.0)` es 1.0 `log10(100.0)` es 2.0
`fabs(x)`	valor absoluto de *x*.	si `x > 0` entonces `fabs(x)` es `x` si `x = 0` entonces `fabs(x)` es 0.0 si `x < 0` entonces `fabs(x)` es `-x`
`ceil(x)`	redondea a *x* al entero más pequeño que no sea menor que *x*	`ceil(9.2)` es 10.0 `ceil(-9.8)` es 9.0
`floor(x)`	redondea a *x* al entero más grande no mayor que *x*	`floor(9.2)` es 9.0 `floor(-9.8)` es -10.0
`pow(x, y)`	*x* elevado a la potencia *y* (x^y)	`pow(2.7)` es 128.0 `pow(9, .5)` es 3.0
`fmod(x, y)`	residuo de *x*/*y* como un número de punto flotante	`fmod(13.657, 2.333)` es 1.992
`sin(x)`	seno trigonométrico de *x* (*x* en radianes)	`sin(0.0)` es 0.0
`cos(x)`	coseno trigonométrico de *x* (*x* en radianes)	`cos (0.0)` es 1.0
tan(x)	tangente trigonométrica de *x* (*x* en radianes)	`tan(0.0)` es 0.0

Fig. 5.2 Uso común de las funciones matemáticas de biblioteca.

Observación de ingeniería de software 5.2

En programas que contengan muchas funciones, **main** *deberá de ser organizada como un grupo de llamadas a funciones que ejecuten la mayor parte del trabajo del programa.*

Existen varios intereses que dan motivo a la "funcionalización" de un programa. El enfoque de divide y vencerás hace que el desarrollo del programa sea más manipulable. Otra motivación es la *reutilización del software* —el uso de funciones existentes, como bloques constructivos, para crear nuevos programas. La reutilización del software es un factor primordial en el concepto de la programación orientada a objetos. Con buena identificación y definición de funciones, los programas pueden ser creados partiendo de funciones estandarizadas, que ejecuten tareas específicas, en vez de ser elaborados usando código personalizado. Esta técnica se conoce como *abstracción.*

Cada vez que escribimos programas incluyendo funciones estándar de biblioteca, como **printf**, **scanf** y **pow**, utilizamos la abstracción. Un tercer motivo es evitar un programa repetición de código. Empaquetar código en forma de función, permite que se ejecute dicho código, desde distintas posiciones en un programa, simplemente llamando dicha función.

Observación de ingeniería de software 5.3

Cada función debería limitarse a ejecutar una tarea sencilla y bien definida, y el nombre de la función debería expresar de forma clara dicha tarea. Ello facilitaría la abstracción y promovería la reutilización del software.

Observación de ingeniería software 5.4

Si no puede elegir un nombre conciso, que exprese lo que la función ejecuta, es probable que su función este intentando ejecutar demasiadas tareas diversas. A menudo es mejor dividir dicha función en varias funciones más pequeñas.

5.5 Definiciones de función

Cada programa que hemos presentado ha consistido de una función llamada **main**, que para llevar a cabo sus tareas ha llamado funciones estándar de biblioteca. Veremos ahora, como los programadores escriben sus propias funciones personalizadas.

Considere un programa que utiliza una función **square** para calcular los cuadrados de los enteros del 1 al 10 (figura 5.3).

```
/* A programmer-defined square function */
#include <stdio.h>

int square(int);    /* function prototype */

main()
{
   int x;

   for (x = 1; x <= 10; x++)
      printf("%d  ", square(x));

   printf("\n");
   return 0;
}

/* Function definition */
int square(int y)
{
   return y * y;
}
```

```
 1  4  9  16  25  36  49  64  81  100
```

Fig. 5.3 Uso de una función definida-programador.

Práctica sana de programación 5.3

Coloque una línea en blanco entre definiciones de función, para separarlas y para mejorar la legibilidad del programa.

La función **square** es *invocada*, o bien *llamada* en **main** dentro del enunciado **printf**

```
printf("%d ", square(x));
```

La función **square** recibe una copia del valor de **x** en el *parámetro* **y**. A continuación **square** calcula **y * y**. El resultado se regresa a la función **printf** en **main** donde se llamó a **square**, y **printf** despliega el resultado. Este proceso se repite diez veces utilizando la estructura de repetición **for**.

La definición de **square** muestra que **square** espera un parámetro entero **y**. La palabra reservada **int**, que precede al nombre de la función indica que **square** regresa o devuelve un resultado entero. El enunciado **return** en **square** pasa el resultado del cálculo de regreso a la función llamadora.

La línea

```
int square(int);
```

es un *prototipo de función*. El **int** dentro del paréntesis informa al compilador que **square** espera recibir del llamador un valor entero. El **int** a la izquierda del nombre de la función **square** le informa al compilador que **square** regresa al llamador un resultado entero. El compilador hace referencia al prototipo de la función para verificar que las llamadas a **square** contengan el tipo de regreso correcto, el número correcto de argumentos, el tipo correcto de argumentos, y que los argumentos están en el orden correcto. En la Sección 5.6 se analizan los prototipos de las funciones en detalle.

El formato de una definición de función es

tipo de valor de regreso-nombre de la función (*lista de parámetros*)
{
    declaraciones

    enunciados
}

El *nombre de la función* es cualquier identificador válido. El *tipo de valor de regreso* es el tipo de los datos del resultado regresado al llamador. El *tipo de valor de regreso* **void** indica que una función no devolverá un valor. Un *tipo de valor de regreso* no especificado será siempre supuesto por el compilador como **int**.

Error común de programación 5.2

Omitir el tipo de valor de regreso en una definición de función causa un error de sintaxis, si el prototipo de función especifica un regreso de tipo distinto a **int**.

Error común de programación 5.3

Olvidar regresar un valor de una función, que se supone debe regresar un valor, puede llevar a errores inesperados. El estándar ANSI indica que el resultado de esta omisión queda indefinido.

Error común de programación 5.4

Regresar un valor de una función, cuyo tipo de regreso se ha declarado como `void`*, causará un error de sintaxis.*

Práctica sana de programación 5.4

Aun cuando un tipo de regreso omitido resulte en `int` *por omisión, declare siempre en forma explícita el tipo de regreso. Sin embargo, por lo regular, se omite el tipo de regreso correspondiente a* `main`*.*

La *lista de parámetros* consiste en una lista, separada por comas, que contiene las declaraciones de los parámetros recibidas por la función al ser llamada. Si una función no recibe ningún valor, la *lista de parámetros* es **`void`**. Para cada parámetro deberá ser enlistado de forma explícita un tipo, a menos de que el parámetro sea del tipo `int`. Si un tipo no es enlistado, se supondrá como **`int`**.

Error común de programación 5.5

Declarar parámetros de función del mismo tipo como `float x, y` *en vez de* `float x, float y`*. La declaración de parámetros* `float x, y` *convertiría de hecho a* `y` *en un parámetro del tipo* `int`*, porque* `int` *es el valor por omisión.*

Error común de programación 5.6

Es un error de sintaxis colocar un punto y coma después del paréntesis derecho que encierra una lista de parámetros de una definición de función.

Error común de programación 5.7

Volver a definir dentro de la función un parámetro de función como variable local es un error de sintaxis.

Práctica sana de programación 5.5

Incluya en la lista de parámetros el tipo de cada parámetro, inclusive si algún parámetro es del tipo por omisión `int`*.*

Práctica sana de programación 5.6

Aunque hacerlo no es incorrecto, no utilice los mismos nombres para argumentos pasados a una función y parámetros correspondientes de la definición de función. Con ello ayuda a evitar ambigüedad.

Las *declaraciones*, junto con *los enunciados* dentro de las llaves, forman el *cuerpo de la función*. El cuerpo de la función también se conoce como un *bloque*. Un bloque es un enunciado compuesto que incluye declaraciones. Las variables pueden ser declaradas en cualquier bloque, y los bloques pueden estar anidados. *Bajo ninguna circunstancia puede ser definida una función en el interior de otra función.*

Error común de programación 5.8

Definir una función en el interior de otra función es un error de sintaxis.

Práctica sana de programación 5.7

Seleccionar nombres significativos para funciones y nombres significativos para parámetros, hace que sean más legibles los programas y ayuda a evitar un uso de comentarios excesivo.

Observación de ingeniería de software 5.5

Una función no debería tener una longitud mayor que una página. Aún mejor, una función no debería ser más larga que media página. Las funciones pequeñas promueven la reutilización del software.

Observación de ingeniería de software 5.6

Los programas deberían ser escritos como recopilaciones de pequeñas funciones. Esto haría que los programas fuesen más fáciles de escribir, de depurar, de mantener y de modificar.

Observación de ingeniería de software 5.7

Una función que requiera un gran número de parámetros quizás esté ejecutando demasiadas tareas. Considere dividir esta función en funciones más pequeñas, que ejecuten las tareas por separado. El encabezado de función, si es posible, debería caber en una línea.

Observación de ingeniería de software 5.8

El prototipo de función, el encabezado de función y las llamadas de función deberán todas estar de acuerdo en lo que se refiere al número, tipo y orden de argumentos y de parámetros, así como en el tipo de valor de regreso.

Existen tres formas de regresar el control al punto desde el cual se invocó a una función. Si la función no regresa un resultado, el control sólo se devuelve cuando se llega a la llave derecha que termina la función, o al ejecutar el enunciado

```
return;
```

Si la función regresa un resultado, el enunciado

`return`; *expresión*`;`

devuelve el valor de *expresión* al llamador.

En nuestro segundo ejemplo se utiliza una función **`maximum`** definida por el programador, para determinar y regresar el mayor de tres enteros (figura 5.4). Los tres enteros son introducidos mediante **`scanf`**. A continuación los enteros son pasados a **`maximum`**, que determina cual es el más grande. Este valor es regresado a **`main`** mediante el enunciado **`return`** existente en **`maximum`**. El valor regresado es asignado a la variable **`largest`** que entonces es impresa con el enunciado **`printf`**.

5.6 Prototipos de funciones

Una de las características más importantes de ANSI C es el *prototipo de función*. Esta característica fue tomada prestada por el comité de ANSI C de los que estaban desarrollando C++. Un prototipo de función le indica al compilador el tipo de dato regresado por la función, el número de parámetros que la función espera recibir, los tipos de dichos parámetros, y el orden en el cual se esperan dichos parámetros. El compilador utiliza los prototipos de funciones para verificar las llamadas de función. Versiones anteriores de C no ejecutaban este tipo de verificación, por lo que era posible llamar de forma incorrecta a las funciones, sin que el compilador se diera cuenta de los errores. Estas llamadas podían resultar en errores fatales en tiempo de ejecución, o en errores no fatales, que causaban sutiles errores lógicos y difíciles de detectar. Los prototipos de funciones de ANSI C corrigen esta deficiencia.

```
/* Finding the maximum of three integers */
#include <stdio.h>

int maximum(int, int, int);    /* function prototype */

main()
{
   int a, b, c;

   printf("Enter three integers: ");
   scanf("%d%d%d", &a, &b, &c);
   printf("Maximum is: %d\n", maximum(a, b, c));

   return 0;
}

/* Function maximum definition */
int maximum(int x, int y, int z)
{
   int max = x;

   if (y > max)
      max = y;

   if (z > max)
      max = z;

   return max;
}
```

```
Enter three integers: 22  85  17
Maximum is: 85
```

```
Enter three integers: 85  22  17
Maximum is: 85
```

```
Enter three integers: 22  17  85
Maximum is: 85
```

Fig. 5.4 Definición-programador de función de **maximum**.

Práctica sana de programación 5.8

Incluya prototipos de funciones para todas las funciones para aprovechar las capacidades de C de verificación de tipo. Utilice las directrices de preprocesador **#include** *para obtener los prototipos de las funciones estándar de biblioteca a partir de los archivos de cabecera de las bibliotecas apropiadas. También utilice* **#include** *para obtener archivos de cabecera que contengan prototipos de funciones utilizadas por usted y los miembros de su grupo.*

El prototipo de función para **maximum** en la figura 5.4 es

```
int maximum(int, int, int);
```

Este prototipo de función indica que **maximum** toma tres argumentos del tipo **int**, y regresa un resultado del tipo **int**. Note que el prototipo de función es la misma que la primera línea de la definición de función de **maximum**, a excepción de los nombres de los parámetros (**x**, **y** y **z**), mismos que no se incluyen.

Práctica sana de programación 5.9

Los nombres de los parámetros a veces se incluyen en los prototipos de función por razones de documentación. El compilador ignora estos nombres.

Error común de programación 5.9

Olvidar el punto y coma al final de un prototipo de función hará que ocurra un error de sintaxis.

Una llamada de función que no coincida con el prototipo de la función causará un error de sintaxis. Un error también se generará si el prototipo de la función y la definición de ésta no están de acuerdo. Por ejemplo, en la figura 5.4, si el prototipo de función hubiera sido escrita

```
void maximum (int, int, int);
```

el compilador habría generado un error, porque el tipo de regreso **void** en el prototipo de función, es diferente del tipo de regreso **int** del encabezado de función.

Otra característica importante de prototipos de función es la *coerción de argumentos*, es decir, obligar a los argumentos al tipo apropiado.

Por ejemplo, la función matemática de biblioteca **sqrt** puede ser llamada con un argumento entero, aun cuando el prototipo de función en **math.h** especifica un argumento **double** y aún así la función operará de forma correcta. El enunciado

```
printf("%.3f\n", sqrt(4));
```

valuará de forma correcta **sqrt(4)** e imprimirá el valor **2.000**. El prototipo de función hace que el compilador convierta el valor entero **4** al valor **double 4.0**, antes de que el valor sea pasado a **sqrt**. En general, los valores de los argumentos que no correspondan precisamente a los tipos de los parámetros de el prototipo de función serán convertidos al tipo apropiado, antes de que la función sea llamada. Estas conversiones pueden llevar a resultados incorrectos si no son seguidas las *reglas de promoción* de C. Las reglas de promoción definen como deben ser convertidos los tipos a otros tipos, sin perder datos. En nuestro ejemplo **sqrt** arriba citado, un **int** es convertido automáticamente a un **double**, sin cambiar su valor. Sin embargo, un **double** convertido a **int** trunca la parte fraccional del valor **double**. Convertir tipos enteros grandes a tipos enteros pequeños (es decir, **long** a **short**), también puede dar como resultado valores cambiados.

Las reglas de promoción se aplican automáticamente a expresiones que contengan valores de dos o más tipos de datos (también conocidas como expresiones de *tipo mixto*). El tipo de cada valor en una expresión de tipo mixto es automáticamente promovida al tipo más alto en la expresión (de hecho se crea una versión temporal de cada valor y se utiliza para la expresión conservándose sin cambio los valores originales). En la figura 5.5 se enlistan los tipos de datos en orden, desde el tipo más alto hasta el tipo más bajo, con cada uno de los tipos de especificaciones de conversión de **printf** y de **scanf**.

Tipos de datos	Especificaciones de conversión printf	Especificaciones de conversión scanf
long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
unsigned long int	%lu	%lu
long int	%ld	%ld
unsigned int	%u	%u
int	%d	%d
short	%hd	%hd
char	%c	%c

Fig. 5.5 Jerarquía de promoción para tipos de datos.

La conversión de valores a tipos inferiores por lo regular resulta en un valor incorrecto. Por lo tanto, un valor puede ser convertido sólo a un valor inferior, asignando de manera explícita el valor a una variable de tipo menor inferior, o mediante el uso de un operador cast. Los valores de los argumentos de las funciones son convertidos a los tipos de parámetro en un prototipo de función, como si hubieran sido asignados directamente a las variables de esos tipos. Si nuestra función **square**, que utiliza un parámetro entero (figura 5.3), es llamada con un argumento de punto flotante, el argumento se convertirá a **int** (un tipo inferior) y **square** normalmente regresará un valor incorrecto. Por ejemplo, **square(4.5)** regresaría **16** y no **20.25**.

Error común de programación 5.10

Convertir de un tipo de datos superior en la jerarquía de promoción a un tipo inferior, puede modificar el valor del dato.

Si el prototipo de función de una función no ha sido incluida en un programa, el compilador forma su propio prototipo de función utilizando la primera ocurrencia de la función ya sea la definición de función o una llamada a dicha función. Por omisión, el compilador asumirá que la función regresa un **int**, y no se supone nada en relación con los argumentos. Por lo tanto, si los argumentos pasados a la función no son correctos, los errores no serán detectados por el compilador.

Error común de programación 5.11

Olvidar un prototipo de función generará un error de sintaxis, si el tipo de regreso de la función no es int y la definición de función aparece después de la llamada a la función dentro del programa. De lo contrario, el olvidar un prototipo de función puede causar un error en tiempo de ejecución o un resultado inesperado.

Observación de ingeniería de software 5.9

Un prototipo de función, colocada fuera de una definición de función, se aplica a todas las llamadas de función que aparezcan dentro del archivo después del prototipo de la función. Una prototipo de función colocado en una función, se aplica sólo a las llamadas efectuadas en esa función.

5.7 Archivos de cabecera

Cada biblioteca estándar tiene su *archivo de cabecera* correspondiente, que contiene los prototipos de función de todas las funciones de dicha biblioteca, y las definiciones de varios tipos de datos y de constantes requeridas por dichas funciones. La figura 5.6 enlista de forma alfabética los archivos de cabecera estándar de biblioteca que pudieran incluirse en programas. El término "macros" que se utiliza varias veces en la figura 5.6, se analiza en detalle en el capítulo 13, "Preprocesador".

Archivo de cabecera de la biblioteca estándar	Explicación
<asser.h>	Contiene macros así como información para añadir diagnósticos que ayudan a la depuración de programas.
<ctype.h>	Contiene prototipos de funión para funciones que prueban caracteres en relación con ciertas propiedades, y prototipos de funciones que pueden ser utilizadas para la conversión de minúsculas a mayúsculas y viceversa.
<errno.h>	Define macros que son útiles para la información de condiciones de error.
<float.h>	Contiene los límites de tamaño de punto flotante del sistema.
<limits.h>	Incluye los límites de tamaño integral del sistema.
<locale.h>	Contiene los prototipos de función y otra información que le permite a un programa ser modificado en relación con la localización actual en la cual se está ejecutando. La noción de localización le permite al sistema de cómputo manejar diferentes reglas convencionales para la expresión de datos como son fechas, horas, monedas y números grandes en diferentes áreas del mundo.
<math.h>	Contiene prototipos para funciones matemáticas de biblioteca.
<setjmp.h>	Contiene prototipos para funciones que permiten pasar por alto la secuencia usual de llamadas de función y regreso.
<signal.h>	Contiene prototipos de función y macros para manejar varias condiciones que pudieran ocurrir durante la ejecución de un programa.
<stdarg.h>	Define macros para manejar con una lista de argumentos a una función cuyo número y cuyo tipo son desconocidos.
<stddef.h>	Contiene definiciones comunes de tipos utilizados en C para ejecutar ciertos cálculos.
<stdio.h>	Contiene prototipos para las funciones de biblioteca de entrada y salida estándar y la información que éstas utilizan.
<stdlib.h>	Contiene prototipos de función para las conversiones de números a texto y de texto a números, para la asignación de memoria, para números aleatorios y otras funciones de utilería.
<string.h>	Contiene prototipos para las funciones de procesamiento de cadenas.
<time.h>	Contiene prototipos de función y tipos para manipular hora y fecha.

Fig. 5.6 Archivo de cabecera de la biblioteca estándar.

El programador puede crear archivos de cabecera personalizados. Los archivos de cabecera personalizados definidos por el usuario, también deben terminar en **.h**. Un archivo de cabecera definido por el programador puede ser incluido utilizando la directriz de preprocesador **#include**. Por ejemplo, el archivo de cabecera **square.h** puede ser incluido en nuestro programa mediante la directriz

```
#include "square.h"
```

en la parte superior del programa. En la sección 13.2 se presenta información adicional sobre la inclusión de archivos de cabecera.

5.8 Cómo llamar funciones: llamada por valor y llamada por referencia

Dos formas de invocar funciones en muchos lenguajes de programación son la *llamada por valor* y la *llamada por referencia*. Cuando los argumentos se pasan en llamada por valor, se efectúa una *copia* del valor del argumento y ésta se pasa a la función llamada. Las modificaciones a la copia no afectan al valor original de la variable del llamador. Cuando un argumento es pasado en llamada por referencia, el llamador de hecho permite que la función llamada modifique el valor original de la variable.

La llamada por valor debería ser utilizada siempre que la función llamada no necesite modificar el valor de la variable original del llamador. Esto evita *efectos colaterales* accidentales, que obstaculizan de forma tan importante el desarrollo de sistemas correctos y confiables de software. La llamada por referencia debe ser utilizada sólo en funciones llamadas confiables, que necesitan modificar la variable original.

En C, todas las llamadas son llamadas por valor. Como veremos en el capítulo 7, es posible *simular* la llamada por referencia mediante la utilización de operadores de dirección y de indirección. En el capítulo 6, veremos que los arreglos son pasados en forma automática, en forma de llamadas por referencia simuladas. Tendremos que esperar hasta que lleguemos al capítulo 7 para una comprensión completa de este tema complejo. Por ahora, nos concentraremos en la llamada por valor.

5.9 Generación de números aleatorios

Ahora nos desviaremos en forma breve y se espera en forma entretenida una popular aplicación de programación, es decir, la simulación y la ejecución de juegos. En esta sección y en la siguiente, desarrollaremos un programa para ejecución de juegos bien estructurado, que incluye muchas funciones. El programa utiliza la mayor parte de las estructuras de control que hemos estudiado.

Existe algo en el aire de los casinos de juego, que excita a todo tipo de personas, desde profesionales de las elegantes mesas de caoba y fieltro para dados, a los bandidos de un solo brazo de las máquinas tragamonedas. Es el *elemento suerte*, la posibilidad que la suerte pueda convertir una simple bolsa de dinero en una montaña de riquezas. El *elemento suerte* puede ser introducido en las aplicaciones de cómputo, mediante el uso de la función **rand** existente en la biblioteca estándar de C

Considere el enunciado siguiente:

```
i = rand();
```

La función **rand** genera un entero entre 0 y **RAND_MAX** (una constante simbólica definida en el archivo de cabecera **<stdlib.h>** El estándar ANSI indica que el valor de **RAND_MAX** debe ser por lo menos 32767, que es el valor máximo de un entero de dos bytes (es decir, 16 bits). Los

programas en esta sección fueron probados en un sistema C, con un valor máximo de 32767 para **RAND_MAX**. Si **rand** en verdad produce enteros aleatorios, cualquier número entre 0 y **RAND_MAX** tiene la misma *oportunidad* (*o probabilidad*) de ser elegido, cada vez que **rand** es llamado.

El rango de valores producidos directamente por **rand**, a menudo son distintos de lo que se requiere para una aplicación específica. Por ejemplo, un programa que simule lanzar una moneda al aire, pudiera requerir sólo 0 para "caras" y 1 para "cruces". Un programa de juego de dados, que simule un dado de seis caras, requeriría de enteros aleatorios del rango 1 al 6.

A fin de demostrar **rand**, desarrollemos un programa para simular 20 tiradas de un dado de seis caras, e imprimamos el valor de cada tirada. El prototipo para la función **rand** puede ser encontrada en **<stdlib.h>** Utilizaremos el operador de módulo (%) en conjunción con **rand**, como sigue

```
rand() % 6
```

para producir enteros, en el rango 0 a 5. Esto se llama *dimensionar*. El número 6 se conoce como *factor de dimensionamiento*. Entonces después *desplazamos* el rango de números producidos añadiendo a nuestro resultado anterior la unidad. La figura 5.7 confirma que los resultados quedan dentro del rango de 1 a 6.

Para mostrar que estos números ocurren aproximadamente con la misma probabilidad, con el programa de la figura 5.8, simularemos 6000 tiradas de un dado. Cada entero del 1 al 6 debería de aparecer aproximadamente 1000 veces.

```
/* Shifted, scaled integers produced by 1 + rand() % 6 */
#include <stdio.h>
#include <stdlib.h>

main()
{
   int i;

   for (i = 1; i <= 20; i++) {
      printf("%10d", 1 + (rand() % 6));

      if (i % 5 == 0)
         printf("\n");
   }

   return 0;
}
```

```
 5     5     3     5     5
 2     4     2     5     5
 5     3     2     2     1
 5     1     4     6     4
```

Fig. 5.7 Desplazado y dimensionado de enteros producidos por **1 + rand ( ) % 6.**

```
/* Roll a six-sided die 6000 times */
#include <stdio.h>
#include <stdlib.h>

main()
{
   int face, roll, frequency1 = 0, frequency2 = 0,
       frequency3 = 0, frequency4 = 0,
       frequency5 = 0, frequency6 = 0;

   for (roll = 1; roll <= 6000; roll++) {
      face = 1 + rand() % 6;

      switch (face) {
         case 1:
            ++frequency1;
            break;
         case 2:
            ++frequency2;
            break;
         case 3:
            ++frequency3;
            break;
         case 4:
            ++frequency4;
            break;
         case 5:
            ++frequency5;
            break;
         case 6:
            ++frequency6;
            break;
      }
   }

   printf("%s%13s\n", "Face", "Frequency");
   printf("   1%13d\n", frequency1);
   printf("   2%13d\n", frequency2);
   printf("   3%13d\n", frequency3);
   printf("   4%13d\n", frequency4);
   printf("   5%13d\n", frequency5);
   printf("   6%13d\n", frequency6);
   return 0;
}
```

```
Face      Frequency
   1            987
   2            984
   3           1029
   4            974
   5           1004
   6           1022
```

Fig. 5.8 Tirar un dado de seis caras 6000 veces.

Como muestra el resultado del programa, mediante el dimensionamiento y el desplazamiento hemos utilizado la función **rand** para simular en forma realista tirar un dado de seis caras. Advierta que en la estructura **switch** *no* se ha previsto un caso **default**. También note la utilización del especificador de conversión **%s**, para imprimir las cadenas de caracteres **"Face"** y **"Frequency"** como encabezados de columna. Después de que en el capítulo 6 estudiemos los arreglos, mostraremos como remplazar de una manera elegante toda la estructura **switch** con un solo enunciado en una línea.

La ejecución del programa de la figura 5.7 por segunda vez produce

```
   5     5     3     5     5
   2     4     2     5     5
   5     3     2     2     1
   5     1     4     6     4
```

Note que apareció impresa exactamente la misma secuencia de valores. ¿Cómo pueden ser aleatorios estos valores? Irónicamente, esta capacidad de repetición es una característica importante de la función **rand**. Al depurar un programa, esta capacidad es esencial para comprobar que las correcciones efectuadas a un programa se ejecuten de manera correcta.

La función **rand** de hecho genera *números seudoaleatorios*. Al llamar repetidamente a **rand** produce una secuencia de números que parecen ser aleatorios. Sin embargo, cada vez que el programa se ejecute la secuencia se repetirá a sí misma. Una vez que el programa se haya depurado con cuidado, puede ser condicionado para producir en cada una de las ejecuciones una secuencia diferente de números aleatorios. A esto se le llama hacerlo *aleatorio* y se lleva a cabo mediante la función estándar de biblioteca **srand**. La función **srand** toma un argumento entero **unsigned** (la semilla), para que en cada ejecución del programa la función **rand** produzca una secuencia diferente de números aleatorios.

El uso de **srand** queda demostrado en la figura 5.9. En el programa, utilizamos el tipo de datos **unsigned**, que es una abreviación de **unsigned.int**. Se almacena un **int** en por lo menos dos bytes de memoria, y puede tener valores positivos y negativos. Una variable del tipo **unsigned** también queda almacenada en por lo menos dos bytes de memoria. Un **unsigned.int** de dos bytes puede tener sólo valores positivos dentro del rango 0 hasta 65535. Un **unsigned.int** de cuatro bytes puede tener sólo valores positivos en el rango desde 0 hasta 4294967295. La función **srand** toma un valor **unsigned** como argumento. Se utiliza el especificador de conversión %u para leer un valor **unsigned** utilizando **scanf**. El prototipo de función **srand** se encuentra en **<stdlib.h>**

Ejecutemos varias veces el programa y observemos los resultados. Note que cada vez que se ejecuta el programa se obtiene una secuencia *diferente* de números aleatorios, siempre y cuando se le dé una semilla diferente.

Si deseamos hacerlo aleatorio sin necesidad de introducir cada vez una semilla, pudiéramos utilizar un enunciado como

```
srand(time(NULL));
```

Esto hace que la computadora lea su reloj para obtener en forma automática un valor para la semilla. La función **time** devuelve la hora actual del día en segundos. Este valor es convertido a un entero unsigned y es utilizado como semilla para el generador de números aleatorios. La función **time** toma **NULL** como argumento (**time** es capaz de proporcionar al programador una cadena

```
/* Randomizing die-rolling program */
#include <stdlib.h>
#include <stdio.h>

main()
{
   int i;
   unsigned seed;

   printf("Enter seed: ");
   scanf("%u", &seed);
   srand(seed);

   for (i = 1; i <= 10; i++) {
      printf("%10d", 1 + (rand() % 6));

      if (i % 5 == 0)
         printf("\n");
   }

   return 0;
}
```

```
Enter seed: 67
         1         6         5         1         4
         5         6         3         1         2
```

```
Enter seed: 432
         4         2         5         4         3
         2         5         1         4         4
```

```
Enter seed: 67
         1         6         5         1         4
         5         6         3         1         2
```

Fig. 5.9 Programa de tirada del dado para hacerlo aleatorio.

representando la hora del día; **NULL** deshabilita esta capacidad para una llamada específica a **time**). El prototipo de función correspondiente a **time** está en **<time.h>**.

Los valores producidos directamente por **rand** estarán siempre en el rango:

```
0 ≤ rand() ≤ RAND_MAX
```

Previamente hemos demostrado cómo escribir un solo enunciado en C para simular la tirada de un dado de seis caras:


```
face = 1 + rand() % 6;
```

Este enunciado siempre asigna un valor entero (aleatorio) a la variable **face** en el rango $1 \leq$ **face** ≤ 6. Note que el ancho de este rango (es decir, el número de enteros consecutivos dentro del rango) es 6 y el número inicial dentro del rango es 1. Regresando al enunciado anterior, vemos que el ancho del rango se determina por el número utilizado para dimensionar a rand utilizando el operador de módulo (es decir 6), y el número inicial del rango es igual al número (es decir 1) que se añade a **rand % 6.** Podemos generalizar este resultado como sigue

```
n = a + rand() % b;
```

donde **a** es el *valor de desplazamiento* (que resulta igual al primer número del rango deseado de enteros consecutivos), y **b** es el factor de dimensionamiento (que es igual al ancho del rango deseado de enteros consecutivos). En los ejercicios, veremos que es posible escoger enteros al azar de conjuntos de valores distintos que los rangos de enteros consecutivos.

Error común de programación 5.12

Usar **srand** *en vez de* **rand** *para generar números aleatorios.*

5.10 Ejemplo: un juego de azar

Uno de los juegos de azar más populares en el juego de dados es el juego de dados conocido como "craps" (tiro perdedor), mismo que se juega en casinos y en callejuelas en todo el mundo. Las reglas del juego son sencillas:

Un jugador tira dos dados. Cada dado tiene seis caras. Las caras contienen 1, 2, 3, 4, 5, y 6 puntos. Una vez que los dados se hayan detenido, se calcula la suma de los puntos en las dos caras superiores. Si a la primera tirada, la suma es 7, o bien 11, el jugador gana. Si en la primera tirada la suma es 2, 3, o 12 (conocido como "craps"), el jugador pierde (es decir, la casa "gana"). Si en la primera tirada, la suma es 4, 5, 6, 8, 9 ó 10, entonces dicha suma se convierte en el "punto" o en la tirada. Para ganar, el jugador deberá continuar tirando los dados hasta que haga su "tirada". El jugador perderá si antes de hacer su tirada sale una tirada de 7.

El programa de la figura 5.10 simula el juego de craps. La figura 5.11 muestra varias ejecuciones de muestra.

Note que el jugador debe tirar dos dados en la primera tirada, y debe hacer lo mismo en todas las tiradas subsecuentes. Definimos una función **rollDice** para tirar los dados y calcular e imprimir su suma. La función **rollDice** se define una vez, pero es llamada desde dos lugares en el programa. De forma interesante, **rollDice** no toma argumentos, por lo que en la lista de parámetros hemos indicado **void**. La función **rollDice** sí regresa la suma de los dos dados, por lo que es indicado colocar un regreso de tipo **int** en el encabezado de función.

El juego es razonablemente complicado. El jugador puede perder o ganar en la primera tirada, o puede perder o ganar en cualquier tirada subsecuente. La variable **gameStatus** se utiliza para llevar registro de todo lo anterior.

Cuando se gana el juego, ya sea en la primera tirada o en una tirada subsecuente, **gameStatus** se define en 1. Cuando se pierde el juego, ya sea en la primera tirada o en una tirada subsecuente, **gameStatus** se define como 2. De lo contrario, **gameStatus** es cero y el juego deberá continuar.

```
/* Craps */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int rollDice(void);

main()
{
   int gameStatus, sum, myPoint;

   srand(time(NULL));
   sum = rollDice();              /* first roll of the dice */

   switch(sum) {
      case 7: case 11:            /* win on first roll */
         gameStatus = 1;
         break;
      case 2: case 3: case 12:    /* lose on first roll */
         gameStatus = 2;
         break;
      default:                    /* remember point */
         gameStatus = 0;
         myPoint = sum;
         printf("Point is %d\n", myPoint);
         break;
   }

   while (gameStatus == 0) {      /* keep rolling */
      sum = rollDice();

      if (sum == myPoint)         /* win by making point */
         gameStatus = 1;
      else
         if (sum == 7)            /* lose by rolling 7 */
            gameStatus = 2;
   }

   if (gameStatus == 1)
      printf("Player wins\n");
   else
      printf("Player loses\n");

   return 0;
}

int rollDice(void)
{
   int die1, die2, workSum;

   die1 = 1 + (rand() % 6);
   die2 = 1 + (rand() % 6);
   workSum = die1 + die2;
   printf("Player rolled %d + %d = %d\n", die1, die2, workSum);
   return workSum;
}
```

Fig. 5.10 Programa que simula el juego de craps.


```
Player rolled 6 + 5 = 11
Player wins
```

```
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 4 + 6 = 10
Point is 10
Player rolled 2 + 4 = 6
Player rolled 6 + 5 = 11
Player rolled 3 + 3 = 6
Player rolled 6 + 4 = 10
Player wins
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 1 + 4 = 5
Player rolled 5 + 4 = 9
Player rolled 4 + 6 = 10
Player rolled 6 + 3 = 9
Player rolled 1 + 2 = 3
Player rolled 5 + 2 = 7
Player loses
```

Fig. 5.11 Muestra de ejecuciones en el juego de craps.

Después de la primera tirada, si se ha terminado el juego, la estructura **while** es saltada, dado que **gameStatus** no es igual a cero. El programa continúa a la estructura **if/else**, que imprime "**Player wins**", en el caso que **gameStatus** sea **1** y "**Player loses**", si **gameStatus** es igual a **2**.

Después de la primera tirada, si el juego no se ha terminado, entonces **sum** quedará guardada en **myPoint**. La ejecución continúa con la estructura **while** porque **gameStatus** es **0**. Cada vez a través del ciclo **while**, se llama a **rollDice** para producir una nueva **sum**. Si **sum** coincide con **myPoint**, **gameStatus** se define como **1** para indicar que el jugador ganó, entonces la prueba de continuidad **while** falla, y la estructura **if/else** imprimirá "**Player wins**" y la ejecución se terminará. Si **sum** es igual a **7** **gameStatus** se define como **2** para indicar que el jugador perdió, la prueba de terminación **while** falla, el enunciado **if/else** imprime "**Player loses**" y la ejecución termina.

Note la interesante estructura de control de este programa. Hemos utilizado dos funciones **main** y **rollDice** y las estructuras **switch**, **while**, **if/else**, y la **if** anidada. En los ejercicios, investigarémos varias características interesantes del juego de craps.

5.11 Clases de almacenamiento

En los capítulos 2 al 4, utilizamos identificadores para los nombres de variables. Los atributos de variables son: nombre, tipo y valor. En este capítulo, también usamos identificadores como nombre para funciones definidas por usuario. De hecho, cada identificador en un programa tiene otros atributos incluyendo clase de almacenamiento, duración de almacenamiento, alcance y enlace.

C proporciona cuatro clases de almacenamiento, que se indican por los *especificadores de clase de almacenamiento:* **auto**, **register**, **extern** y **static**. La *clase de almacenamiento* de un identificador ayuda a determinar su duración de almacenamiento, su alcance y su enlace. La *duración de almacenamiento* de un identificador es el periodo durante el cual dicho identificador existe en memoria. Algunos identificadores tienen una existencia breve, otros son creados y destruidos en forma repetida, y otros existen durante toda la ejecución de un programa. El *alcance* de un identificador en un programa es dónde puede ser referenciado el mismo. Algunos identificadores pueden ser referenciados a todo lo largo de un programa, y otros a partir de sólo porciones de un programa. El *enlace* de un identificador determina, para un programa de varios archivos fuete (un tema que investigaremos en el capítulo 14), si un identificador es conocido solamente en el archivo fuente actua, o en cualquier archivo fuente, mediante declaraciones apropiadas. Esta sección analiza las cuatro clases de almacenamiento y la duración del almacenamiento. En la sección 5.12 se analiza el alcance de los identificadores. En el capítulo 14, Temas avanzados, se estudia el enlace de identificadores y la programación con varios archivos fuente.

Los cuatro especificadores de clase de almacenamiento (persistencia) pueden ser divididos en dos: *persistencia automática* y *persistencia estática.* Las palabras reservadas **auto** y **register** se utilizan para declarar variables de persistencia automática. Estas variables se crean al introducirse al ámbito del bloque en el cual están declaradas, existen mientras dicho bloque está activo, y se destruyen cuando se sale de ese bloque.

Sólo las variables pueden tener persistencia automática. Las variables locales de una función (aquellas declaradas en la lista de parámetros o en el cuerpo de la función) por lo regular tienen una persistencia automática. La palabra reservada **auto** declara en forma explícita a las variables de persitencia automática. Por ejemplo, la siguiente declaración indica que las variables **float**, **x** e **y** son variables locales automáticas, y existen sólo en el cuerpo de la función en el cual aparece dicha declaración:

```
auto float x, y;
```

Por omisión, las variables locales tienen persistencia automática, por lo que la palabra reservada **auto** es rara vez utilizada. Durante el resto de este texto, nos ocuparemos de las variables con persistencia automática simplemente como variables automáticas.

Sugerencia de rendimiento 5.1

La persistencia automática es una forma de conservar memoria, porque las variables automáticas existen sólo cuando son necesitadas. Son creadas al introducirse la función en la cual son declaradas, y son destruidas cuando se sale de dicha función.

Observación de ingeniería de software 5.10

El almacenamiento automático es otra vez otro ejemplo del principio del menor privilegio. ¿Por qué tendrían que estar las variables almacenadas en memoria y accesibles cuando de hecho no son necesarias?

Los datos de un programa en la versión en lenguaje máquina, para cálculos y otros procesos normalmente se cargan en registros.

Sugerencia de rendimiento 5.2

El especificador de clase de almacenamiento **register** *puede ser colocado antes de una declaración de variable automática, para sugerir que el compilador conserve la variable en uno de los registros de alta velocidad del hardware de la computadora. Si se puede mantener en registros de hardware las variables muy utilizadas como son contadores y totales, puede ser eliminada la sobrecarga correspondiente a cargar en forma repetida las variables de las memorias a los registros y almacenar los resultados de vuelta en memoria.*

El compilador pudiera ignorar declaraciones **register**. Por ejemplo, quizás no exista un suficiente número de registros disponibles para que los utilice la computadora. La siguiente declaración sugiere que se coloque la variable entera **counter** en uno de los registros de la computadora y se inicialice a 1:

```
register int counter = 1;
```

La palabra reservada **register** puede ser utilizada sólo con variables automáticas.

Sugerencia de rendimiento 5.3

A menudo, son innecesarias las declaraciones **register**. *Los compiladores optimizadores de hoy son capaces de reconocer variables de uso frecuente, y pueden decidir colocarlos en registro, sin necesidad de una declaración* **register** *proveniente del programador.*

Las palabras reservadas **extern** y **static** se utilizan para declarar identificadores de variables y de funciones de persistencia estática. Los identificadores de persistencia estática existen a partir del momento de que el programa se inicia en ejecución. Se asigna y se inicializa almacenamiento para las variables desde el momento que se empieza a operar el programa. Para las funciones, existe el nombre de la función al iniciarse la ejecución del programa. Sin embargo, aunque las variables y los nombres de función existen a partir del inicio de la ejecución del programa, esto no significa que estos identificadores pudieran ser utilizados a todo lo largo del mismo. La persistencia y el alcance (dónde puede ser utilizado un nombre), son temas separados, como veremos en la sección 5.12.

Existen dos tipos de identificadores con persistencia estática: los identificadores externos (como son las variables globales y los nombres de función) y las variables locales declaradas con el especificador de clase de almacenamiento **static**. Las variables globales y los nombres de función son por omisión de la clase de almacenamiento **extern**. Las variables globales se crean colocando declaraciones variables por fuera de cualquier definición de función, y conservan sus valores a todo lo largo de la ejecución del programa. Las variables globales y las funciones pueden ser referenciadas por cualquier función posteriores a sus declaraciones o definiciones dentro del archivo. Esta es una razón para la utilización de prototipos de función. Cuando incluimos **stdio.h** en un programa que hace llamadas a **printf**, el prototipo de función se coloca al principio de nuestro archivo para hacer que el nombre **printf** sea conocido para el resto del archivo.

Observación de ingeniería de software 5.11

La declaración de una variable como global en vez de como local, permite que ocurran efectos colaterales no deseados cuando una función que no requiere de acceso a la variable la modifica accidental o maliciosamente. En general, el uso de las variables globales debería ser evitado, excepto en ciertas situaciones, con requerimientos de rendimiento únicos (como se analizan en el capítulo 14).

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE

Práctica sana de programación 5.10

Las variables utilizadas sólo en una función particular deberían ser declaradas como variables locales en esa función, en vez de como variables externas.

Las variables locales declaradas con la palabra reservada **static** son aún conocidas sólo en la función para la cual son definidas, pero a diferencia de las variables automáticas, las variables locales **static** conservan su valor cuando sale de la función. La próxima vez que se llama a esa función, la variable local **static** contiene el valor que tenía cuando la función salió por última vez. El siguiente enunciado declara la variable local **count** como **static** y la inicializa a 1.

```
static int count = 1;
```

Todas las variables numéricas **static** se incializan a cero si no son inicializadas de forma explícita por el programador. (Las variables de apuntador, analizadas en el capítulo 7, se inicializan a **NULL**).

Error común de programación 5.13

Usar múltiples especificadores de clase de almacenamiento para un identificador. Sólo puede ser aplicado un especificador de clase de almacenamiento a un identificador.

Las palabras reservadass **extern** y **static** tienen un significado especial, si se aplican explícitamente a identificadores externos. En el capítulo 14, Temas avanzados, analizaremos el uso explícito de **extern** y de **static**, junto con identificadores externos y con programas de archivos de fuentes múltiples.

5.12 Reglas de alcance

El *alcance* de un identificador es la porción del programa en el cual dicho identificador puede ser referenciado. Por ejemplo, cuando en un bloque declaramos una variable local, puede ser referenciada sólo en dicho bloque o en los bloques anidados dentro de dicho bloque. Los cuatro alcances posibles para un identificador son: *alcance de función , alcance de archivo, alcance de bloque,* y *alcance del prototipo de función.*

Las etiquetas (un identificador seguido por dos puntos como **start:**) son los únicos identificadores con *alcance de función*. Las etiquetas pueden ser utilizadas en cualquier parte dentro de la función en la cual aparecen, pero no pueden ser referenciadas fuera del cuerpo de la función. Las etiquetas se utilizan en estructuras **switch** (como etiquetas **case**) y en enunciados **goto** (vea el capítulo 14, Temas avanzados). Las etiquetas son detalles de puesta en marcha que las funciones ocultan una de la otra. Este ocultamiento conocido formalmente como *ocultamiento de información* es uno de los principios fundamentales de la buena ingeniería de software.

Un identificador declarado por fuera de cualquier función tiene *alcance de archivo*. Tal indicador es "conocido" en todas las funciones desde el punto donde el identificador se declara hasta el final del archivo. Las variables globales, las definiciones de función y los prototipos de función colocados fuera de una función, todas ellas tienen alcance de archivo.

Los identificadores dentro de un bloque, tienen *alcance de bloque*. El alcance de bloque termina en la llave derecha de terminación (}) del bloque. Las variables locales declaradas al principio de una función tienen alcance de bloque, como lo tienen los parámetros de función, que son consideradas por la función como variables locales. Cualquier bloque puede contener declaracio nes variables. Cuando los bloques están anidados, y un identificador de un bloque externo tiene el mismo nombre que un identificador de un bloque interno, el identificador del bloque externo estará "oculto" hasta que el bloque interno termine. Esto significa que, en tanto se

ejecute el bloque interno, el bloque interno ve el valor de su propio identificador local y no el valor del identificador de nombre idéntico, del bloque que lo contiene. Las variables locales declaradas **static**, aun tendrán alcance de bloque aunque existan a partir del momento en que empieza a ejecutarse el programa. Por lo tanto, la persitencia no afecta el alcance de un identificador.

Los únicos identificadores con *alcance de prototipo de función* son aquellos que se utilizan en la lista de parámetros del prototipo de una función. Tal y como se mencionó, los prototipos de función no requieren de nombres en la lista de parámetros —solo requieren de tipos. Si en la lista de parámetros de un prototipo de función se utiliza un nombre, el compilador ignorará dicho nombre. Los identificadores utilizados en un prototipo de función, pueden ser reutilizados en cualquier parte del programa, sin ambigüedad.

Error común de programación 5.14

Utilizar accidentalmente el mismo nombre para un identificador en un bloque interno, que se haya usado para un identificador en un bloque externo, cuando de hecho, el programador desea que durante la duración del bloque interno el identificador del bloque externo esté activo.

Práctica sana de programación 5.11

Evite nombres variables que oculten nombres en alcances externos. Esto se puede conseguir dentro de un programa evitando el uso de identificadores duplicados.

El programa de la figura 5.12 demuestra temas de alcance de variables globales, de variables locales automáticas, y de variables locales **static**. Una variable global **x** es declarada e inicializada a 1. Esta variable global se oculta en cualquier bloque (o función) en el cual una variable de nombre **x** está declarada. En **main**, una variable local **x** es declarada e inicializada a 5. Esta variable es impresa a continuación para mostrar que la **x** global está oculta en **main**. A continuación, se define un nuevo bloque en **main**, utilizando otra variable local **x** inicializada a 7. Esta variable se imprime, para mostrar que oculta a **x** en el bloque externo de **main**. La variable **x** con valor 7 de forma automática queda destruida al salir del bloque, y la variable local **x** del bloque externo de **main** se vuelve a imprimir otra vez, para mostrar que ya no está oculta. El programa define tres funciones, dónde cada una de ellas no toma argumento y no regresan nada. La función **a** define una variable automática **x**, y la inicializa a 25. Cuando se llama a **a**, la variable es impresa, incrementada y vuelta a imprimir, antes de salir de la función. Cada vez que esta función es llamada, la variable automática **x** es reinicializada a 25. La función **b** declara una variable **static** de nombre **x**, y la inicializa a 50. Las variables locales declaradas como **static** conservan sus valores, aun si están fuera de alcance. Cuando se llama a **b**, **x** es impreso, incrementado y vuelto a imprimir, antes de salir de la función. En la siguiente llamada a esta función, la variable local **static**, **x** contendrá el valor 51. La función C no declara ninguna variable. Por lo tanto, cuando se refiere a la variable **x**, se utiliza la global **x**. Cuando **c** es llamada, la variable global es impresa, multiplicada por 10, y vuelta a imprimir, antes de salir de la función. La siguiente llamada a la función **c**, la variable global aún tiene su valor modificado, 10. Por último, el programa imprime otra vez la variable local **x** de **main**, para mostrar que ninguna de las llamadas de función modificó el valor de **x**, porque las funciones todas ellas se referían a variables en otros alcances.

5.13 Recursión

Los programas que hemos analizado están estructurados en general como funciones que llaman unas a otras, en forma disciplinada y jerárquica. Para algunos tipos de problemas, es útil tener

```
/* A scoping example */
#include <stdio.h>

void a(void);    /* function prototype */
void b(void);    /* function prototype */
void c(void);    /* function prototype */

int x = 1;       /* global variable */

main()
{
   int x = 5;    /* local variable to main */

   printf("local x in outer scope of main is %d\n", x);

   {             /* start new scope */
      int x = 7;

      printf("local x in inner scope of main is %d\n", x);
   }             /* end new scope */

   printf("local x in outer scope of main is %d\n", x);

   a();          /* a has automatic local x */
   b();          /* b has static local x */
   c();          /* c uses global x */
   a();          /* a reinitializes automatic local x */
   b();          /* static local x retains its previous value */
   c();          /* global x also retains its value */

   printf("local x in main is %d\n", x);
   return 0;
}

void a(void)
{
   int x = 25;  /* initialized each time a is called */

   printf("\nlocal x in a is %d after entering a\n", x);
   ++x;
   printf("local x in a is %d before exiting a\n", x);
}

void b(void)
{
    static int x = 50;  /* static initialization only */
                        /* first time b is called */
    printf("\nlocal static x is %d on entering b\n", x);
    ++x;
    printf("local static x is %d on exiting b\n", x);
}

void c(void)
{
   printf("\nglobal x is %d on entering c\n", x);
   x *= 10;
   printf("global x is %d on exiting c\n", x);
}
```

Fig. 5.12 Ejemplo de alcance (parte 1 de 2).


```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 50 on entering b
local static x is 51 on exiting b

global x is 1 on entering c
global x is 10 on exiting c

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 51 on entering b
local static x is 52 on exiting b

global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5
```

Fig. 5.12 Ejemplo de alcance (parte 2 de 2).

funciones que se llamen a sí mismas. Una *función recursiva* es una función que se llama a sí misma, ya sea directa, o indirecta a través de otra función. La recursión es un tema complejo analizado con profundidad en los cursos de ciencia de cómputo de alto nivel. En esta sección y en la siguiente, se presentan ejemplos simples de recursión. Este libro contiene un extenso tratamiento de la recursión, distribuído a través de los capítulos 5 hasta el 12. En la figura 5.17, localizada al final de la sección 5.15, se resumen los 31 ejemplos de recursión y ejercicios del libro.

Primero consideraremos la recursión en forma conceptual, y a continuación examinarémos varios programas que contienen funciones recursivas. Los enfoques recursivos a la solución de problemas tienen ciertos elementos en común. Una función recursiva es llamada para resolver un problema. La función, de hecho, sabe sólo cómo resolver el caso más simple, es decir el llamado *base case* (caso base o *casos base*). Si la función es llamada con caso base, la función simplemente regresa un resultado. Si la función es llamada con un problema más complejo, la función divide dicho problema en dos partes conceptuales: una parte que la función ya sabe como ejecutar, y una parte que la función no sabe como ejecutar. Para hacer factible la recursión, esta última parte debe parecerse al problema original, pero resulta ligeramente más simple o una versión ligeramente más pequeña del problema original. Dado que este nuevo problema aparenta o se ve similar al problema original, la función emite (llama) a una copia nueva de sí misma, para que empiece a trabajar sobre el problema más pequeño y esto se conoce como una *llamada recursiva* y también se llama el *paso de recursión*. El paso de recursión también incluye la palabra reservada **return**, porque su resultado será combinado con la parte del problema que la función supo como resolver, para formar un resultado que será regresado al llamador original, posiblemente **main**.

El paso de recursión se ejecuta en tanto la llamada original a la función esté abierta, es decir, que no se haya terminado su ejecución. El paso de recursión puede dar como resultado muchas

más llamadas recursivas como éstas, conforme la función continúe dividiendo cada problema sobre el cual es llamada en dos partes conceptuales. A fin de que la recursión de forma eventual se termine, cada vez que la función se llame a sí misma sobre una versión ligeramente más sencilla que el problema original, esta secuencia de problemas cada vez más pequeños, eventualmente deben de convergir en el caso base. Llegado a este punto, la función reconocerá el caso base, regresa un resultado a la copia anterior de la función, y sigue a continuación una secuencia de regresos a todo lo largo de la línea, hasta llegar a la llamada original de la función, devolviendo el resultado final a **main**. Todo esto se escucha muy exótico, en comparación con el tipo de resolución convencional de problemas que hemos estado utilizando hasta este punto. De hecho, se requiere de gran cantidad de experiencia en la escritura de programas recursivos, antes de que el proceso se convierta en natural. Como un ejemplo de estos conceptos en operación, escribamos un programa recursivo para ejecutar un cálculo matemático popular.

El factorial de un entero no negativo *n*. escrito *n!* (y pronunciado " factorial de *n* "), es el producto

$$n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$$

con 1! igual a 1, y 0! definido como 1. Por ejemplo, 5! es el producto 5 * 4 * 3 * 2 * 1, lo cual resulta igual a 120.

El factorial de un entero, **number**, mayor que o igual a 0, puede ser calculado en forma *iterativa* (y no recursiva) utilizando a **for** como sigue:

```
factorial = 1;
for (counter = number; counter >= 1; counter--)
   factorial *= counter;
```

Una definición recursiva de la función factorial se consigue al observar la siguiente relación:

$$n! = n \cdot (n - 1)!$$

Por ejemplo, 5! claramente es lo mismo que 5 * 4!, como se muestra mediante el siguiente:

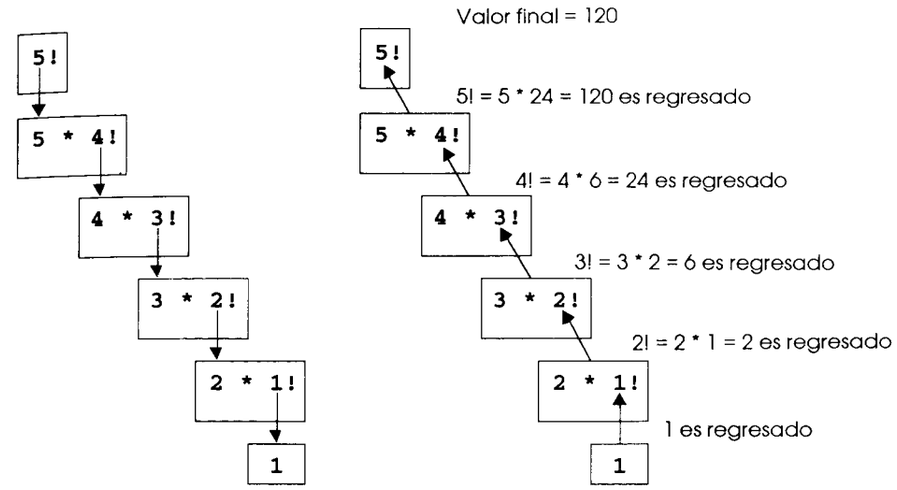
$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

La evaluación de 5! se haría como se muestra en la figura 5.13. La figura 5.13a muestra, como la sucesión de llamadas recursivas continúa hasta que 1! se evalúa al valor 1, lo que termina la recursión. En la figura 5.13b se muestran los valores regresados por cada llamada recursiva a su llamador, hasta que el valor final es calculado y regresado.

El programa de la figura 5.14 utiliza la recursión para calcular e imprimir los factoriales de los enteros 0 a 10 (la selección del tipo de datos **long** se explicará a continuación). La función recursiva **factorial** primero prueba para ver si una condición de terminación es verdadera, es decir, es **number** menor que o igual a 1. Si **number** es en verdad menor que o igual a 1, **factorial** regresa 1, ya no es necesaria mayor recursión, y el programa termina. Si **number** es mayor que 1, el enunciado

```
return number * factorial (number - 1);
```

expresa el problema como el producto de **number** y una llamada recursiva a **factorial** evaluando el **factorial** de **number - 1**. Note que el **factorial** (**number - 1**) es un problema ligeramente más simple que el cálculo original **factorial** (**number**).



a) Secuencia de llamadas recursivas b) Valores regresados de cada llamada recursiva.

Fig. 5.13 Evaluación recursiva de 5!

La función **factorial** ha sido declarada para recibir un parámetro del tipo **long** y regresar un resultado del tipo **long**. Esto es una notación abreviada correspondiente a **long int**. El estándar ANSI define que una variable del tipo **long int** se almacena en por lo menos 4 bytes, y, por lo tanto, puede contener un valor tan grande como +2147483647. Como se puede ver en la figura 5.14, los valores factoriales con rápidez se hacen grandes. Hemos escogido el tipo de datos **long**, a fin de que el programa pueda calcular factoriales mayores que 7! en computadoras con enteros pequeños (como de 2 bytes). El especificador de conversión **%ld** se utiliza para imprimir los valores long. Desafortunadamente la función factorial produce tan rápido valores grandes que inclusive **long int** no nos ayuda a imprimir muchos valores factoriales, antes de que el tamaño de la variable **long int** quede excedida.

Conforme exploremos los ejercicios, **float** y **double** pudieran al final ser necesarios para el usuario que desee calcular factoriales de número grandes. Esto apunta a una debilidad en C (y en la mayor parte de otros lenguajes de programación), es decir, que el lenguaje no se extiende con fácilidad para manejar los requisitos únicos de varias aplicaciones. Como veremos más adelante, C++ es un lenguaje extensible, que nos permite, si así lo deseamos, crear enteros grandes en forma arbitraria.

Error común de programación 5.15

Olvidar el regresar un valor de una función recursiva cuando se requiere de uno.

Error común de programación 5.16

Omitir ya sea el caso base, o escribir el paso de recursión en forma incorrecta, de tal forma que no converja al caso base, causando recursión infinita, y agotando de forma eventual la memoria. Esto es análogo al problema de un ciclo infinito en una solución iterativa (no recursiva). La recursión infinita también puede ser causada al proporcionarle una entrada no esperada.

```
/* Recursive factorial function */
#include <stdio.h>

long factorial(long);

main()
{
   int i;

   for (i = 1; i <= 10; i++)
      printf("%2d! = %ld\n", i, factorial(i));

   return 0;
}

/* Recursive definition of function factorial */
long factorial(long number)
{
   if (number <= 1)
      return 1;
   else
      return (number * factorial(number - 1));
}
```

```
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Fig. 5.14 Cálculos fáctoriales con una función recursiva.

5.14 Ejemplo utilizando recursión: la serie Fibonacci

La serie Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

empieza con 0 y 1, y tiene la propiedad que cada número Fibonacci subsecuente es la suma de los dos números Fibonacci previos.

La serie ocurre en la naturaleza y en particular describe una forma de espiral. La relación de números sucesivos Fibonacci converge a un valor constante de 1.618.... Este número, también, ocurre en forma repetida en la naturaleza y ha sido llamada la *regla áurea* o la *media áurea.*

Los seres humanos tienen tendencia a encontrar la media áurea estéticamente agradable. Los arquitectos a menudo diseñan las ventanas, habitaciones y edificios cuya longitud de ancho están en la relación de la media áurea. Las tarjetas postales a menudo se diseñan en una relación de longitud/ancho de la media áurea.

La serie Fibonacci puede ser definida de forma recursiva como sigue:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

El programa de la figura 5.15 calcula el número Fibonacci i^a en forma recursiva, utilizando la función **fibonacci**. Advierta que los números Fibonacci tienden con rápidez hacerse grandes. Por lo tanto, en la función **fibonacci** hemos escogido el tipo de datos **long** para el tipo de parámetros y para el de regreso. En la figura 5.15, cada par de líneas de salida muestra una ejecución separada del programa.

La llamada de **fibonacci** a partir de **main** no es una llamada recursiva, pero todas las llamadas subsecuentes a **fibonacci**, sí son recursivas. Cada vez que se invoca a **fibonacci**, de inmediato prueba por el caso base **n** es igual a 0 ó a 1. Si esto es verdadero, se regresa **n**. De forma interesante, si **n** es mayor que 1, el paso de recursión genera *dos* llamadas recursivas, cada una de las cuales es para un problema ligeramente más sencillo que la llamada original a **fibonacci**. La figura 5.16 muestra cómo la función **fibonacci** evaluaría **fibonacci(3)**, sólo abreviamos **fibonacci** como una **f**, a fin de hacer la figura más legible.

```
/* Recursive fibonacci function */
#include <stdio.h>

long fibonacci(long);

main()
{
   long result, number;

   printf("Enter an integer: ");
   scanf("%ld", &number);
   result = fibonacci(number);
   printf("Fibonacci(%ld) = %ld\n", number, result);
   return 0;
}

/* Recursive definition of function fibonacci */
long fibonacci(long n)
{
   if (n == 0 || n == 1)
      return n;
   else
      return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Fig. 5.15 Los números Fibonacci generan en forma recursiva (parte 1 de 2).

```
Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1

Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 3
Fibonacci(3) = 2

Enter an integer: 4
Fibonacci(4) = 3

Enter an integer: 5
Fibonacci(5) = 5

Enter an integer: 6
Fibonacci(6) = 8

Enter an integer: 10
Fibonacci(10) = 55

Enter an integer: 20
Fibonacci(20) = 6765

Enter an integer: 30
Fibonacci(30) = 832040

Enter an integer: 35
Fibonacci(35) = 9227465
```

Fig. 5.15 Los números Fibonacci generan en forma recursiva (Parte 2 de 2)

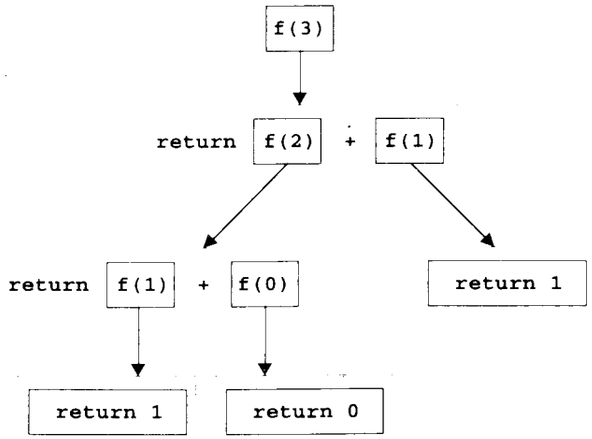


Fig. 5.16 Conjunto de llamadas recursivas a la función fibonacci.

Esta figura plantea algunos temas interesantes en relación con el orden en el cual los compiladores de C evaluarán los operandos de operadores. Se trata de un tema distinto al orden en el cual los operadores son aplicados a sus operandos, es decir al orden dictado por las reglas de precedencia de operadores. A partir de la figura 5.16 aparece que durante la evaluación de `f(3)`, se harán dos llamadas recursivas, es decir `f(2)` y `f(1)`. ¿Pero en qué orden se harán estas llamadas? La mayor parte de los programadores supone que los operandos serán evaluados de izquierda a derecha. De forma extraña, el estándar ANSI no especifica el orden en que deban ser evaluados los operandos de la mayor parte de los operadores (incluyendo `+`). Por lo tanto, el programador no debería hacer ninguna suposición en relación con el orden en el cual se ejecutarán estas llamadas. Las llamadas de hecho podrían ejecutar `f(2)` primero y a continuación `f(1)`, o las llamadas podrían ejecutarse en el orden inverso, `f(1)` y a continuación `f(2)`. En este programa y en la mayor parte de otros programas, de hecho el resultado final sería el mismo. Pero en algunos programas la evaluación de un operando pudiera tener efectos colaterales que pudieran afectar el resultado final de la expresión. De los muchos operadores de C, el estándar ANSI especifica el orden de evaluación de los operandos de sólo cuatro operadores —es decir, `&&`, `||`, el operador coma (`,`) e `?:`. Los primeros tres son operadores binarios, cuyos dos operandos están garantizados en su evaluación de izquierda a derecha. El último operador es el único operador ternario de C. Su operando más a la izquierda es el que primero se evalúa siempre; si el operando más a la izquierda tiene el valor de no cero, se evalúa a continuación el operando de enmedio y el último se ignora; si el operando más a la izquierda se evalúa a cero, el tercer operando se evalúa a continuación y el operando central es ignorado.

Error común de programación 5.17

Escribir programas que dependan del orden de evaluación de los operandos de operadores distintos que `&&`, `||`, `?:`, y el operador coma (`,`) puede llevar a errores, porque los compiladores no necesariamente evalúan los operandos en el orden en que los programas esperan.

Sugerencia de portabilidad 5.2

Los programas que dependen del orden de evaluación de los operandos o de operadores, diferentes a `&&`, `||`, `?:`, y el operador coma (`,`) pueden funcionar de forma distinta sobre sistemas con compiladores diferentes.

Una palabra de precaución merece ser dicha en relación con programas recursivos, como el que estamos utilizando aquí para generar números Fibonacci. Cada nivel de recursión en la función **fibonacci** tiene un efecto duplicador sobre el número de llamadas, es decir, el número de llamadas recursivas que se ejecutarán para calcular el número Fibonacci del orden n será del orden de 2^n. Esto se sale rápido de control. Simplemente calcular el 20º número Fibonacci requeriría del orden de 2^{20} o aproximadamente de un millón de llamadas, y calcular el número Fibonacci del orden treintavo requeriría del orden de 2^{30} o aproximadamente de mil millones de llamadas, y así en lo sucesivo. Los científicos de la computación se refieren a lo anterior como *complejidad exponencial*. ¡Problemas de esta naturaleza inclusive humillan a las computadoras más poderosas del mundo!. Los temas de complejidad en general, y la complejidad exponencial en particular, se analizan en detalle en un curso del curriculum de ciencia de las computadoras de alto nivel conocido como "Algoritmos".

Sugerencia de rendimiento 5.4

Evite programas recursivos de tipo fibonacci que resultan en una "explosión" exponencial de llamadas.

5.15 Recursión en comparación con iteración

En las secciones anteriores, hemos estudiado dos funciones que pueden con fácilidad instaurarse, ya sea en forma recursiva o en forma iterativa. En esta sección compararemos los dos enfoques y analizaremos por qué el programador debe escoger un enfoque sobre el otro, en una situación en particular.

Tanto la iteración como la recursión se basan en una estructura de control: la iteración utiliza una estructura de repetición; la recursión utiliza una estructura de selección. Tanto la iteración como la recursión implican repetición: la iteración utiliza la estructura de repetición en forma explícita; la recursión consigue la repetición mediante llamadas de función repetidas. La iteración y la recursión ambas involucran una prueba de terminación: la iteración termina cuando falla la condición de continuación de ciclo; la recursión termina cuando se reconoce un caso base. La iteración con una repetición controlada por contador y la recursión ambas se acercan de forma gradual a la terminación: la iteración continúa modificando un contador hasta que éste asume un valor que hace que la condición de continuación del ciclo falle; la recursión continúa produciendo versiones más sencillas del problema original hasta que se alcanza el caso base. Tanto la iteración como la recursión pueden ocurrir en forma indefinida: ocurre un ciclo infinito en la iteración si la prueba de continuación de ciclo nunca se hace falsa; la recursión infinita ocurre si el paso de recursión no reduce el problema en cada ocasión, de tal forma que converja al caso base.

La recursión tiene muchas negativas. Invoca en forma repetida al mecanismo y, por lo tanto, a la sobrecarga de las llamadas de función. Esto puede resultar costoso tanto en tiempo de procesador como en espacio en memoria. Cada llamada recursiva genera otra copia de la función (de hecho sólo de las variables de función); esto puede consumir gran cantidad de memoria. La iteración por lo regular ocurre dentro de una función, por lo que no ocurre la sobrecarga de llamadas repetidas de función y asignación extra de memoria. Por lo tanto, ¿por qué elegir la recursión?

Observación de ingeniería de software 5.12

Cualquier problema que puede ser resuelto en forma recursiva, también puede ser resuelto en forma iterativa (no recursiva). Normalmente se escoge un enfoque recursivo en preferencia a uno iterativo cuando el enfoque recursivo es más natural al problema y resulta en un programa que sea más fácil de comprender y de depurar. Otra razón para seleccionar una solución recursiva, es que la solución iterativa pudiera no resultar aparente.

Sugerencia de rendimiento 5.5

Evite el uso de la recursión cuando se requiera de rendimiento. Las llamadas recursivas toman tiempo y consumen memoria adicional.

Error común de programación 5.18

Hacer que accidentalmente una función no recursiva se llame a sí misma, ya sea directa o indirecta a través de otra función.

Muchos libros de texto de programación presentan la recursión mucho más adelante de lo que nosotros hemos hecho aquí. Pensamos que la recursión es un tema lo suficiente rico y complejo que es mejor presentarlo pronto y repartir los ejemplos sobre el resto del libro. La figura 5.17 resume por capítulos los 31 ejemplos de recursión y los ejercicios del libro.

Cerremos este capítulo con algunas observaciones que hacemos en forma repetida a todo lo largo del libro. Es importante la buena ingeniería de software. Es importante un alto rendimiento. Desafortunadamente, a menudo estas metas se contraponen. Es clave una buena ingeniería de

software para hacer manejable la tarea de desarrollar sistemas mayores y más complejos de software que estamos necesitando. Un alto rendimiento es clave también para llevar a cabo los sistemas del futuro que colocarán crecientes demandas sobre el hardware. ¿Dónde entran en este panorama las funciones?

Observación de ingeniería de software 5.13

La funcionalización de los programas de una forma nítida y jerárquica promueve buena ingeniería de software. Pero tiene un costo.

Sugerencia de rendimiento 5.6

Un programa muy funcionalizado en comparación con uno monolítico (es decir, de una pieza) sin funciones potencialmente hace grandes cantidades de llamadas de función y estas consumen tiempo de ejecución en el procesador de una computadora. Pero los programas monolíticos son difíciles de programar, de probar, de depurar, de mantener y de modificar.

Por lo tanto, funcionalice sus programas con juicios, manteniendo en mente siempre un delcado equilibrio entre rendimiento y buena ingeniería de software.

Resumen

- La mejor forma de desarrollar y mantener un programa grande es dividirlo en varios módulos de programa más pequeños, siendo cada uno de ellos más manejables que el programa original. En C los módulos se escriben como funciones.
- Una función se invoca mediante una llamada de función. La llamada de función menciona a la función por su nombre y proporciona información (en forma de argumentos) que la función llamada requiere para ejecutar su tarea.
- El objeto del ocultamiento de información es para que las funciones tengan acceso sólo a la información que requieren para completar sus tareas. Esto significa instaurar el principio del mínimo privilegio, uno de los principios más importantes de la buena ingeniería de software.
- Las funciones por lo regular se invocan en un programa escribiendo el nombre de la función, seguido por un paréntesis izquierdo, a su vez por el *argumento* (o una lista separada por comas de los argumentos) de la función, seguida por un paréntesis derecho.
- El tipo de datos `double` es un tipo de punto flotante similar a `float`. Una variable de tipo `double` puede almacenar un valor de una magnitud mayor y con una precisión mayor de lo que puede almacenar un `float`.
- Cada argumento de una función puede ser una constante, una variable o una expresión.
- Una variable local es conocida sólo en una definición de función. A otras funciones no se les permite conocer los nombres de las variables locales de una función, ni se le permite a ninguna función conocer los detalles de instauración de cualquier otra función.
- El formato general para una definición de función es

```
tipo de valor de regreso nombre de función (lista de parámetros)
{
declaraciones

enunciados
}
```

Capítulo	Ejemplos y ejercicios de recursión
Capítulo 5	Función factorial Funciones Fibonacci Máximo común divisor Sumar dos enteros Multiplicar dos enteros Elevar un entero a una potencia entera Torres de Hanoi **main** recursivo Imprimir entradas del teclado a la inversa Visualización de la recursión
Capítulo 6	Sumar los elementos de un arreglo Imprimir un arreglo Imprimir un arreglo a la inversa Imprimir una cadena a la inversa Verificar si una cadena es un palíndromo Valor mínimo en un arreglo Clasificación de selección Clasificación rápida Búsqueda lineal Búsqueda binaria
Capítulo 7	Ocho reinas Atravesar Maze
Capítulo 8	Imprimir una entrada de cadena desde el teclado a la inversa
Capítulo 12	Inserción de lista enlazada Eliminación de lista enlazada Búsqueda en una lista enlazada Impresión a la inversa de una lista enlazada Inserción de un árbol binario Recorrido en preorden de un árbol binario Recorrido en orden de un árbol binario Recorrido en postorden de un árbol binario

Fig. 5.17 Resumen de ejemplos y ejercicios de recursión del libro.

- El *tipo de valor de regreso*, indica el tipo de valor regresado a la función llamadora. Si la función no regresa un valor, el *tipo de valor de regreso* se declara como **void**. El *nombre de función* es cualquier identificador válido. La *lista de parámetros* es una lista separada por comas, que contiene las declaraciones de las variables que serán pasadas a la función. Si una función no recibe ningún valor, la *lista de parámetros* se declara como **void**. El *cuerpo de función* es el conjunto de declaraciones y de enunciados que la constituyen.
- Los argumentos pasados a una función deberán coincidir en número, tipo y orden con los parámetros en la definición de función.

- Cuando un programa encuentra una función, el control se transfiere desde el punto de invocación a la función llamada, los enunciados de la función llamada se ejecutan, y el control se regresa al llamador.
- Una función llamada puede devolver el control al llamador de una de tres formas diferentes. Si la función no regresa un valor, el control se regresa con la llave derecha de terminación de función, o con la ejecución del enunciado

```
return;
```

- Si la función regresa un valor, el enunciado

```
return expression;
```

devolverá el valor de **expression**.

- Un prototipo de función declara el tipo devuelto por de la función y declara el número, tipos y órdenes de los parámetros que la función espera recibir.
- Los prototipos de función permiten al compilador verificar que las funciones están llamadas correctamente.
- El compilador ignorará los nombres de variable mencionadas en el prototipo de función.
- Cada biblioteca estándar tiene su correspondiente archivo de cabecera, conteniendo los prototipos de función para todas las funciones en dicha biblioteca, así como las definiciones de varias constates simbólicas necesitadas por dichas funciones.
- Los programadores pueden crear e incluir sus propios archivos de cabecera.
- Cuando un argumento se pasa en llamada por valor, se efectúa una *copia* del valor de la variable y la copia se pasa a la función llamada. Las modificaciones a la copia de la función llamada no afectan al valor original de la variable.
- Todas las llamadas en C son llamadas por valor.
- La función **rand** genera un entero entre 0 y **RAND_MAX** definido por el estándar ANSI C a ser por lo menos 32767.
- Los prototipos de función de **rand** y **srand** están contenidas en **<stdlib.h>**.
- Los valores producidos por **rand** pueden ser dimensionados y desplazados para producir valores dentro de un rango específico.
- Para aleatorizar un programa, utilice la función estándar **srand** de biblioteca de C.
- El enunciado **srand** se inserta normalmente en un programa, sólo después de que el programa haya sido en totalidad depurado. Durante la depuración, es mejor omitir **srand**. Esto asegura la capacidad de repetición, que es esencial para comprobar que las correcciones a un programa de generación de números aleatorios funcione correctamente.
- A fin de poder tener números aleatorios sin necesidad de introducir cada vez una semilla, podemos utilizar srand(time(NULL)). La función time regresa el número de segundos a partir del principio del día. El prototipo de función **time** está localizada en la cabecera **<time.h>**
- La ecuación general para dimensionar y desplazar un número aleatorio es

```
n = a + rand() % b;
```

donde **a** es el valor a desplazar (que es igual al primer número del rango deseado de enteros consecutivos), y **b** es el factor de dimensionamiento (que es igual al ancho del rango deseado de enteros consecutivos).

UNIVERSIDAD DE LA REPUBLICA
FACULTAD DE INGENIERIA

- Cada identificador en un programa tiene los atributos de clase de almacenamiento, persistencia, alcance y enlace.
- C proporciona cuatro clases de almacenamiento indicados por los especificadores de clase de almacenamiento (persistencia): **auto**, **register**, **extern** y **static**.
- La persistencia de un identificador es el tiempo que el identificador existe en memoria.
- El alcance de un identificador es dónde puede ser referenciado dicho identificador dentro de un programa.
- El enlace de un identificador determina para un programa de archivos de fuentes múltiples si un identificador es conocido sólo en el archivo fuente actual o en cualquiera de los archivos fuentes mediante declaraciones apropiadas.
- Las variables con persistencia automático son creadas cuando se introduce el bloque en el cual están declaradas, existen mientras el bloque está activo, y se destruyen cuando el bloque se termina. Las variables locales de una función por lo regular tienen persistencia automática.
- El especificador de clase de almacenamiento **register** puede ser colocado delante de una declaración de variable automática, para sugerir que el compilador mantenga la variable en alguno de los registros de hardware de alta velocidad de la computadora. El compilador pudiera ignorar las declaraciones **register**. La palabra reservada **register** puede ser utilizada sólo con variables de persistencia automática.
- Las palabras reservadas **extern** y **static** se utilizan para declarar identificadores de variables y de función de persistencia estática.
- Las variables con persistencia estática, son asignadas e inicializadas una vez cuando el programa inicie su ejecución.
- Existen dos tipos de identificadores para la persistencia estática: los identificadores externos (como las variables y nombres de funciones globales) y las variables locales declaradas utilizando el especificador de clase de almacenamiento **static**.
- Las variables globales se crean colocando declaraciones variables fuera de cualquier definición de función, y conservan sus valores a todo lo largo de la ejecución del programa.
- Las variables locales declaradas como **static** conservan su valor cuando la función en las cuales han sido declaradas se termina.
- Todas las variables numéricas de persistencia estática se inicializan a cero, si no son de forma explícita incializadas por el programador.
- Los cuatro alcances para un identificador son: alcance de función, alcance de archivo, alcance de bloque y alcance de prototipo de función.
- Las etiquetas son los únicos identificadores con alcance de función. Las etiquetas pueden ser utilizadas en cualquier parte en la función en la cual aparecen, pero no pueden ser referenciadas fuera del cuerpo de la función.
- Un identificador declarado fuera de cualquier función tiene un alcance de archivo. Tal identificador es "conocido" en todas las funciones, desde el momento en el cual es declarado el identificador, hasta el final del archivo.
- Los identificadores declarados dentro de un bloque tienen un alcance de bloque. El alcance de bloque termina al terminar la llave derecha (}) de dicho bloque.

- Las variables locales declaradas al principio de una función tienen alcance de bloque, igual que los parámetros de función, que son considerados como variables locales por la función.
- Cualquier bloque puede contener declaraciones variables. Cuando los bloques están anidados, y un identificador de un bloque exterior tiene el mismo nombre que un identificador de un bloque interior, el identificador del bloque exterior está "oculto" hasta que termine el bloque interior.
- Los únicos identificadores con alcance de prototipo de función son aquellos utilizados en la lista de parámetros de un prototipo de función. Los identificadores utilizados en un prototipo de función pueden ser vueltos a utilizar sin ambigüedad en alguna otra parte del programa.
- Una función recursiva es una función que se llama a sí misma, ya sea directa o indirecta.
- Si una función recursiva es llamada con un caso base, la función simplemente regresa un resultado. Si la función es llamada con un problema más complejo, la función divide el problema en dos partes conceptuales: una parte que la función sabe cómo ejecutar y una versión ligeramente más pequeña del problema original. Dado que este nuevo problema asemeja el problema original, la función emite una llamada recursiva, para trabajar en el problema menor.
- Para que termine una recursión, cada vez que la función recursiva se llama a sí misma, con una versión ligeramente más simple del problema original, la secuencia de problemas más y más pequeños debe de converger al caso base. Cuando la función reconoce el caso base, el resultado es regresado a la llamada de función previa, y a continuación sigue una secuencia de regresos hacia atrás, hasta que la llamada original de la función eventualmente devuelve el resultado final.
- El estándar ANSI no especifica el orden en el cual deben ser evaluados los operandos de la mayor parte de los operadores (incluyendo el signo de +). De los muchos operadores de C, el estándar especifica el orden de evaluación de operandos de los operadores **&&**, **||**, el operador coma (**,**) y **?:**. Los primeros tres anteriores son operadores binarios cuyos operandos se evalúan de izquierda a derecha. El último operador es el único operador ternario de C. Su operando más a la izquierda se evalúa primero; si el operando más a la izquierda toma el valor diferente de cero, el operando intermedio se evalúa a continuación ignorándose el operando a la derecha; si el operando más a la izquierda tiene el valor de cero, el tercer operando se evalúa a continuación y el operando de en medio es ignorado.
- Tanto la iteración como la recursión se basan en una estructura de control: la iteración utiliza una estructura de repetición; la recursión una estructura de selección.
- Tanto la iteración como la recursión incluyen repetición: la iteración utiliza una estructura de repetición en forma explícita; la recursión consigue la repetición mediante llamadas de función repetidas.
- La iteración y la recursión ambas incluyen una prueba de terminación: la iteración termina cuando falla la condición de continuación de ciclo; la recursión termina cuando se reconoce el caso base.
- Tanto la iteración como la recursión pueden ocurrir en forma infinita: ocurrirá un ciclo infinito en el caso de la iteración si la prueba de continuación de ciclo nunca se convierte en falsa; ocurrirá la recursión infinita si el paso de recursión no reduce el problema de tal forma que converja en el caso base.
- La recursión invoca el mecanismo en forma repetida y, por lo tanto, sobrecarga las llamadas de función. Esto puede resultar costoso tanto en tiempo de procesador como en espacio de memoria.

Terminología

abstracción
argumento en una llamada de función
almacenamiento automático
persistencia automática
variable automática
especificador de clase de almacenamiento **auto**
caso base en recursión
bloque
alcance de bloque
biblioteca estándar de C
llamada a función
llamada por referencia
llamada por valor
función llamada
llamador
función llamadora
clock
coerción de argumentos
copia de un valor
dividir y vencer
elemento de azar
especificador de clase de almacenamiento **extern**
función factorial
simulación
función
llamada de función
declaración de función
definición de función
prototipo de función
alcance de prototipo de función
alcance de función
variable global
archivo de cabecera
ocultamiento de información
invocar una función
iteración
enlace
variable local
funciones matemáticas de biblioteca
expresión de tipo mixto
programa modular
compilador optimizador
parámetro en una definición de función
principio del mínimo privilegio
función definida por el programador
jerarquía de promoción
números seudoaleatorios
rand
RAND_MAX
hacer aleatorio
generación de números aleatorios
recursión
llamada recursiva
función recursiva
especificador de clase de almacenamiento **register**
return
tipo de valor de regreso
dimensionamiento
especificador de conversión **%s**
alcance
desplazamiento
efectos colaterales

ingeniería de software
reutilización de software
srand
archivos de cabecera de la biblioteca estándar
especificador de clase de almacenamiento **static**
persistencia estática
variable **static**
clases de almacenamiento
especificador de clase de almacenamiento
persistencia
time
unsigned
void

Errores comunes de programación

5.1 Olvidar incluir el archivo de cabecera matemático, al usar las funciones matemáticas de biblioteca, puede causar resultados extraños

5.2 Omitir el tipo de valor de regreso en una definición de función causa un error de sintaxis, si el prototipo de función especifica un regreso de tipo distinto a `int`.

5.3 Olvidar regresar un valor de una función, que se supone debe regresar un valor, puede llevar a errores inesperados. El estándar ANSI indica que el resultado de esta omisión queda indefinido.

5.4 Regresar un valor de una función, cuyo tipo de regreso se ha declarado como `void`, causará un error de sintaxis.

5.5 Declarar parámetros de función del mismo tipo como `float x, y` en vez de `float x, float y`. La declaración de parámetros `float x, y` convertiría de hecho a `y` en un parámetro del tipo `int`, porque `int` es el valor por omisión.

5.6 Es un error de sintaxis colocar un punto y coma después del paréntesis derecho que encierra una lista de parámetros de una definición de función.

5.7 Volver a definir dentro de la función un parámetro de función como variable local es un error de sintaxis.

5.8 Definir una función en el interior de otra función es un error de sintaxis.

5.9 Olvidar el punto y coma al final de prototipo de función hará que ocurra un error de sintaxis.

5.10 Convertir de un tipo de datos superior en la jerarquía de promoción a un tipo inferior, puede modificar el valor del dato.

5.11 Olvidar un prototipo de función generará un error de sintaxis, si el tipo de regreso de la función no es `int` y la definición de función aparece después de la llamada a la función dentro del programa. De lo contrario, el olvidar un prototipo de función puede causar un error en tiempo de ejecución o un resultado inesperado.

5.12 Usar `srand` en vez de `rand` para generar números aleatorios.

5.13 Usar múltiples especificadores de clase de almacenamiento para un identificador. Sólo puede ser aplicado un especificador de clase de almacenamiento a un identificador.

5.14 Utilizar accidentalmente el mismo nombre para un identificador en un bloque interno, que se haya usado para un identificador en un bloque externo, cuando de hecho, el programador desea que durante la duración del bloque interno el identificador del bloque externo esté activo.

5.15 Olvidar el regresar un valor de una función recursiva cuando se requiere de uno.

5.16 Omitir ya sea el caso base, o escribir el paso de recursión en forma incorrecta, de tal forma que no converja al caso base, causando recursión infinita, y agotando de forma eventual la memoria. Esto es análogo al problema de un ciclo infinito en una solución iterativa (no recursiva). La recursión infinita también puede ser causada al proporcionarle una entrada no esperada.

5.17 Escribir programas que dependan del orden de evaluación de los operandos de operadores distintos que `&&`, `||`, `?:`, y el operador coma (`,`) puede llevar a errores, porque los compiladores no necesariamente evalúan los operandos en el orden en que los programas esperan.

5.18 Hacer que accidentalmente una función no recursiva se llame a sí misma, ya sea directa o indirecta a través de otra función.

Prácticas sanas de programación

5.1 Familiarícese con la amplia colección de funciones de la biblioteca estándar ANSI C.

5.2 Incluya el archivo de cabecera de matemáticas utilizando la directiva de preprocesador `#include <math.h>` cuando esté utilizando funciones de la biblioteca de matemáticas.

5.3 Coloque una línea en blanco entre definiciones de función, para separarlas y para mejorar la legibilidad del programa.

5.4 Aun cuando un tipo de regreso omitido resulte en `int` por omisión, declare siempre en forma explícita el tipo de regreso. Sin embargo, normalmente, se omite el tipo de regreso correspondiente a `main`.

5.5 Incluya en la lista de parámetros el tipo de cada parámetro, inclusive si algún parámetro es del tipo por omisión `int`.

5.6 Aunque hacerlo no es incorrecto, no utilice los mismos nombres para argumentos pasados a una función y parámetros correspondientes de la definición de función. Con ello ayuda a evitar ambigüedad.

5.7 Seleccionar nombres significativos para funciones y nombres significativos para parámetros, hace que sean más legibles los programas y ayuda a evitar un uso de comentarios excesivo.

5.8 Incluya prototipos de función para todas las funciones para aprovechar las capacidades de C de verificación de tipo. Utilice las directivas de preprocesador `#include` para obtener prototipos para las funciones estándar de biblioteca a partir de los archivos de cabecera de las bibliotecas apropiadas. También utilice `#include` para obtener archivos de cabecera que contengan prototipos de función utilizados por usted y los miembros de su grupo.

5.9 Los nombres de los parámetros a veces se incluyen en los prototipos de función por razones de documentación. El compilador ignora estos nombres.

5.10 Las variables utilizadas sólo en una función particular deberían ser declaradas como variables locales en esa función, en vez de variables externas.

5.11 Evite nombres variables que oculten nombres en alcances externos. Esto se puede conseguir dentro de un programa evitando el uso de identificadores duplicados.

Sugerencias de portabilidad

5.1 Usar funciones de la biblioteca estándar ANSI C auxilia a que los programas sean más portables.

5.2 Los programas que dependen del orden de evaluación de los operandos o de operadores, diferentes a `&&`, `||`, `?:`, y el operador coma (`,`) pueden funcionar de forma distinta sobre sistemas con compiladores diferentes.

Sugerencias de rendimiento

5.1 El almacenamiento automático es una forma de conservar memoria, porque las variables automáticas existen sólo cuando son necesitadas. Son creadas al introducirse la función en la cual son declaradas, y son destruidas cuando se sale de dicha función.

5.2 El especificador de clase de almacenamiento `register` puede ser colocado antes de una declaración de variable automática, para sugerir que el compilador conserve la variable en uno de los registros de alta velocidad del hardware de la computadora. Si se puede mantener en registros de hardware las variables intensamente utilizadas como son contadores y totales, puede ser eliminada la sobrecarga correspondiente a cargar en forma repetida las variables de las memorias a los registros y almacenar los resultados de vuelta en memoria.

5.3 A menudo, son innecesarias las declaraciones `register`. Los compiladores optimizadores de hoy día son capaces de reconocer variables de uso frecuente, y pueden decidir colocarlos en registro, sin necesidad de una declaración `register` proveniente del programador.

5.4 Evite programas recursivos de tipo fibonacci que resultan en una "explosión" exponencial de llamadas.

5.5 Evite el uso de la recursión cuando se requiera de rendimiento. Las llamadas recursivas toman tiempo y consumen memoria adicional.

5.6 Un programa muy funcionalizado en comparación con uno monolítico (es decir de una pieza) sin funciones potencialmente hace grandes cantidades de llamadas de función y estas consumen tiempo de ejecución en el procesador de una computadora. Pero los programas monolíticos son difíciles de programar, probar, depurar, mantener y de modificar.

Observaciones de ingeniería de software

5.1 Evite reinventar la rueda. Siempre que sea posible, utilice funciones estándar de biblioteca ANSI C, en vez de escribir nuevas funciones. Esto reduce el tiempo de desarrollo del programa.

5.2 En programas que contengan muchas funciones, `main` deberá de ser organizada como un grupo de llamadas a funciones que ejecuten la mayor parte del trabajo del programa.

5.3 Cada función debería limitarse a ejecutar una tarea sencilla y bien definida, y el nombre de la función debería expresar dicha tarea con claridad. Ello facilitaría la abstracción y promovería la reutilización del software.

5.4 Si no puede elegir un nombre conciso, que exprese lo que la función ejecuta, es probable que su función está intentando ejecutar demasiadas tareas diversas. A menudo es mejor dividir dicha función en varias funciones más pequeñas.

5.5 Una función no debería tener una longitud mayor que una página. Aún mejor, una función no debería ser más larga que media página. Las funciones pequeñas promueven la reutilización del software.

5.6 Los programas deberían ser escritos como recopilaciones de pequeñas funciones. Esto haría que los programas fuesen más fáciles de escribir, depurar, mantener y de modificar.

5.7 Una función que requiera un gran número de parámetros quizás esté ejecutando demasiadas tareas. Piense en dividir esta función en funciones más pequeñas, que ejecuten las tareas por separado. El encabezado de función, si es posible, debería poder caber en una línea.

5.8 El prototipo de función, el encabezado de función y las llamadas de función deberán todas estar de acuerdo en lo que se refiere al número, tipo y orden de argumentos y de parámetros, así como en el tipo de valor de regreso.

5.9 Un prototipo de función, colocado fuera de una definición de función, se aplica a todas las llamadas de función que aparezcan dentro del archivo después del prototipo de función. Un prototipo de función colocado en una función, se aplica sólo a las llamadas efectuadas en esa función.

5.10 El almacenamiento automático es otra vez un ejemplo del principio del menor privilegio. ¿Por qué tendrían que estar las variables almacenadas en memoria y accesibles cuando de hecho no son necesarias?

5.11 La declaración de una variable como global en vez de como local, permite que ocurran efectos colaterales no deseados cuando una función que no requiere de acceso a la variable la modifica accidental o maliciosamente. En general, el uso de las variables globales debería ser evitado, excepto en ciertas situaciones, con requerimientos de rendimiento únicos (como se analizan en el capítulo 14).

5.12 Cualquier problema que puede ser resuelto en forma recursiva, también puede ser resuelto en forma iterativa (no recursiva). Por lo regular se escoge un enfoque recursivo en preferencia a uno iterativo cuando el enfoque recursivo es más natural al problema y resulta en un programa que sea más fácil de comprender y de depurar. Otra razón para seleccionar una solución recursiva, es que la solución iterativa pudiera no resultar aparente.

5.13 La funcionalización de los programas de una forma nítida y jerárquica promueve buena ingeniería de software. Pero tiene un costo.

Ejercicios de autoevaluación

5.1 Llene cada uno de los siguientes espacios en blanco:

a) En C un módulo de programas se conoce como un _____.
b) Una función se invoca mediante un _____.
c) Una variable que es conocida solo dentro de la función en la cual está definida se conoce como una _____.
d) Un enunciado _____ en una función llamada se utiliza para pasar el valor de una expresión de regreso a la función llamada.
e) La palabra reservada _____ se utiliza en un encabezado de función para indicar que una función no regresará un valor o para indicar que la función no contiene parámetros.
f) El _____ de un identificador es la parte del programa en el cual se puede utilizar dicho identificador.
g) Las tres distintas maneras para regresar control a partir de una función llamada hacia su llamador son _____, _____, y _____.
h) Una _____ le permite al compilador verificar el número, tipos y orden de los argumentos pasados a una función.
i) La función _____ se utiliza para producir números aleatorios.

j) La función _____ se utiliza para establecer la semilla de números aleatorios, a fin de hacer aleatorio un programa.
k) Los especificadores de clase de almacenamiento son _____, _____, _____, y _____.
l) Las variables declaradas en un bloque o en la lista de parámetros de una función se suponen son de la clase de almacenamiento _____, a menos de que se especifique lo contrario.
m) El especificador de clase de almacenamiento _____ es una recomendación para el compilador para que almacene una variable en uno de los registros de la computadora.
n) Una variable declarada por fuera de cualquier bloque o de función es una variable _____.
o) Para que la variable local en una función conserve su valor entre llamadas a la función, debe de ser declarada como de especificador de clase de almacenamiento _____.
p) Los cuatro posibles alcances de un identificador son _____, _____, _____, y _____.
q) Una función que se llama a sí misma, ya sea directa o indirecta es una función _____.
r) Una función recursiva típica consta de dos componentes: uno que proporciona una forma para que la recursión se termine mediante la prueba buscando un caso _____, y otra que expresa el problema como una llamada recursiva para un problema ligeramente más simple que la llamada original.

5.2 Para el programa siguiente, indique el alcance (ya sea alcance de función, archivo, bloque o de prototipo de función) de cada uno de los elementos siguientes:

a) La variable **x** en **main**.
b) La variable **y** en **cube**.
c) La función **cube**.
d) La función **main**.
e) El prototipo de función correspondiente a **cube**.
f) El identificador **y** en el prototipo de función correspondiente a **cube**.

```
#include <stdio.h>
int cube(int y);

main()
{
   int x;

   for (x = 1; x <= 10; x++)
      printf("%d\n", cube(x));
}

int cube(int y)
{
   return y * y * y;
}
```

5.3 Escriba un programa que compruebe si los ejemplos de las llamadas de funciones matemáticas de biblioteca mostradas en la figura 5.2 de verdad producen los resultados indicados.

5.4 Proporcione el encabezamiento de función para cada una de las función siguientes.

a) Función **hypotenuse** que toma dos argumentos de punto flotante de doble precisión, **side1** y **side2**, y regresa un resultado de punto flotante de doble precisión.
b) Función **smallest** que toma tres enteros, **x**, **y**, **z** y regresa un entero.
c) Función **instructions** que no recibe ningún argumento y no regresa ningún valor. (Nota: se utilizan estas funciones por lo regular para mostrar instrucciones a un usuario.)
d) Función **intToFloat** que toma un argumento entero, **number** y regresa un resultado en punto flotante.

5.5 Proporcione el prototipo de función de cada uno de los siguientes:

a) La función descrita en el ejercicio 5.4a

b) La función descrita en el ejercicio 5.4b
c) La función descrita en el ejercicio 5.4c
d) La función descrita en el ejercicio 5.4d

5.6 Escriba una declaración para cada uno de los siguientes:

a) **count** entero que debe de ser conservado en un registro. Inicialice **count a 0.**
b) Variable de punto flotante **lastVal** que debe de conservar su valor entre llamadas a la función en la cual ha sido definida.
c) Entero externo **number** cuyo alcance debe de quedar restringido al resto del archivo en el cual ha sido definido.

5.7 Encuentre el error en cada uno de los segmentos siguientes de programa y explique cómo puede corregirse dicho error (vea también el ejercicio 5.50):

a)
```
int g(void) {
   printf("Inside function g\n");

   int h(void) {
      printf("Inside function h\n");
   }
}
```

b)
```
int sum(int x, int y) {
   int result;

   result = x + y;
}
```

c)
```
int sum(int n) {
   if (n == 0)
      return 0;
   else
      n + sum(n - 1);
}
```

d)
```
void f(float a); {
   float a;

   printf("%f", a);
}
```

e)
```
void product(void) {
   int a, b, c, result;

   printf("Enter three integers: ")
   scanf("%d%d%d", &a, &b, &c);
   result = a * b * c;
   printf("Result is %d", result);
   return result;
}
```

Respuestas a los ejercicios de autoevaluación

5.1 a) Función. b) Llamada de función. c) Variable local. d) **return**. e) **void**. f) Alcance. g) **return**; o bien **return expression**; o bien al encontrar la llave de cierre izquierda de una función. h) Prototipo de función. i) **rand**. j) **srand**. k) **auto**, **register**, **extern**, **static**. l) Automático. m) **register**. n) External, global. o) **static**. p) Alcance de función, alcance de archivo, alcance de bloque, alcance de prototipo de función. q) Recursivo. r) Base.

5.2 a) Alcance de bloque. b) Alcance de bloque. c) Alcance de archivo. d) Alcance de archivo. e) Alcance de archivo. f) Alcance de prototipo de función.

5.3

```
/* Testing the math library functions */
#include <stdio.h>
#include <math.h>

main()
{
   printf("sqrt(%.1f) = %.1f\n", 900.0, sqrt(900.0));
   printf("sqrt(%.1f) = %.1f\n", 9.0, sqrt(9.0));
   printf("exp(%.1f) = %f\n", 1.0, exp(1.0));
   printf("exp(%.1f) = %f\n", 2.0, exp(2.0));
   printf("log(%f) = %.1f\n", 2.718282, log(2.718282));
   printf("log(%f) = %.1f\n", 7.389056, log(7.389056));
   printf("log10(%.1f) = %.1f\n", 1.0, log10(1.0));
   printf("log10(%.1f) = %.1f\n", 10.0, log10(10.0));
   printf("log10(%.1f) = %.1f\n", 100.0, log10(100.0));
   printf("fabs(%.1f) = %.1f\n", 13.5, fabs(13.5));
   printf("fabs(%.1f) = %.1f\n", 0.0, fabs(0.0));
   printf("fabs(%.1f) = %.1f\n", -13.5, fabs(-13.5));
   printf("ceil(%.1f) = %.1f\n", 9.2, ceil(9.2));
   printf("ceil(%.1f) = %.1f\n", -9.8, ceil(-9.8));
   printf("floor(%.1f) = %.1f\n", 9.2, floor(9.2));
   printf("floor(%.1f) = %.1f\n", -9.8, floor(-9.8));
   printf("pow(%.1f, %.1f) = %.1f\n", 2.0, 7.0, pow(2.0, 7.0));
   printf("pow(%.1f, %.1f) = %.1f\n", 9.0, 0.5, pow(9.0, 0.5));
   printf("fmod(%.3f/%.3f) = %.3f\n",
      13.675, 2.333, fmod(13.675, 2.333));
   printf("sin(%.1f) = %.1f\n", 0.0, sin(0.0));
   printf("cos(%.1f) = %.1f\n", 0.0, cos(0.0));
   printf("tan(%.1f) = %.1f\n", 0.0, tan(0.0));
}
```

```
sqrt(900.0) = 30.0
sqrt(9.0) = 3.0
exp(1.0) = 2.718282
exp(2.0) = 7.389056
log(2.718282) = 1.0
log(7.389056) = 2.0
log10(1.0) = 0.0
log10(10.0) = 1.0
log10(100.0) = 2.0
fabs(13.5) = 13.5
fabs(0.0) = 0.0
fabs(-13.5) = 13.5
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor (-9.8) = -10.0
pow(2.0, 7.0) = 128.0                    continúa
```



```
pow(9.0, 0.5) = 3.0                      continuación
fmod(13.675/2.333) = 2.010
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0
```

5.4 a) `double hypotenuse(double side1, double side2)`
b) `int smallest(int x, int y, int z)`
c) `void instructions(void)`
d) `float intToFloat(int number)`

5.5 a) `double hypotenuse(double, double);`
b) `int smallest(int, int, int);`
c) `void instructions(void)`
d) `float intToFloat(int)`

5.6 a) `register int count = 0;`
b) `static float lastVal;`
c) `static int number;`
Nota: Esto aparecería por fuera de cualquier definición de función.

5.7 a) Error: la función `h` está definida en la función `g`.
Corrección: mueva la definición de `h` fuera de la definición de `g`
b) Error: la función debe supuestamente regresar un entero, pero no lo hace.
Corrección: borrar la variable `result` y colocar el siguiente enunciado en la función:

```
return x + y;
```

c) Error: el resultado de `n + sum(n-1)` no es regresado; `sum` regresa un resultado inadecuado.
Corrección: volver a escribir el enunciado en la cláusula `else` como

```
return n + sum(n - 1);
```

d) Error: punto y coma, antes del paréntesis derecho que encierra la lista de parámetros, y redefinir el parámetro `a` en la definición de función.
Corrección: Borre el punto y coma después del paréntesis derecho de la lista de parámetros, y borre la declaración `float a;`
e) Error: la función regresa un valor cuando no se supone que lo haga.
Corrección: eliminar el enunciado `return`.

Ejercicios

5.8 Muestre el valor de `x` después de que se hayan ejecutado los siguientes enunciados:
a) `x = fabs(7.5)`
b) `x = floor(7.5)`
c) `x = fabs(0.0)`
d) `x = ceil(0.0)`
e) `x = fabs(-6.4)`
f) `x = ceil(-6.4)`
g) `x = ceil(-fabs(-8+floor(-5.5)))`

5.9 Un estacionamiento público carga $2.00 de estacionamiento mínimo por las primeras tres horas. El estacionamiento carga $0.50 adicíonales por cada hora o *parte de la misma* en exceso de tres horas. El cargo máximo para cualquier periodo de 24 horas es $10.00. Suponga que no existe ningún vehículo que se quede más de 24 horas a la vez. Escriba un programa que calcule e imprima los cargos por estacionamiento

para cada uno de tres clientes que ayer estacionaron sus automóviles en este garaje. Deberá de introducir las horas de estacionamiento para cada uno de los clientes. Su programa deberá imprimir los resultados en un formato tabular nítido, y deberá calcular e imprimir el total de los ingresos de ayer. El programa deberá utilizar la función **calculate-Charges** para determinar los cargos de cada cliente. Sus salidas deberán de aparecer en el formato siguiente:

```
Car     Hours     Charge
1         1.5       2.00
2         4.0       2.50
3        24.0      10.00
TOTAL    29.5      14.50
```

5.10 Una aplicación de la función **floor** es redondear un valor al entero más cercano. El enunciado

```
y = floor(x + .5);
```

redondeará el número **x** al entero más cercano, y asignará el resultado a **y**. Escriba un programa que lea varios números y que utilice el enunciado anterior para redondear cada uno de estos números al entero más cercano. Para cada número procesado, imprima tanto el número original como el número redondeado.

5.11 La función **floor** puede ser utilizada para redondear un número a una cantidad específica de lugares decimales. El enunciado

```
y = floor(x * 10 + .5) / 10;
```

redondea **x** a la posición de décimos (la primera posición a la derecha del punto decimal). El enunciado

```
y = floor(x * 100 + .5) / 100;
```

redondea **x** a la posición de las centésimos (es decir, a la segunda posición a la derecha del punto decimal). Escriba un programa que defina cuatro funciones para redondear un número **x** de varias formas:

a) **roundToInteger(number)**
b) **roundToTenths(number)**
c) **roundToHundreths(number)**
d) **roundToThousandths(number)**

Para cada uno de los valores leídos, su programa debe imprimir el valor original, el número redondeado para el entero más cercano, el número redondeado a la décima más cercana, el número redondeado a la centésima más cercana, y el número redondeado a la milésima más cercana.

5.12 Conteste cada una de las preguntas siguientes:

a) ¿Qué significa seleccionar números "al azar"?
b) ¿Por qué la función **rand** es útil para la simulación de juegos de azar?.
c) ¿Por qué tendría que hacer aleatorio un programa utilizando a **srand**? ¿y bajo qué circunstancias no sería deseable hacerlo?.
d) ¿Por qué resulta necesario ha menudo el dimensionar y/o desplazar los valores producidos por **rand**?
e) ¿Por qué es una técnica útil la simulación computarizada de situaciones del mundo real?

5.13 Escriba enunciados que asignen números enteros aleatorios a la variable n en los rangos siguientes:

a) $1 \le n \le 2$
b) $1 \le n \le 100$
c) $0 \le n \le 9$
d) $1000 \le n \le 1112$
e) $-1 \le n \le 1$
f) $-3 \le n \le 11$

5.14 Para cada uno de los conjuntos siguientes de enteros, escriba un solo enunciado que imprima un número al azar del conjunto.

a) 2, 4, 6, 8, 10.
b) 3, 5, 7, 9, 11.
c) 6, 10, 14, 18, 22.

5.15 Defina una función **hypotenuse** que calcule la longitud de la hipotenusa de un triángulo rectángulo, cuando son conocidos los otros dos lados. Utilice esta función en un programa para determinar la longitud de la hipotenusa de los triángulos siguientes. La función debe de tomar dos argumentos del tipo **double** y regresar la hipotenusa también como **double**.

Triángulo	Lado 1	Lado 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

5.16 Escriba una función **integerPower(base, exponent)** que devuelva el valor de

$base^{exponente}$

Por ejemplo, **integerPower(3,4) = 3 * 3 * 3 * 3**. Suponga que **exponent** es un entero positivo, no cero, y **base** es un entero. La función **integerPower** deberá utilizar **for** para controlar el cálculo. No utilice ninguna función matemáticas de biblioteca.

5.17 Escriba una función **multiple** que determine para un par de enteros, si el segundo de ellos es múltiplo del primero. La función debe tomar dos argumentos enteros y regresar **1** (verdadero) si el segundo es un múltiplo del primero, y **0** (falso) de no ser así. Utilice esta función en un programa que introduzca una serie de pares de enteros.

5.18 Escriba un programa que introduzca una serie de enteros y que los pase uno a la vez a la función **even** que utiliza el operador de módulo, para determinar si el entero es par. La función deberá tomar un argumento entero y regresar **1** si el entero es par, y **0** si no lo es.

5.19 Escriba una función que despliegue en el margen izquierdo de la pantalla un cuadrado sólido de asteriscos, cuyo costado o lado está especificado en el parámetro entero **side**. Por ejemplo, si **side** es **4**, la función mostrará

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

5.20 Modifique la función creada en el ejercicio 5.19 para formar el cuadrado en base a cualquier carácter que esté contenido en el parámetro de carácter **fillCharacter**. Por lo tanto si **side** es **5** y **fillCharacter** es "**#**" entonces esta función debería imprimir.

```
#####
#####
#####
#####
#####
```

5.21 Utilice técnicas similares a las desarrolladas en los ejercicios 5.19 y 5.20 para producir un programa que grafique una amplia gama de formas

5.22 Escriba segmentos de programa que lleven a cabo cada uno de ellos lo siguiente:

a) Calcule la parte entera del cociente cuando el entero `a` se divide por el entero `b`.
b) Calcule el residuo entero cuando el entero `a` es dividido por el entero `b`.
c) Utilice las porciones de programa desarrolladas en `a)` y en `b)` para describir una función que introduzca un entero entre `1` y `32767` y lo imprima como una serie de dígitos, estando separado cada par de dígitos por dos espacios. Por ejemplo el entero `4562` deberá ser impreso como

```
4  5  6  2
```

5.23 Escriba una función que obtenga el tiempo como tres argumentos enteros (para horas, minutos y segundos) y regrese el número de segundos desde la última vez que el reloj "llegó a las 12". Utilice esta función para calcular la cantidad de tiempo en segundos entre dos horas, cuando ambas estén dentro de un ciclo de 12 horas del reloj.

5.24 Ponga en marcha las siguientes funciones enteras:

a) La función `celsius` que regresa el equivalente Celsius de una temperatura en Fahrenheit.
b) La función `fahrenheit` que regresa el equivalente en Fahrenheit de una temperatura en Celsius.
c) Utilice ambas funciones para escribir un programa que imprima gráficas mostrando los equivalentes Fahrenheit de todas las temperaturas Celsius desde 0 hasta 100 grados, y los equivalentes Celsius de todas las temperaturas Fahrenheit entre 32 y 212 grados. Imprima las salidas en un formato tabular nítido, que minimice el número de líneas de salida manteniéndose legible.

5.25 Escriba una función que regrese el más pequeño de tres números de punto flotante.

5.26 Un número entero se dice que se trata de un *número perfecto* si sus factores, incluyendo a 1 (pero excluyendo en el número mismo), suman igual que el número. Por ejemplo, 6 es un número perfecto porque 6 = 1+2+3. Escriba una función `perfect` que determine si el parámetro `number` es un número perfecto. Utilice esta función en un programa que determine e imprima todos los números perfectos entre 1 y 1000. Imprima los factores de cada número perfecto para confirmar que el número de verdad es perfecto. Ponga en acción la potencia de su computadora para probar números más grandes que 1000.

5.27 Se dice que un entero es *primo* si es divisible sólo entre 1 y sí mismo. Por ejemplo, 2, 3, 5, y 7 son primos, pero 4, 6, 8 y 9 no lo son.

a) Escriba una función que determine si un número es primo.
b) Utilice esta función en un programa que determine e imprima todos los números primos entre 1 y 10,000. ¿Cuántos de estos 10,000 números tendrá que probar verdaderamente antes de estar seguro de que se han encontrado todos los números primos?
c) Inicialmente pudiera pensar que $n/2$ es el límite superior para el cual debe usted probar para ver si un número es primo, pero sólo necesita llegar hasta la raíz cuadrada de n. ¿Por qué? Vuelva a escribir el programa, y ejecútelo de ambas formas. Estime la mejoría en rendimiento.

5.28 Escriba una función que tome un valor entero y regrese el número con sus dígitos invertidos. Por ejemplo, dado el número 7631, la función debería regresar 1367.

5.29 El *máximo común divisor* de dos enteros es el entero más grande que divide de forma uniforme cada uno de los dos números. Escriba una función `gcd` que regrese el máximo común divisor de dos enteros.

5.30 Escriba una función `qualityPoints` que introduzca el promedio de un alumno y regrese 4 si el promedio es entre 90 - 100, 3 si el promedio es entre 80 - 89, 2 si el promedio es entre 70 - 79, 1 si el promedio está entre 60 - 69 0 si el promedio es menor de 60.

5.31 Escriba un programa que simule lanzar una moneda. Para cada lanzamiento de la moneda el programa deberá imprimir `Heads` o `Tails`. Permita que el programa lance la moneda 100 veces, y cuente el número de veces que aparece alguno de los dos lados de la moneda. Imprima los resultados. El programa deberá llamar una función separada o distinta `flip`, que no toma argumentos y que regresa `0` para las caras, y `1` para las cruces. *Nota*: si el programa simula en forma realista el lanzamiento de la moneda, entonces cada cara de la misma deberá aparecer aproximadamente la mitad del tiempo para un total de aproximadamente 50 caras y 50 cruces.

5.32 Las computadoras están jugando un papel creciente en la educación. Escriba un programa que ayudaría a un alumno de escuela primaria a aprender a multiplicar. Utilice `rand` para producir dos enteros positivos de un dígito. A continuación debería escribir una pregunta como la siguiente:

```
How much is 6 times 7?
```

A continuación el alumno escribe la respuesta. Su programa verifica la respuesta del alumno. Si es correcta, imprime `"Very good!"` y a continuación solicita otra multiplicación. Si la respuesta es incorrecta, imprimirá "`No. Please try again.`" y a continuación permitirá que el alumno vuelva a intentar la misma pregunta en forma repetida, hasta que al final la conteste correctamente.

5.33 La utilización de las computadoras en la educación se conoce como *instrucción asistida por computadora* (CAI). Un problema que se desarrolla en los entornos CAI es la fatiga del alumno. Esto puede ser eliminado variando el diálogo de la computadora para retener la atención del alumno. Modifique el programa del ejercicio 5.32 de tal forma que los comentarios que se impriman para respuesta correcta y cada respuesta incorrecta sean como sigue:

Respuestas a las contestaciones correctas

```
Very good!
Excellent!
Nice work!
Keep up the good work!
```

Respuestas a las contestaciones incorrectas

```
No. Please try again.
Wrong. Try once more.
Don't give up!
No. Keep trying.
```

Utilice el generador de números aleatorios para escoger el número de 1 a 4 y seleccionar una respuesta apropiada para cada una de las contestaciones. Utilice una estructura `switch` con enunciados `printf` para emitir las respuestas.

5.34 Sistemas más avanzados de instrucción asistida por computadora vigila el rendimiento del alumno a lo largo de un periodo de tiempo. La decisión para empezar un nuevo tema, a menudo se basa en el léxico del alumno en relación con temas anteriores. Modifique el programa del ejercicio 5.33 para contar el número de respuestas correctas e incorrectas escritas por el estudiante. Una vez que el alumno escriba 10 respuestas, su programa deberá calcular el porcentaje de respuestas correctas. Si el porcentaje es menor de 75%, su programa deberá de imprimir "`Please ask you instructor for extra help`" y a continuación terminar.

5.35 Escriba un programa en C que juegue el juego de "adivine el número" como sigue: su programa escoge el número que se debe de adivinar seleccionando un entero al azar en el rango del 1 al 1000. El programa a continuación escribe:

```
I have a number between 1 and 1000.
Can ycu guess my number?
Please type your first guess.
```

El jugador entonces escribe su primera estimación. El programa responde con una de las siguientes:

```
1. Excellent! You guessed the number!
   Would you like to play again (y or n)?
2. Too low. Try again.
3. Too high. Try again.
```

Si la adivinanza del jugador es incorrecta, su programa deberá de ciclar hasta que el jugador obtiene al final el número correcto. Su programa debe de insistir en indicarle al jugador `Too high` o bien `Too low` para ayudarlo a centrarse a la contestación correcta. Nota: la técnica de búsqueda empleada en este problema es conocida como *búsqueda binaria*. Diremos más en relación con lo anterior en el siguiente problema.

5.36 Modifique el programa del ejercicio 5.35 para contar el número de veces que intenta adivinar el jugador. Si el número es 10 o menor, imprima `Either you know the secret or you got lucky!` Si el jugador adivina el número en 10 intentos, entonces imprima `Ahah! You Know the secret!` Si el jugador hace más de 10 intentos, entonces imprima `You should be able to do better!` ¿Por qué debería de tomar no más de 10 intentos? "Bueno", con cada una de las estimaciones buenas, el jugador tendría que estar en posición de eliminar la mitad de los números. Ahora muestre porque cualquier número del 1 al 1000 puede ser encontrado en 10 o menos intentos.

5.37 Escriba una función recursiva `power (base, exponente)` que al ser invocada regrese

$$base^{exponente}$$

Por ejemplo, `power(3, 4) = 3 * 3 * 3 * 3`. Suponga que `exponente` es un entero mayor o igual a 1. *Sugerencia:* el paso de recursión deberá de utilizar la relación

$$base^{exponente} = base \cdot base^{exponente-1}$$

y la condición de terminación ocurrirá cuando `exponente` es igual a `1` porque

$$base^{1} = base$$

5.38 La serie Fibonacci

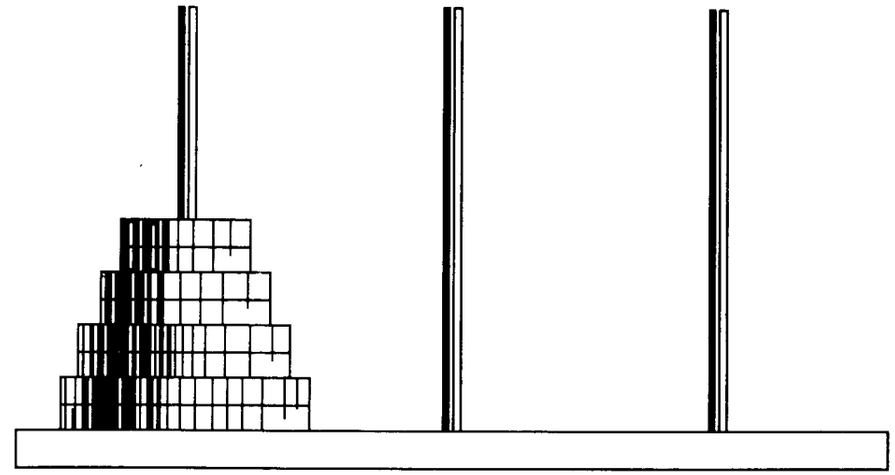
0, 1, 1, 2, 3, 5, 8, 13, 21,...

empieza con los términos 0 y 1 y tiene la propiedad que cada término siguiente es la suma de los dos términos precedentes. a) Escriba una función *no recursiva* `fibonacci(n)` que calcule el número Fibonacci de orden **n**. b) Determine el número Fibonacci más grande que pueda ser impreso en su sistema. Modifique el programa de la parte a) para utilizar `double` en vez de `int`, a fin de calcular y regresar números Fibonacci. Deje que el programa cicle hasta que falle debido a valores en exceso altos.

5.39 (*Torres de Hanoi*) Todos los científicos de cómputo incipientes deben de enfrentarse con ciertos problemas clásicos, y las Torres de Hanoi (vea la figura 5.18) es uno de los más famosos. Dice la leyenda que en un templo del Lejano Este, los monjes están intentando mover una pila de discos de una estaca hacia otra. La pila inicial tenía 64 discos ensartados en una estaca y acomodados de la parte inferior a la superior en tamaño decreciente. Los monjes están intentando mover la pila de esta estaca a la segunda con las limitaciones que exactamente un disco debe de ser movido a la vez, y en ningún momento se puede colocar un disco mayor por encima de un disco menor. Existe una tercera estaca disponible para almacenamiento temporal de discos. Se supone que cuando los monjes terminen su tarea llegará el fin del mundo, por lo cual para nosotros existe poca motivación en ayudarles en sus esfuerzos.

Supongamos que los monjes están intentando mover los discos de la estaca 1 a la estaca 3. Deseamos desarrollar un algoritmo que imprima la secuencia precisa de las transferencias disco a disco entre estacas.

Si fuéramos a enfocar este problema con métodos convencionales, nos encontraríamos rápidamente enmarañados y sin esperanza de poder manejar los discos. En vez de ello, si atacamos el problema teniendo en mente la recursión, de inmediato se vuelve manejable. El mover n discos puede ser visualizado en términos de sólo mover n-1 discos (y de ahí la recursión), como sigue:



UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE DOCUMENTACIÓN Y BIBLIOTECA

Fig. 5.18 Las Torres de Hanoi para el caso de cuatro discos.

1. Mover n-1 discos de la estaca 1 a la estaca 2, utilizando a la estaca 3 como un área de almacenamiento temporal.
2. Mover el último disco (el más grande) de la estaca 1 a la estaca 3.
3. Mover los n-1 discos de la estaca 2 a la estaca 3, utilizando la estaca 1 como área de almacenamiento temporal.

El proceso termina cuando la última tarea consiste en mover el disco n = 1, es decir, el caso base. Esto se lleva a cabo en forma trivial moviendo el disco, sin necesidad de utilizar el área temporal de almacenamiento.

Escriba un programa para resolver el problema de las Torres de Hanoi. Utilice una función recursiva con cuatro parámetros:

1. El número de discos a moverse
2. La estaca en la cual se acumularán estos discos al inicio
3. La estaca a la cual esta pila de discos se moverá
4. La estaca a utilizarse como área de almacenamiento temporal

Su programa deberá imprimir las instrucciones precisas que deberán seguirse para mover los discos de la estaca de arranque a la estaca destino. Por ejemplo, para mover una pila de tres discos de la estaca 1 a la estaca 3, su programa deberá imprimir la serie siguiente de movimientos:

1 → 3 (Esto significa mover un disco de la estaca 1 a la estaca 3)
1 → 2
3 → 2
1 →3
2 → 1
2 → 3
1 → 3

5.40 Cualquier programa que puede ser organizado en forma recursiva, puede ser organizado también en forma iterativa, aunque algunas veces con mayor dificultad y con menor claridad. Intente escribir una versión iterativa de las Torres de Hanoi. Si tiene éxito, compare su versión iterativa con la versión recursiva que desarrolló en el ejercicio 5.39. Investigue los temas correspondientes a rendimiento, claridad y a su capacidad para demostrar la corrección de los programas.

5.41 (Visualización de la recursión). Es interesante poder observar la recursión en "acción". Modifique la función factorial de la figura 5.14 para imprimir su variable local y su parámetro de llamada recursiva. Para cada llamada recursiva, despliegue las salidas en una línea por separado, y añada un nivel de sangría. Haga todo lo posible para que las salidas resulten claras, interesantes y significativas. Su meta aquí es diseñar e implantar un formato de salida que le ayude a una persona a mejor comprender la recursión. Quizás desee añadir capacidades de despliegue, tales como éstas, a los muchos otros ejemplos de recursión así como a los ejercicios a todo lo largo del texto.

5.42 El máximo común divisor de los enteros **x** e **y** es el entero más grande que divide en forma completa tanto a **x** como a **y**. Escriba una función recursiva **gcd** que regrese el máximo común divisor de **x** y de **y**. El **gcd** de **x** y de **y** se define en forma recursiva como sigue: Si **y** es igual a **0**, entonces **gcd**(de **x, y**) es **x**; de lo contrario **gcd**(de **x, y**) es igual a **gcd(y, x%y)** donde % es el operador de módulo.

5.43 ¿Es posible llamar recursivamente a **main**? Escriba un programa que contenga una función **main**. Incluya la variable local **count** de tipo **static** incializada a 1. Postincremente e imprima el valor de **count** cada vez que **main** es llamada. Ejecute su programa. ¿Qué ocurre?.

5.44 Los ejercicios 5.32 al 5.34 desarrollaron un programa de instrucciones asistido por computadora, para enseñar la multiplicación a un alumno de escuela primaria. Este ejercicio sugiere mejoras a dicho programa.

a) Modifique el programa para permitir al usuario la introducción de una capacidad de nivel de grado. Un nivel de grado 1 significa que utilice en los problemas sólo números de un dígito, un nivel de grado 2 significa la utilización de números de hasta dos dígitos de grande, etcétera.

b) Modifique el programa para permitirle al usuario escoger el tipo de problemas aritméticos que él o ella desee estudiar. Una opción 1 significa sólo problemas de suma, 2 significa problemas de resta, 3 significa problemas de multiplicación, 4 significa sólo problemas de división y 5 significa problemas entremezclados al azar, de todos los tipos anteriores.

5.45 Escriba la función **distance** que calcule la distancia entre dos puntos (x1, y1) y (x2, y2). Todos los valores de los números y valores de regreso deberán de ser del tipo **float**.

5.46 ¿Qué es lo que hace el siguiente programa?

```
main()
{
   int c;

   if ((c = getchar()) != EOF) {
      main();
      printf("%c", c);
   }

   return 0;
}
```

5.47 ¿Qué es lo que hace el siguiente programa?

```
int mystery(int, int);

main()
{
   int x, y;

   printf("Enter two integers: ");
   scanf("%d%d", &x, &y);
   printf("The result is %d\n", mystery(x, y));
   return 0;
}
```



```
/* Parameter b must be a positive
   integer to prevent infinite recursion */
int mystery(int a, int b)
{
   if (b == 1)
      return a;
   else
      return a + mystery(a, b - 1);
}
```

5.48 Una vez que haya determinado lo que hace el programa del ejercicio 5,47, modifíquelo para que funcione correctamente, después de eliminar la restricción del segundo argumento que no sea negativo.

5.49 Escriba un programa que pruebe todas las funciones matemáticas posibles de biblioteca en la figura 5.2 como pueda. Ejecute cada una de estas funciones haciendo que su programa imprima tablas de valores de regreso para una diversidad de valores de argumentos.

5.50 Encuentre el error en cada uno de los segmentos de programa, y explique cómo corregirlo:

a)
```
float cube(float);   /* function prototype */
...
cube(float number)   /* function definition */
{
   return number * number * number;
}
```

b)
```
register auto int x = 7;
```

c)
```
int randomNumber = srand();
```

d)
```
float y = 123.45678;
int x;

x = y;
printf("%f\n", (float) x);
```

e)
```
double square(double number)
{
   double number;

   return number * number;
}
```

f)
```
int sum(int n)
{
   if (n == 0)
      return 0;
   else
      return n + sum(n);
}
```

5.51 Modifique el programa de craps de la figura 5.10 para permitir apuestas. Empaquete como una función aquella parte del programa que ejecuta un juego de craps. Inicialice la variable **bankBalance** a 1000 dólares. Solicite al jugador que introduzca una apuesta. Utilice un ciclo **while** para verificar que **wager** es menor o igual a **bankBalance** y de lo contrario indíquele al usuario de que vuelva a entrar **wager** hasta que se introduzca un **wager** válido. Una vez introducido un **wager** válido, ejecute un juego de craps. Si el jugador gana, aumente **bankBalance** por la cantidad **wager** e imprima el nuevo **bankBalance**. Si el jugador pierde, reduzca **bankBalance** por la cantidad **wager**, imprima el nuevo **bankBalance**, verifique si **bankBalance** se ha convertido en cero, y si es así, imprima el mensaje **"Sorry. You busted!"** Conforme progrese el juego, imprima varios mensajes para crear algo de "plática" como es **"Oh, you're going for broke, huh?"**, o bien **"Aw cmon, take a chance!"** o bien **"You're up big. Now's the time to cash in your chips!"**

6

Arreglos

Objetivos

- Presentar el concepto de la estructura de arreglos de datos.
- Comprender el uso de los arreglos para almacenar, ordenar y buscar listas y tablas de valores.
- Comprender como declarar un arreglo, como incializarlo y como referirse a los elementos individuales de un arreglo.
- Ser capaz de pasar arreglos a funciones.
- Comprender técnicas básicas de clasificación.
- Ser capaz de declarar y de manipular arreglos de varios subíndices.

Con sollozos y lágrimas escogió
Aquellos de mayor tamaño ...
Lewis Carroll

Intente lo último, y nunca acepte la duda;
Nada es tan duro, y la búsqueda lo encontrará.
Robert Herrick

Ahora ve, escríbelo delante de ellos en una mesa,
y anótalo en un libro.
Isaías 30:8

'Está bajo llave en mi memoria'
Y tu mismo conservarás la llave.
William Shakespeare.

Sinopsis

6.1 Introducción

Este capítulo sirve como una introducción al tema tan importante de las estructuras de datos. Los *arreglos* son estructuras de datos consistentes en elementos de datos relacionados del mismo tipo. En el capítulo 10, analizamos la idea o noción en C de **`struct`** (estructuras) —una estructura de datos consistente de elementos relacionados de datos, de tipos posiblemente distintos. Los arreglos y las estructuras son entidades "estáticas", debido a que se conservan del mismo tamaño a todo lo largo de la ejecución del programa (pudieran, naturalmente, ser de la clase de almacenamiento automático y, por lo tanto, creadas y destruidas cada vez que los bloques en los cuales se definen entren o salgan). En el capítulo 12, introducirémos estructuras dinámicas de datos como son listas, colas de espera, pilas y árboles, que pueden crecer o encogerse conforme los programas se ejecutan.

6.2 Arreglos

Un arreglo es un grupo de posiciones en memoria relacionadas entre sí, por el hecho de que todas tienen el mismo nombre y son del mismo tipo. Para referirse a una posición en particular o elemento dentro del arreglo, especificamos el nombre del arreglo y el número de posición del elemento particular dentro del mismo.

En la figura 6.1 se muestra un arreglo de enteros llamado **c**. Este arreglo contiene doce *elementos*. Cualquiera de estos elementos puede ser referenciado dándole el nombre del arreglo seguido por el número de posición de dicho elemento en particular en paréntesis cuadrados o corchetes (**`[]`**). El primer elemento de cualquier arreglo es el *elemento cero*. Entonces, el primer elemento de un arreglo **c** se conoce como **`c[0]`**, el segundo como **`c[1]`**, el séptimo como **`c[6]`** y en general, el elemento de orden *i* del arreglo **c** se conoce como **`c[i-1]`**. Los nombres de los arreglos siguen las mismas reglas convencionales que los demás nombres de variables.

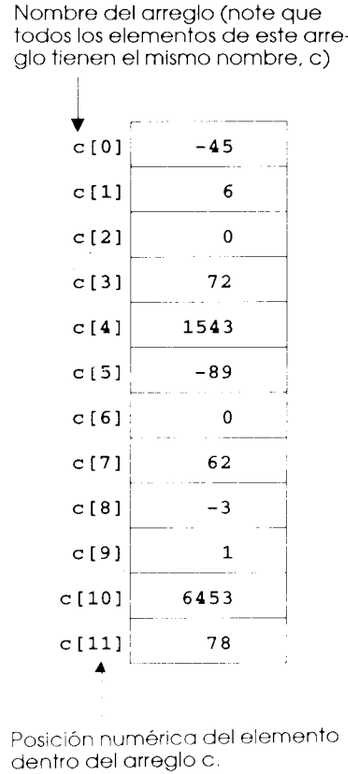


Fig. 6.1 Un arreglo de 12 elementos.

El número de posición que aparece dentro de los corchetes se conoce más formalmente como *subíndice*. Un subíndice debe ser un entero o una expresión entera. Si un programa utiliza una expresión como subíndice, entonces la expresión se evalúa para determinar el subíndice. Por ejemplo, si **`a = 5`** y **`b = 6`**, entonces el enunciado

```
c[a + b] += 2;
```

añade 2 al elemento del arreglo **`c[11]`**. Note que un nombre de arreglo con subíndice es un lvalue que puede ser utilizado en el lado izquierdo de una asignación.

Examinemos más de cerca el arreglo **c** de la figura 6.1. El *nombre* del arreglo es **c**. Sus doce elementos se conocen como **`c[0]`**, **`c[1]`**, **`c[2]`**, ..., **`c[11]`**. El *valor* de **`c[0]`** es **`-45`**, el valor de **`c[1]`** es **`6`**, el valor de **`c[2]`** es **`0`**, el valor de **`c[7]`** es **`62`** y el valor de **`c[11]`** es **`78`**. Para imprimir la suma de los valores contenidos en los primeros tres elementos del arreglo **c**, escribiríamos

```
printf("%d", c[0] + c[1] + c[2]);
```

Para dividir el valor del séptimo elemento del arreglo **c** entre **2** y asignar el resultado a la variable **x**, escribiríamos

```
x = c[6] / 2;
```

Error común de programación 6.1

Es importante notar la diferencia entre el séptimo elemento del arreglo "y el elemento siete del arreglo". Dado que los subíndices de los arreglos empiezan en 0, "el séptimo elemento del arreglo" tiene un subíndice de 6, en tanto que "el elemento 7 del arreglo" tiene un subíndice de siete y de hecho es el octavo elemento del arreglo. Este es una fuente de error por "diferencia de uno".

Los corchetes utilizados para cerrar el subíndice de un arreglo, son de hecho en C considerados como un operador. Tienen el mismo nivel de precedencia que los paréntesis. La gráfica de la figura 6.2 muestra la precedencia y asociatividad de los operadores introducidos hasta este momento dentro del texto. Se muestran de arriba a abajo en orden decreciente de precedencia.

6.3 Cómo declarar los arreglos

Los arreglos ocupan espacio en memoria. El programador especifica el tipo de cada elemento y el número de elementos requerido por cada arreglo, de tal forma que la computadora pueda reservar la cantidad apropiada de memoria. Para indicarle a la computadora que reserve 12 elementos para el arreglo entero *c*, la declaración

```
int c[12];
```

es utilizada. La memoria puede ser reservada para varios arreglos dentro de una sola declaración. Para reservar 100 elementos para el arreglo entero **b** y 27 elementos para el arreglo entero **x**, se utiliza la siguiente declaración:

Operador	Asociatividad	Tipo
`()  []`	de izquierda a derecha	máximo
`++  -  !  (tipo)`	de derecha a izquierda	unario
`*  /  %`	de izquierda a derecha	multiplicativo
`+`	de izquierda a derecha	aditivo
`<  <=  >  >=`	de izquierda a derecha	relacional
`==  !=`	de izquierda a derecha	igualdad
`&&`	de izquierda a derecha	lógica y
`||`	de izquierda a derecha	lógica o
`?:`	de derecha a izquierda	condicional
`=  +=  -=  *=  /=  %=`	de derecha a izquierda	asignación
`,`	de izquierda a derecha	coma

Fig. 6.2 Precedencia de operadores.


```
int b[100], x[27];
```

Los arreglos pueden ser declarados para que contengan otros tipos de datos. Por ejemplo, un arreglo del tipo **char** puede ser utilizado para almacenar una cadena de caracteres. Las cadenas de caracteres y su similitud con arreglos son analizadas en el capítulo 8. La relación entre apuntadores y arreglos es analizada en el capítulo 7.

6.4 Ejemplos utilizando arreglos

El programa de la figura 6.3 utiliza la estructura de repetición **for** para inicializar los elementos de un arreglo entero de diez elementos **n** a ceros, e imprime el arreglo en formato tabular.

Advierta que decidimos no colocar una línea en blanco entre el primer enunciado **printf** y la estructura **for** en la figura. 6.3, dado que están relacionados de forma íntima. En este caso, el enunciado **printf** muestra los encabezados de columnas de las dos columnas impresas en la estructura **for**. Los programadores omiten la línea en blanco entre la estructura **for** y algún enunciado **printf** que esté muy relacionado.

```
/* initializing an array */
#include <stdio.h>

main()
{
   int n[10], i;

   for (i = 0; i <= 9; i++)        /* initialize array */
      n[i] = 0;

   printf("%s%13s\n", "Element", "Value");
   for(i = 0; i <= 9; i++)         /* print array */
      printf("%7d%13d\n", i, n[i]);

   return 0;
}
```

```
Element         Value
      0             0
      1             0
      2             0
      3             0
      4             0
      5             0
      6             0
      7             0
      8             0
      9             0
```

Fig. 6.3 Cómo inicializar los elementos de un arreglo a ceros.

Los elementos de un arreglo también pueden ser inicializados en la declaración del arreglo mismo, haciendo seguir a la declaración con el signo igual y una lista separada por comas (encerrada entre llaves) de *inicializadores*. El programa de la figura 6.4 inicializa un arreglo entero con diez valores e imprime el arreglo en formato tabular.

Si dentro del arreglo existe un número menor de inicializadores que de elementos, los elementos restantes son inicializados a cero de forma automática. Por ejemplo, los elementos del arreglo **n** de la figura 6.3 podrían haber sido inicializados a cero con la declaración

```
int n[10] = {0};
```

con lo que de manera explícita se inicializa el primer elemento a cero y, por lo demás, los nueve elementos restantes se inicializan automáticamente a cero, porque existen menos inicializadores que elementos del arreglo. Es importante recordar que los arreglos no son de forma automática inicializados a cero. El programador debe por lo menos inicializar el primer elemento a cero, para que los demás queden automáticamente inicializados a cero. Este método de inicializar los elementos del arreglo a `0` se ejecuta en tiempo de compilación. El método utilizado en la figura 6.3, puede ser llevado a cabo en forma repetida, conforme se ejecuta el programa.

```
/* Initializing an array with a declaration */
#include <stdio.h>

main()
{
   int n[10] = {32, 27, 64, 18, 95, 14, 90, 70, 60, 37};
   int i;

   printf("%s%13s\n", "Element", "Value");

   for(i = 0; i <= 9; i++)
      printf("%7d%13d\n", i, n[i]);

   return 0;
}
```

```
Element         Value
      0            32
      1            27
      2            64
      3            18
      4            95
      5            14
      6            90
      7            70
      8            60
      9            37
```

Fig. 6.4 Cómo inicializar los elementos de un arreglo mediante una declaración.

Error común de programación 6.2

Olvidar inicializar los elementos de un arreglo cuyos elementos deban de estar inicializados.

La siguiente declaración de arreglo

```
int n[5] = {32, 27, 64, 18, 95, 14};
```

generaría un error de sintaxis, porque en el arreglo existen 6 inicializadores y únicamente 5 elementos.

Error común de programación 6.3

El proporcionar más inicializadores en una lista inicializadora de arreglo que elementos existan dentro del mismo constituye un error de sintaxis.

Si de una declaración con una lista inicializadora se omite el tamaño del arreglo, el número de elementos en el arreglo será el número de elementos incluidos en la lista inicializadora. Por ejemplo,

```
int n[] = {1, 2, 3, 4, 5};
```

crearía un arreglo de cinco elementos.

El programa de la figura 6.5 incializa los elementos de un arreglo **s** de diez elementos a los valores `2, 4, 6, ... 20`, e imprime el arreglo en formato tabular. Los valores se generan multiplicando el contador del ciclo por `2` y añadiendo `2`.

La directiva de preprocesador **`#define`** se introduce en este programa. La línea

```
#define SIZE 10
```

define una *constante simbólica* **`SIZE`** cuyo valor es `10`. Una constante simbólica es un identificador que se *remplaza con texto* de remplazo en el preprocesador C, antes de que el programa sea compilado. Cuando el programa es preprocesado, todas las instancias de la constante simbólica **`SIZE`** serán remplazadas por el texto de remplazo `10`. El uso de constantes simbólicas para especificar tamaños de arreglo hacen que los programas sean más *dimensionables*. En la figura 6.5, el primer ciclo **`for`** podría llenar un arreglo de 1000 elementos sólo modificando el valor de **`SIZE`** de la directiva **`#define`** de `10` a `1000`. Si no se hubiera utilizado la constante simbólica **`SIZE`**, tendríamos que modificar el programa en tres lugares distintos para dimensionar el programa, a fin de que el arreglo pudiera manejar 1000 elementos. Conforme los programas se hacen más grandes, esta técnica se hace más útil para la escritura de programas claros.

Error común de programación 6.4

Terminar una directiva de preprocesador **`#define`**, *o bien* **`#include`** *con un punto y coma. Recuerde que las directivas de preprocesador no son enunciados C.*

En la directiva de preprocesador **`#define`** anterior, si ésta estuviera terminada con un punto y coma, todas las ocurrencias de aparición de la constante simbólica **`SIZE`** en el programa serían remplazadas por el texto `10;` por dicho preprocesador. Esto pudiera llevar a errores de sintaxis en tiempo de compilación, o errores lógicos en tiempo de ejecución. Recuerde que el preprocesador no es C, es sólo un manipulador de texto.

Error común de programación 6.5

Asignar un valor a una constante simbólica en un enunciado ejecutable es un error de sintaxis. Una constante simbólica no es una variable. El compilador no reserva espacio para ella como hace con las variables que contienen valores durante la ejecución.

```
/* Initialize the elements of array s to
   the even integers from 2 to 20 */
#include <stdio.h>
#define SIZE 10

main()
{
   int s[SIZE], j;

   for (j = 0; j <= SIZE - 1; j++)   /* set the values */
      s[j] = 2 + 2 * j;

   printf("%s%13s\n", "Element", "Value");

   for (j = 0; j <= SIZE - 1; j++)   /* print the values */
      printf("%7d%13d\n", j, s[j]);

   return 0;
}
```

```
Element         Value
      0             2
      1             4
      2             6
      3             8
      4            10
      5            12
      6            14
      7            16
      8            18
      9            20
```

Fig. 6.5 Cómo generar los valores a colocarse en los elementos de un arreglo.

Observación de ingeniería de software 6.1

Definir el tamaño de cada arreglo como una constante simbólica hace a los programas más dimensionables.

Práctica sana de programación 6.1

Utilice sólo letras mayúsculas para los nombres de constantes simbólicas. Esto hace que estas constantes resalten en un programa y le recuerden al programador que las constantes simbólicas no son variables.

El programa de la figura 6.6 suma los valores contenidos en un arreglo entero **a** de doce elementos. El enunciado en el cuerpo del ciclo **for** se ocupa de la totalización.

Nuestro siguiente ejemplo utiliza arreglos para resumir los resultados de datos recopilados en una investigación. Considere el enunciado del problema.

A cuarenta alumnos se les preguntó el nivel de calidad de los alimentos de la cafetería para alumnos en una escala de 1 a 10 (1 significa terrible y 10 significa excelente). Coloque las cuarenta respuestas en un arreglo entero y resuma los resultados de la encuesta.


```
/* Compute the sum of the elements of the array */
#include <stdio.h>
#define SIZE 12

main()
{
   int a[SIZE] = {1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45},
       i, total = 0;

   for (i = 0; i <= SIZE - 1; i++)
      total += a[i];

   printf("Total of array element values is %d\n", total);
   return 0;
}
```

```
Total of array element values is 383
```

Fig. 6.6 Cómo calcular la suma de los elementos de un arreglo.

Esta es una aplicación típica de arreglo (vea la figura 6.7). Deseamos resumir el número de respuestas de cada tipo (es decir, de 1 hasta 10). El arreglo **responses** es un arreglo de 40 elementos correspondiente a las respuestas de los alumnos. Utilizamos un arreglo de once elementos, **frequency** para contar el número de ocurrencias de cada respuesta. Ignoramos el primer elemento **frequency[0]**, porque es más lógico tener un incremento de respuesta 1 **frequency[1]** que **frequency[0]**. Esto nos permite utilizar cada respuesta directa como subíndice en el arreglo **frequency**.

Práctica sana de programación 6.2

Busque claridad en el programa. Algunas veces será preferible sacrificar una utilización más eficiente de memoria o de tiempo de procesador en aras de escribir programas más claros.

Sugerencia de rendimiento 6.1

Algunas veces las consideraciones de rendimiento tienen mayor importancia que las consideraciones de claridad.

El primer ciclo **for** toma las respuestas del arreglo **response** una por una e incrementa uno de los diez contadores (**frequency[1]** hasta **frequency[10]**) en el arreglo **frequency**. El enunciado clave del ciclo es

```
++frequency[responses[answer]];
```

Este enunciado incrementa el contador **frequency** apropiado, dependiendo del valor de **responses[answer]**. Por ejemplo, cuando la variable del contador **answer** es 0, **responses[answer]** es 1 y, por lo tanto, **++frequency[responses[answer]];** se interpreta en realidad como

```
++frequency[1];
```

```
/* Student poll program */
#include <stdio.h>
#define RESPONSE_SIZE 40
#define FREQUENCY_SIZE 11

main()
{
   int answer, rating;
   int responses[RESPONSE_SIZE] = {1, 2, 6, 4, 8, 5, 9, 7, 8,
       10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
       5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10};
   int frequency[FREQUENCY_SIZE] = {0};

   for(answer = 0; answer <= RESPONSE_SIZE - 1; answer++)
      ++frequency[responses[answer]];

   printf("%s%17s\n", "Rating", "Frequency");

   for(rating = 1; rating <= FREQUENCY_SIZE - 1; rating++)
      printf("%6d%17d\n", rating, frequency[rating]);

   return 0;
}
```

```
Rating          Frequency
     1                  2
     2                  2
     3                  2
     4                  2
     5                  5
     6                 11
     7                  5
     8                  7
     9                  1
    10                  3
```

Fig. 6.7 Un programa sencillo de análisis de una encuesta de alumnos.

lo que incrementa el elemento uno del arreglo. Cuando **answer** es **1**, **responses[answer]** es **2** y, por lo que, **++frequency [responses[answer]];** se interpreta como

```
++frequency[2];
```

lo que incrementa el elemento dos del arreglo. Cuando **answer** es **2**, **responses[answer]** es **6**, por lo que **++frequency [responses[answer]];** se interpreta como

```
++frequency[6];
```

lo que incrementa al elemento seis del arreglo, y así en lo sucesivo. Note que, independiente del número de respuestas procesadas en la encuesta, sólo se requiere un arreglo de once elementos

(ignorando al elemento cero) para resumir los resultados. Si los datos tuvieran valores inválidos como el 13, el programa intentaría añadir **1** a **frequency[13]**. Esto quedaría fuera de los límites del arreglo. *C no tiene verificación de límites de arreglo, para impedir que la computadora se refiera a un elemento no existente*. Entonces, un programa en ejecución podría salirse sin aviso del final de un arreglo. El programador debe asegurarse que todas las referencias al arreglo se conservan dentro de los límites del mismo.

Error común de programación 6.6

Referirse a un elemento exterior a los límites del arreglo.

Práctica sana de programación 6.3

Al ciclar a través de un arreglo, el subíndice de un arreglo no debe de pasar nunca por debajo de 0 y siempre tiene que ser menor que el número total de elementos del arreglo (tamaño -1). Asegúrese que la condición de terminación del ciclo impide el acceso a elementos fuera de este rango.

Práctica sana de programación 6.4

Mencione el subíndice alto del arreglo en una estructura `for`, a fin de ayudar a eliminar los errores por diferencia de uno.

Práctica sana de programación 6.5

Los programas deberían verificar la corrección de todos los valores de entrada, para impedir que información errónea afecte los cálculos del programa.

Sugerencia de rendimiento 6.2

Los efectos (normalmente serios) de referirse a elementos fuera de los límites del arreglo, son dependientes del sistema.

Nuestro siguiente ejemplo (figura 6.8) lee números de un arreglo y representa la información en forma de una gráfica de barras o histograma —cada número es impreso, y a continuación al lado del número se imprime una barra, formada por muchos asteriscos. De hecho el que dibuja las barras es el ciclo anidado **for**. Note el uso de **printf ("\n")** para dar por terminada la barra del histograma.

En el capítulo 5 indicamos que mostraríamos un método más elegante de escribir el programa de tirada de dados de la figura 5.10. El problema era tirar un dado de seis caras 6000 veces, para probar si el generador de números aleatorios de verdad produce números al azar. Una versión en arreglo de este programa, se muestra en la figura 6.9

Hasta este punto, sólo nos hemos ocupado de arreglos enteros. Sin embargo, los arreglos son capaces de contener datos de cualquier tipo. Ahora analizarémos el almacenamiento de cadenas en arreglos de caracteres. Hasta ahora, la única capacidad que tenemos de procesamiento de cadenas es la salida de una cadena mediante **printf**. En C, una cadena como "**hello**", de hecho es un arreglo de caracteres individuales.

Los arreglos de caracteres tienen varias características únicas. Un arreglo de caracteres puede ser inicializado utilizando una literal de cadena. Por ejemplo, la declaración

```
char string1[] = "first";
```

inicializa los elementos de la cadena **string1** a los caracteres individuales de la literal de cadena **"first"**. El tamaño del arreglo **string1** en la declaración anterior queda determinada por el compilador, basado en la longitud de la cadena. Es importante hacer notar que la cadena **"first"**

```
/* Histogram printing program */
#include <stdio.h>
#define SIZE 10

main()
{
   int n[SIZE] = {19, 3, 15, 7, 11, 9, 13, 5, 17, 1};
   int i, j;

   printf("%s%13s%17s\n", "Element", "Value", "Histogram");

   for (i = 0; i <= SIZE - 1; i++) {
      printf("%7d%13d        ", i, n[i]);

      for(j = 1; j <= n[i]; j++)    /* print one bar */
         printf("%c", '*');

      printf("\n");
   }

   return 0;
}
```

```
Element        Value        Histogram
      0           19        *******************
      1            3        ***
      2           15        ***************
      3            7        *******
      4           11        ***********
      5            9        *********
      6           13        *************
      7            5        *****
      8           17        *****************
      9            1        *
```

Fig. 6.8 Un programa que imprime histogramas.

contiene cinco caracteres, *más* un carácter especial de terminación de cadena, conocido como *carácter nulo*. Entonces, el arreglo **string1**, de hecho contiene seis elementos. La representación de la constante de caracteres del carácter nulo es **'\0'**. En C todas las cadenas terminan con este carácter. Un arreglo de caracteres, representando una cadena, debería declararse siempre lo suficiente grande para contener el número de caracteres de la cadena, incluyendo el carácter nulo de terminación.

Los arreglos de caracteres también pueden ser inicializados con constantes individuales de caracteres en una lista de incialización. La declaración anterior es equivalente a

```
char string1[] = {'f', 'i', 'r', 's', 't', '\0'};
```

Dado que la cadena de hecho es un arreglo de caracteres, podemos tener acceso directo a los caracteres individuales de una cadena, utilizando la notación de subíndices de arreglos. Por ejemplo, **string1[0]**, es el carácter **'f'** y **string1[3]** es el carácter **'s'**.


```
/* Roll a six-sided die 6000 times */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 7

main()
{
   int face, roll, frequency[SIZE] = {0};

   srand(time(NULL));

   for (roll = 1; roll <= 6000; roll++) {
      face = rand() % 6 + 1;
      ++frequency[face];        /* replaces 20-line switch */
   }                            /* of Fig. 5.10 */

   printf("%s%17s\n", "Face", "Frequency");

   for (face = 1; face <= SIZE - 1; face++)
      printf("%4d%17d\n", face, frequency[face]);

   return 0;
}
```

```
Face        Frequency
   1             1037
   2              987
   3             1013
   4             1028
   5              952
   6              983
```

Fig. 6.9 Programa de tirada de dados utilizando arreglos en vez de **switch**.

También podemos introducir desde el teclado directamente una cadena en un arreglo de caracteres, utilizando **scanf** y la especificación de conversión **%s**. Por ejemplo, la declaración.

```
char string2[20];
```

crea un arreglo de caracteres capaz de almacenar una cadena de 19 caracteres y un carácter nulo de terminación. El enunciado

```
scanf("%s", string2);
```

lee una cadena del teclado y la coloca en **string2**. Note que el nombre del arreglo se pasa a **scanf** sin colocar el **&** precedente, que en otras variables se utiliza. El **&** es utilizado por lo regular para darle a **scanf** una localización de variable en memoria, a fin de que se pueda almacenar un valor ahí. En la sección 6.5 analizarémos cómo pasar arreglos a funciones. Veremos que el nombre de un arreglo es la dirección del inicio del arreglo y, por lo tanto, **&** no es necesario.

Es la responsabilidad del programador asegurarse que el arreglo al cual se lee la cadena, es capaz de contener cualquier cadena que el usuario escriba en el teclado. La función **scanf** lee caracteres del teclado hasta que se encuentra con el primer carácter de espacio en blanco sin importarle qué tan grande es el arreglo. Por lo tanto, **scanf** podría escribir más allá del final del arreglo.

Error común de programación 6.7

No proporcionar, en un programa, a scanf un arreglo de caracteres lo suficiente grande para almacenar una cadena escrita en el teclado, puede dar como resultado una pérdida de datos, así como otros errores en tiempo de ejecución.

Un arreglo de caracteres que represente a una cadena puede ser sacado utilizando **printf** y el especificador de conversión **%s**. El arreglo **string2** se imprime utilizando el enunciado

```
printf("%s\n", string2);
```

Note que a **printf**, al igual que a **scanf**, no le importa qué tan grande es el arreglo de caracteres. Los caracteres de la cadena serán impresos, hasta que un carácter de terminación nulo sea encontrado.

En la figura 6.10 se demuestra la inicialización de un arreglo de caracteres con una literal de caracteres, la lectura de una cadena a un arreglo de caracteres, la impresión de un arreglo de caracteres como una cadena, y el acceso a caracteres individuales de una cadena.

```
/* Treating character arrays as strings */
#include <stdio.h>

main()
{
   char string1[20], string2[] = "string literal";
   int i;

   printf("Enter a string: ");
   scanf("%s", string1);
   printf("string1 is: %s\nstring2: is %s\n"
          "string1 with spaces between characters is:\n",
          string1, string2);

   for (i = 0; string1[i] != '\0'; i++)
      printf("%c ", string1[i]);

   printf("\n");
   return 0;
}
```

```
Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

Fig. 6.10 Cómo tratar arreglos de caracteres como cadenas.

En la figura 6.10 se utiliza la estructura **for** para ciclar a través del arreglo **string1** e imprimir los caracteres individuales separados por espacios mediante la especificación de conversión **%c**. La condición en la estructura **for, string1[i] != '\0'**, es verdadera en tanto, dentro de la cadena, el carácter nulo de terminación no sea encontrado.

En el capítulo 5, se analizó el especificador de clase de almacenamiento **static**. Una variable local **static**, en una definición de función, existe durante la duración del programa, pero resulta sólo visible en el cuerpo de la función. Podemos aplicar **static** a una declaración de arreglo local, de tal forma que el arreglo no sea creado e inicializado cada vez que se llame a la función, y el arreglo no se destruya cada vez que en el programa la función se termine. Esto reduce el tiempo de ejecución del programa, en particular en el caso de programas con funciones de llamado frecuente y que contengan arreglos extensos.

Sugerencia de rendimiento 6.3

En funciones que contengan arreglos automáticos donde la función entra y sale de alcance con frecuencia, haga **static** *dicho arreglo, de tal forma que no sea necesario crearlo cada vez que la función sea llamada.*

Los arreglos que se declaran **static** son inicializados de forma automática una vez en tiempo de compilación. Si un arreglo **static** no se inicializa de manera explícita por el programador, éste quedará inicializado a cero por el compilador.

En la figura 6.11 se demuestra la función **staticArrayInit**, con un arreglo local declarado **static** y una función **automaticArrayInit** con un arreglo local automático. La función **staticArrayInit** es llamada dos veces. El arreglo local **static** en la función es inicializado a cero por el compilador. La función imprime el arreglo, añade 5 a cada uno de los elementos, y lo vuelve a imprimir. La segunda vez que la función es llamada, el arreglo **static** contiene los valores almacenados durante la primera llamada de la función. La función **automaticArrayInit** también es llamada dos veces. Se inicializan los elementos del arreglo local automático en la función con los valores 1, 2 y 3. La función imprime el arreglo, añade 5 a cada elemento y lo vuelve a imprimir. La segunda vez que se llama a la función, los elementos del arreglo están otra vez inicializados a 1, 2, 3, porque el arreglo tiene una duración de almacenamiento automático.

Error común de programación 6.8

Suponer que los elementos de un arreglo local, que está declarado como **static**, *están inicializados a cero, cada vez que la función sea llamada donde se declara el arreglo.*

6.5 Cómo pasar arreglos a funciones

Para pasar cualquier argumento de arreglo a una función, especifique el nombre del arreglo, sin corchete alguno. Por ejemplo, si el arreglo **hourlyTemperatures** ha sido declarado como

```
in hourlyTemperatures[24];
```

el enunciado de llamada a la función

```
modifyArray(hourlyTemperatures, 24);
```

pasa el arreglo **hourlyTemperatures** y su tamaño, a la función **modifyArray**. Al pasar un arreglo a una función, el tamaño del arreglo a menudo se pasa a la función, de tal forma que pueda procesar el número específico de elementos incluidos en dicho arreglo.

```
/* Static arrays are initialized to zero */
#include <stdio.h>

void staticArrayInit(void);
void automaticArrayInit(void);

main()
{
   printf("First call to each function:\n");
   staticArrayInit();
   automaticArrayInit();
   printf("\n\nSecond call to each function:\n");
   staticArrayInit();
   automaticArrayInit();
   return 0;
}

/* function to demonstrate a static local array */
void staticArrayInit(void)
{
   static int a[3];
   int i;

   printf("\nValues on entering staticArrayInit:\n");

   for (i = 0; i <= 2; i++)
      printf("array1[%d] = %d  ", i, a[i]);

   printf("\nValues on exiting staticArrayInit:\n");

   for (i = 0; i <= 2; i++)
      printf("array1[%d] = %d  ", i, a[i] += 5);
}

/* function to demonstrate an automatic local array */
void automaticArrayInit(void)
{
   int a[3] = {1, 2, 3};
   int i;

   printf("\n\nValues on entering automaticArrayInit:\n");

   for (i = 0; i <= 2; i++)
      printf("array1[%d] = %d  ", i, a[i]);

   printf("\nValues on exiting automaticArrayInit:\n");

   for (i = 0; i <= 2; i++)
      printf("array1[%d] = %d  ", i, a[i] += 5);
}
```

Fig. 6.11 Los arreglos estáticos son de forma automática inicializados a cero, si no han sido de manera explícita inicializados por el programador (parte 1 de 2).


```
First call to each function:

Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array[2] = 5

Values on entering automaticArrayInit:
array1[0] = 1  array1[1] = 2  array1[2] = 3
Values on exiting automaticArrayInit:
array1[0] = 6  array1[1] = 7  array[2] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10  array1[1] = 10  array1[2] = 10

Values on entering automaticArrayInit:
array1[0] = 1  array1[1] = 2  array1[2] = 3
Values on exiting automaticArrayInit:
array1[0] = 6  array1[1] = 7  array1[2] = 8
```

Fig. 6.11 Los arreglos estáticos son de forma automática inicializados a cero, si no han sido explícitamente inicializados por el programador (parte 2 de 2).

C pasa de forma automática los arreglos a las funciones utilizando simulación de llamadas por referencia —las funciones llamadas pueden modificar los valores de los elementos en los arreglos originales de los llamadores. ¡El nombre del arreglo de hecho es la dirección del primer elemento de dicho arreglo! Dado que ha sido pasada la dirección inicial del arreglo, la función llamada sabe precisamente dónde está el arreglo almacenado. Por lo tanto, cuando en su cuerpo de función, la función llamada modifica los elementos del arreglo, está modificando los elementos reales del arreglo, en sus localizaciones de memoria originales.

En la figura 6.12 puede demostrarse que un nombre de un arreglo es en realidad la dirección del primer elemento de dicho arreglo, imprimiendo **`array`** y **`&array[0]`**, utilizando la especificación de conversión %p —especificación especial de conversión para impresión de direcciones. La especificación de conversión %p da por lo regular salida a direcciones como de números hexadecimales. Los números hexadecimales (base 16) se forman de los dígitos 0 a 9 y de las letras A hasta la F. Se utilizan a menudo como una escritura abreviada de valores de enteros grandes. En el Apéndice E: en los Sistemas numéricos se proporciona un análisis en profundidad de las relaciones entre enteros binarios (de base 2), octales (en base 8), decimales (en base diez; enteros estándar), y hexadecimales. La salida muestra que tanto **`array`** como **`&array[0]`** tienen el mismo valor, es decir **`FFF0`.** La salida de este programa es dependiente del sistema, pero las direcciones siempre resultarán idénticas.

Sugerencia de rendimiento 6.4

Tiene sentido pasar arreglos simulando llamadas por referencia por razones de rendimiento. Si los arreglos fueran pasados en llamadas por valor, se pasaría una copia de cada uno de los elementos. En el caso de arreglos grandes pasados con frecuencia, esto tomaría mucho tiempo y consumiría gran cantidad de espacio de almacenamiento para las copias de los arreglos.

```
/* The name of an array is the same as &array[0] */
#include <stdio.h>

main()
{
   char array[5];

   printf("    array = %p\n&array[0] = %p\n",
          array, &array[0]);
   return 0;
}
```

```
    array = FFF0
&array[0] = FFF0
```

Fig. 6.12 El nombre de un arreglo es el mismo que la dirección del primer elemento de dicho arreglo.

Observación de ingeniería de software 6.2

Es posible pasar un arreglo por valor utilizando una triquiñuela sencilla que explicaremos en el capítulo 10.

A pesar de que se pasan arreglos completos simulando llamadas por referencia, los elementos individuales del arreglo se pasan en llamadas por valor, de la misma forma que se pasan las variables simples. Estas simples y únicas porciones de datos son conocidas como *escalares o cantidades escalares.* Para pasar un elemento de un arreglo a una función, utilice como argumento en la llamada a la función el nombre con subíndice del elemento del arreglo. En el capítulo 7, mostramos cómo simular llamadas por referencia para escalares (es decir, para elementos del arreglo y variables individuales).

Para que una función reciba un arreglo a través de una llamada de función, la lista de parámetros de la función debe especificar que se va a recibir un arreglo. Por ejemplo, el encabezado de función para la función **modifyArray** pudiera ser escrito como

```
void modifyArray(int b[], int size)
```

indicando que **modifyArray** espera recibir un arreglo de enteros en el parámetro **b** y un cierto número de elementos de arreglo en el parámetro **size.** No es necesario indicar el tamaño del arreglo dentro de los corchetes del arreglo. Si se incluye, el compilador lo ignorará. Dado que los arreglos son pasados de forma automática por simulación de llamadas por referencia, cuando la función llamada utiliza el nombre del arreglo **b**, de hecho se estará refiriendo al arreglo real en el llamador (arreglo **hourlyTemperatures** de la llamada anterior). En el capítulo 7, presentamos otras notaciones para indicar que un arreglo está siendo recibido por una función. Como veremos, en el lenguaje C estas formas de notación se basan en las relaciones íntimas existentes entre arreglos y apuntadores.

Advierta la apariencia extraña del prototipo de función correspondiente a **modifyArray**

```
void modifyArray(int [], int);
```

Este prototipo podría haber sido escrito

```
void modifyArray(int anyArrayName[], int anyVariableName)
```


pero como aprendimos en el capítulo 5, el compilador de C ignora los nombres de variables dentro de los prototipos.

Práctica sana de programación 6.6

Algunos programadores incluyen nombres de variables en los prototipos de función, para que los programas resulten más claros. El compilador ignorará estos nombres.

Recuerde, el prototipo le indica al compilador el número de argumentos y los tipos en que aparecerán cada uno de los argumentos (en el orden en que se ejecutarán estos).

El programa de la figura 6.13 demuestra la diferencia entre pasar un arreglo completo y pasar un elemento de un arreglo. El programa primero imprime los cinco elementos de un arreglo entero **a**. A continuación, **a** y su tamaño son pasados a la función **modifyArray,** donde cada uno de los elementos de **a** es multiplicado por 2. A continuación **a** se vuelve a imprimir en **main**. Como se muestra en la salida, los elementos de **a** en realidad han sido modificados por **modifyArray**. Ahora el programa imprime el valor de **a[3]** y lo pasa a la función **modifyElement**. La función **modifyElement** multiplica su argumento por 2 e imprime el nuevo valor. Note que cuando **a[3]** es vuelto a imprimir en **main**, no ha sido modificado porque los elementos individuales del arreglo han sido pasados en llamada por valor.

```
/* Passing arrays and individual array elements to functions */
#include <stdio.h>
#define SIZE 5

void modifyArray(int [], int);  /* appears strange */
void modifyElement(int);

main()
{
   int a[SIZE] = {0, 1, 2, 3, 4};
   int i;

   printf("Effects of passing entire array call "
          "by reference:\n\nThe values of the "
          "original array are:\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%3d", a[i]);

   printf("\n");
   modifyArray(a, SIZE);  /* array a passed call by reference */
   printf("The values of the modified array are:\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%3d", a[i]);

   printf("\n\n\nEffects of passing array element call "
          "by value:\n\nThe value of a[3] is %d\n", a[3]);
   modifyElement(a[3]);
   printf("The value of a[3] is %d\n", a[3]);
   return 0;
}
```

Fig. 6.13 Cómo pasar arreglos y elementos individuales de arreglo a funciones (parte 1 de 2).

```
void modifyArray(int b[], int size)
{
   int j;

   for (j = 0; j <= size - 1; j++)
      b[j] *= 2;
}

void modifyElement(int e)
{
   printf("Value in modifyElement is %d\n", e *= 2);
}
```

```
Effects of passing entire array call by reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8

Effects of passing array element call by avalue:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
```

Fig. 6.13 Cómo pasar arreglos y elementos individuales de arreglo a funciones (parte 2 de 2).

Pudieran existir en sus programas situaciones en las cuales a una función no se le deberá permitir que modifique los elementos de un arreglo. Dado que los arreglos son siempre pasados simulando llamadas por referencia, resulta difícil de controlar la modificación de los valores de un arreglo. C proporciona un calificador especial de tipo **const**, para evitar la modificación de los valores de arreglo en una función. Cuando un parámetro de arreglo es antecedido por el calificador **const**, los elementos del arreglo se convierten en constantes en el cuerpo de la función, y cualquier intento para modificar un elemento en el cuerpo de la función da como resultado un error en tiempo de compilación. Esto le permite al programador corregir un programa, de forma tal que no intente modificar los elementos del arreglo. A pesar de que el calificador **const** está bien definido en el estándar ANSI, los diferentes sistemas C tienen variantes en sus capacidades para ponerlo en práctica.

En la figura 6.14 se demuestra el calificador **const**. La función **tryToModifyArray** está definida con el parámetro **const int b[]** que especifica que el arreglo **b** es constante, y no puede ser modificado. La salida muestra los mensajes de error producidos por el compilador Borland C++. Cada uno de los tres intentos de la función para modificar elementos del arreglo resultan en el error de compilación **"Cannot modify a const object"**. El calificador **const** será vuelto a analizar en el capítulo 7.

Observación de ingeniería de software 6.3

El calificador de tipo **const** *puede ser aplicado a un parámetro de arreglo en una definición de función, para impedir que en el cuerpo de la función el arreglo original sea modificado. Este es otro ejemplo del principio de mínimo privilegio. Las funciones no deben tener, ni se les dará, capacidad de modificar un arreglo, a menos de que sea en lo absoluto necesario.*


```
/* Demonstrating the const type qualifier */
#include <stdio.h>

void tryToModifyArray(const int []);

main()
{
   int a[] = {10, 20, 30};

   tryToModifyArray(a);
   printf("%d %d %d\n", a[0], a[1], a[2]);
   return 0;
}

void tryToModifyArray(const int b[])
{
   b[0] /= 2;     /* error */
   b[1] /= 2;     /* error */
   b[2] /= 2;     /* error */
}
```

```
Compiling FIG6_14.C:
Error FIG6_14-C 16: Cannot modify a const object
Error FIG6_14.C 17: Cannot modify a const object
Error FIG6_14.C 18: Cannot modify a const object
Warning FIG6_14.C 19: Parameter 'b' is never used
```

Fig. 6.14 Demostración del calificador de tipo **const**.

6.6 Cómo ordenar arreglos

La *ordenación* de datos (es decir, colocar los datos en un orden particular, como orden ascendente o descendente) es una de las aplicaciones más importantes de la computación. Un banco clasifica todos los cheques por número de cuenta, de tal forma que al final de cada mes pueda preparar estados bancarios individuales. Para facilitar la búsqueda de números telefónicos, las empresas telefónicas clasifican sus listas de cuentas por apellido y dentro de ello, por nombre. Virtualmente todas las organizaciones deben clasificar algún dato y en muchos casos, cantidades masivas de información. La ordenación de datos es un problema intrigante que ha atraído alguno de los esfuerzos más intensos de investigación en el campo de la ciencia de la computación. En este capítulo analizamos lo que quizás es el esquema más simple de ordenación conocido. En los ejercicios y en el capítulo 12, investigamos esquemas más complejos, que consiguen rendimientos muy superiores.

Sugerencia de rendimiento 6.5

A menudo, los algoritmos más sencillos tienen rendimientos pobres. Su virtud estriba en que son fáciles de escribir, de probar y de depurar. Sin embargo, para obtener rendimiento máximo con frecuencia se requiere de algoritmos más complejos.

El programa en la figura 6.15 ordena en orden ascendente los valores de los elementos de un arreglo **a** de diez elementos. La técnica que utilizamos se conoce como *ordenación tipo burbuja*

u ordenación por hundimiento, porque los valores más pequeños de forma gradual "flotan" hacia la parte superior del arreglo, como suben las burbujas de aire en el agua, en tanto que los valores más grandes se hunden hacia el fondo del arreglo. La técnica consiste en llevar a cabo varias pasadas a través del arreglo. En cada pasada, se comparan pares sucesivos de elementos. Si un par está en orden creciente (o son idénticos sus valores), dejamos los valores tal y como están. Si un par aparece en orden decreciente, sus valores en el arreglo se intercambian de lugar.

```
/* This program sorts an array's values into
   ascending order */
#include <stdio.h>
#define SIZE 10

main()
{
   int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
   int i, pass, hold;

   printf("Data items in original order\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%4d", a[i]);

   for (pass = 1; pass <= SIZE - 1; pass++)  /* passes */

      for (i = 0; i <= SIZE - 2; i++)       /* one pass */

         if (a[i] > a[i + 1]) {       /* one comparison */
            hold = a[i];                 /* one swap */
            a[i] = a[i + 1];
            a[i + 1] = hold;
         }

   printf("\nData items in ascending order\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%4d", a[i]);

   printf("\n");

   return 0;
}
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Fig. 6.15 Cómo ordenar un arreglo utilizando la ordenación tipo burbuja.

Primero el programa compara **a[0]** con **a[1]**, después **a[1]** con **a[2]**, a continuación **a[2]** con **a[3]**, y así en lo sucesivo, hasta completar la pasada, comparando **a[8]** con **a[9]**. Note que aunque existen 10 elementos, sólo se ejecutan nueve comparaciones. En razón de la forma en que se llevan a cabo las sucesivas comparaciones, en una pasada un valor grande puede pasar hacia abajo en el arreglo muchas posiciones, pero un valor pequeño pudiera moverse hacia arriba en una posición. En la primera pasada, el valor más grande está garantizado que se hundirá hasta el elemento más inferior del arreglo, **a[9]**. En la segunda pasada el segundo valor más grande está garantizado que se hundirá a **a[8]**. En la novena pasada, el valor noveno en tamaño se hundirá a **a[1]**. Esto dejará el tamaño más pequeño en **a[0]**, por lo que sólo se necesitan de nueve pasadas del arreglo para ordenarlo, aun cuando existan diez elementos.

La ordenación se ejecuta mediante el ciclo **for** anidado. Si un intercambio es necesario, es ejecutado mediante las tres asignaciones

```
hold = a[i];
a[i] = a[i + 1];
a[i + 1] = hold;
```

donde la variable adicional **hold** almacena en forma temporal uno de los dos valores en intercambio. El intercambio no se puede ejecutar con sólo dos asignaciones

```
a[i] = a[i + 1];
a[i + 1] = a[i];
```

Si, por ejemplo, **a[i]** es 7 y **a[i + 1]** es 5, después de la primera asignación ambos valores serán 5 y se perderá el valor 7. De ahí la necesidad de la variable adicional **hold**.

La virtud más importante de la ordenación tipo burbuja, es que es fácil de programar. Sin embargo, la ordenación tipo burbujas es de lenta ejecución. Esto se hace aparente al ordenar arreglos extensos. En los ejercicios, desarrollaremos versiones más eficientes de la ordenación tipo burbuja. Se han desarrollado ordenaciones mucho más eficientes que la clasificación tipo burbuja. Más adelante en el texto investigaremos unas cuantas de éstas. Cursos más avanzados investigan con mayor profundidad la ordenación y búsqueda.

6.7 Estudio de caso: cómo calcular el promedio, la mediana y el modo utilizando arreglos

Consideremos ahora un ejemplo mayor. Las computadoras se utilizan por lo común para compilar y analizar los resultados de encuestas y de encuestas de opinión. El programa de la figura 6.16 utiliza el arreglo **response** inicializado con 99 respuestas (representadas por la constante simbólica **SIZE**) de una encuesta. Cada una de las respuestas es un número del 1 al 9. El programa calcula el promedio, la mediana y el modo de los 99 valores.

El promedio es el promedio aritmético de los 99 valores. La función **mean** calcula el promedio totalizando los 99 elementos y dividiendo el resultado entre 99.

La mediana es el "valor medio". La función **median** determina la mediana llamando a la función **bubbleSort** para ordenar el arreglo de respuestas en orden ascendente, y seleccionando el elemento medio, **answer [SIZE / 2]**, del arreglo ya ordenado. Note que cuando existe un número par de elementos, la mediana deberá ser calculada como el promedio de los dos elementos de en medio. Esta versión de la función **median** no proporciona esta capacidad. La función **printArray** es llamada para la salida del arreglo **response**.

El modo es el valor que ocurre con mayor frecuencia entre las 99 respuestas. La función **mode** determina el modo, contando el número de respuestas de cada tipo, y a continuación escogiendo

UNIVERSIDAD DE LA REPUBLICA
FACULTAD DE INGENIERIA

el valor que tenga el conteo mayor. Esta versión de la función **mode** no maneja un empate (vea el ejercicio 6.14). La función **mode** también produce un histograma, para auxiliar en la determinación gráfica del modo. En la figura 6.17 se da una ejecución de muestra de este programa. Este ejemplo incluye la mayor parte de las manipulaciones normales requeridas con frecuencia en los problemas de arreglos, incluyendo el pasar arreglos a funciones.

```
/* This program introduces the topic of survey data analysis.
   It computes the mean, median, and  mode of the data */
#include <stdio.h>
#define SIZE 99

void mean(int []);
void median(int []);
void mode(int [], int []);
void bubbleSort(int[]);
void printArray(int[]);

main()
{
   int frequency[10] = {0},
       response[SIZE] = {6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
                         7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
                         6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
                         7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
                         6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
                         7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
                         5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
                         7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
                         7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
                         4, 5, 6, 1, 6, 5, 7, 8, 7};
   mean(response);
   median(response);
   mode(frequency, response);
   return 0;
}

void mean(int answer[])
{
   int j, total = 0;

   printf("%s\n%s\n%s\n", "*********", "  Mean", "*********");

   for (j = 0; j <= SIZE - 1; j++)
      total += answer[j];

   printf("The mean is the average value of the data\n"
          "items. The mean is equal to the total of\n"
          "all the data items divided by the number\n"
          "of data items (%d). The mean value for\n"
          "this run is: %d / %d = %.4f\n\n",
          SIZE, total, SIZE, (float) total / SIZE);
}
```

Fig. 6.16 Programa de análisis de datos de encuesta (parte 1 de 3).


```
void median(int answer[])
{
   printf("\n%s\n%s\n%s\n%s",
          "*********", " Median", "*********",
          "The unsorted array of responses is");

   printArray(answer);
   bubbleSort(answer);
   printf("\n\nThe sorted array is");
   printArray(answer);
   printf("\n\nThe median is element %d of\n"
          "the sorted %d element array.\n"
          "For this run the median is %d\n\n",
          SIZE / 2, SIZE, answer[SIZE / 2]);
}

void mode(int freq[], int answer[])
{
   int rating, j, h, largest = 0, modeValue = 0;

   printf("\n%s\n%s\n%s\n",
          "*********", "  Mode", "*********");

   for (rating = 1; rating <= 9; rating++)
      freq[rating] = 0;

   for (j = 0; j <= SIZE - 1; j++)
      ++freq[answer[j]];

   printf("%s%11s%19s\n\n%54s\n%54s\n\n",
          "Response", "Frequency", "Histogram",
          "1    1    2    2", "5    0    5    0    5");

   for (rating = 1; rating <= 9; rating++) {
      printf("%8d%11d          ", rating, freq[rating]);

      if (freq[rating] > largest) {
         largest = freq[rating];
         modeValue = rating;
      }

      for (h = 1; h <= freq[rating]; h++)
         printf("*");

      printf("\n");
   }

   printf("The mode is the most frequent value.\n"
          "For this run the mode is %d which occurred"
          " %d times.\n", modeValue, largest);
}
```

Fig. 6.16 Programa de análisis de datos de encuesta (parte 2 de 3).

```
void bubbleSort(int a[])
{
   int pass, j, hold;

   for (pass = 1; pass <= SIZE - 1; pass++)

      for (j = 0; j <= SIZE - 2; j++)

         if (a[j] > a[j+1]) {
            hold = a[j];
            a[j] = a[j+1];
            a[j+1] = hold;
         }
}

void printArray(int a[])
{
   int j;

   for (j = 0; j <= SIZE - 1; j++) {

      if (j % 20 == 0)
         printf("\n");

      printf("%2d", a[j]);
   }
}
```

Fig. 6.16 Programa de análisis de datos de encuesta (parte 3 de 3).

6.8 Búsqueda en arreglos

A menudo, un programador estará trabajando con grandes cantidades de datos almacenados en arreglos. Pudiera resultar necesario determinar si un arreglo contiene un valor que coincide con algún *valor clave o buscado*. El proceso de encontrar en un arreglo un elemento en particular, se llama *búsqueda*. En esta sección se analizan dos técnicas de búsqueda —la técnica simple de *búsqueda lineal* y la técnica más eficiente de *búsqueda binaria*. Los ejercicios 6.34 y 6.35 al final de este capítulo le piden que diseñe versiones recursivas, tanto de la búsqueda lineal como de la búsqueda binaria.

La búsqueda lineal (figura 6.18) compara cada uno de los elementos del arreglo con la *el valor buscado*. Dado que el arreglo no está en ningún orden en particular, existe la misma probabilidad de que el valor se encuentre, ya sea en el primer elemento como en el último. Por lo tanto, en promedio, el programa tendrá que comparar el valor buscado con la mitad de los elementos del arreglo.

El método de búsqueda lineal funciona bien para arreglos pequeños o para arreglos no ordenados. Sin embargo, para la búsqueda de arreglos extensos, el sistema lineal es ineficiente. Si el arreglo está ordenado, se puede utilizar la técnica de alta velocidad de búsqueda binaria.

El algoritmo de búsqueda binaria, después de cada una de las comparaciones, elimina la mitad de los elementos en el arreglo bajo búsqueda. El algoritmo localiza el elemento medio del arreglo y lo compara con el valor buscado. Si son iguales, la clave de búsqueda ha sido encontrada y se


```
********
  Mean
********
The mean is the average value of the data
items. The mean is equal to the total of
all the data items divided by the number
of data items (99). The mean value for
this run is: 681 / 99 = 6.8788

********
  Median
********
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

The median is element 49 of
the sorted 99 element array.
For this run the median is 7

********
  Mode
********
Response   Frequency              Histogram

                                    1  1  2  2
                                 5  0  5  0  5

       1          1        *
       2          3        ***
       3          4        ****
       4          5        *****
       5          8        ********
       6          9        *********
       7         23        ***********************
       8         27        ***************************
       9         19        *******************
The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.
```

Fig. 6.17 Ejecución de muestra del programa de análisis de datos de encuesta.

```
/* Linear search of an array */
#include <stdio.h>
#define SIZE 100

int linearSearch(int [], int, int);

main()
{
   int a[SIZE], x, searchKey, element;

   for (x = 0; x <= SIZE - 1; x++)  /* create some data */
      a[x] = 2 * x;

   printf("Enter integer search key:\n");
   scanf("%d", &searchKey);
   element = linearSearch(a, searchKey, SIZE);

   if (element != -1)
      printf("Found value in element %d\n", element);
   else
      printf("Value not found\n");

   return 0;
}

int linearSearch(int array[], int key, int size)
{
   int n;

   for (n = 0; n <= size - 1; ++n)
      if (array[n] == key)
         return n;

   return -1;
}
```

```
Enter integer search key:
36
Found value in element 18
```

```
Enter integer search key:
37
Value not found
```

Fig. 6.18 Búsqueda lineal en un arreglo.

regresa el subíndice del arreglo correspondiente a dicho elemento. Si no son iguales, el problema se reduce a buscar en una mitad del arreglo. Si el valor buscado es menor que el elemento medio del arreglo, se seguirá buscando en la primera parte del arreglo, de lo contrario se buscará en la

segunda parte. Si el valor buscado no se encuentra en el subarreglo especificado (es la porción del arreglo original), el algoritmo se repite en una cuarta parte del arreglo original. La búsqueda continúa, hasta que el valor buscado es igual al elemento del medio del subarreglo, o hasta que el subarreglo ha quedado reducido a un elemento diferente a el valor buscado (es decir, el valor buscado no ha sido encontrado).

En el peor escenario, utilizando la búsqueda binaria, la búsqueda de un arreglo de 1024 elementos sólo tomará 10 comparaciones. La división repetida de 1024 entre 2, da los valores 512, 256, 128, 64, 32, 16, 8, 4, 2 y 1. El número 1024 (2^{10}) se divide entre 2 sólo diez veces para obtener el valor de 1. En el algoritmo de búsqueda binaria la división por 2 es equivalente a una comparación. Un arreglo de 1048576 (2^{20}) elementos toma un máximo de 20 comparaciones, para encontrar el valor buscado. Un arreglo de mil millones de elementos, toma un máximo de 30 comparaciones para encontrar el valor buscado. Esto es un incremento tremendo en rendimiento en comparación con la búsqueda lineal, que en promedio requería la comparación de el valor buscado en la mitad de los elementos del arreglo. ¡En el caso de un arreglo de mil millones de elementos, es una diferencia entre un promedio de 500 millones de comparaciones y un máximo de 30! Las comparaciones máximas para cualquier arreglo pueden ser determinadas encontrando la primera potencia de 2 mayor que el número de elementos en el arreglo.

En la figura 6.19 se presenta la versión iterativa de la función **binarySearch**. La función recibe cuatro argumentos —un arreglo entero **b**, un entero **searchKey**, el subíndice de arreglo **low** y el subíndice de arreglo **high**. Si el valor buscado no coincide con el elemento medio de un subarreglo, el subíndice **low** o el subíndice **high** es modificado, de tal forma que pueda buscarse en un subarreglo más pequeño. Si el valor buscado es menor que el elemento medio, el subíndice **high** se define como **middle - 1**, y la búsqueda se continúa sobre los elementos desde **low** hasta **middle - 1**. Si el valor buscado es mayor que el elemento medio, el subíndice **low** se define como **middle + 1** y la búsqueda se continúa sobre los elementos desde **middle + 1** hasta **high**. El programa utiliza un arreglo de 15 elementos. La primera potencia de 2 mayor que el número de elementos en este arreglo es 16 (2^4), por lo que se requerirán un máximo de 4 comparaciones para encontrar el valor buscado. El programa utiliza la función **printHeader** para sacar los subíndices del arreglo y la función **printRow** para sacar cada subarreglo durante el proceso de búsqueda binaria. El elemento medio de cada subarreglo queda marcado con un asterisco (*) para indicar cuál es el elemento con el cual el valor buscado se compara.

6.9 Arreglos con múltiples subíndices

En C los arreglos pueden tener múltiples subíndices. Una utilización común de los arreglos con múltiples subíndices es la representación de *tablas* de valores, consistiendo de información arreglada en *renglones* y *columnas*. Para identificar un elemento particular de la tabla, deberemos especificar dos subíndices; el primero (por regla convencional) identifica el renglón del elemento, y el segundo (también por regla convencional) identifica la columna del elemento. Tablas o arreglos que requieren dos subíndices para identificar un elemento en particular se conocen como *arreglos de doble subíndice*. Note que los arreglos de múltiples subíndices pueden tener más de dos subíndices. El estándar ANSI indica que un sistema ANSI C debe soportar por lo menos 12 subíndices de arreglo.

En la figura 6.20 se ilustra un arreglo de doble subíndice, **a**. El arreglo contiene tres renglones y cuatro columnas, por lo que se dice que se trata de un arreglo de 3 por 4. En general, un arreglo con *m* renglones y *n* columnas se llama un *arreglo de m por n*.

```
/* Binary search of an array */

#include <stdio.h>
#define SIZE 15

int binarySearch(int [], int, int, int);
void printHeader(void);
void printRow(int [], int, int, int);


main()
{
   int a[SIZE], i, key, result;

   for (i = 0; i <= SIZE - 1; i++)
      a[i] = 2 * i;

   printf("Enter a number between 0 and 28: ");
   scanf("%d", &key);

   printHeader();
   result = binarySearch(a, key, 0, SIZE - 1);

   if (result != -1)
      printf("\n%d found in array element %d\n", key, result);
   else
      printf("\n%d not found\n", key);

   return 0;
}


int binarySearch(int b[], int searchKey, int low, int high)
{
   int middle;

   while (low <= high) {
      middle = (low + high) / 2;

      printRow(b, low, middle, high);

      if (searchKey == b[middle])
         return middle;
      else if (searchKey < b[middle])
         high = middle - 1;
      else
         low = middle + 1;
   }

   return -1;    /* searchKey not found */
}
```

Fig. 6.19 Búsqueda binaria de un arreglo ordenado (parte 1 de 3).


```
/* Print a header for the output */
void printHeader(void)
{
   int i;

   printf("\nSubscripts:\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%3d ", i);

   printf("\n");

   for (i = 1; i <= 4 * SIZE; i++)
      printf("-");

   printf("\n");
}

/* Print one row of output showing the current
   part of the array being processed. */
void printRow(int b[], int low, int mid, int high)
{
   int i;

   for (i = 0; i <= SIZE - 1; i++)
      if (i < low || i > high)
         printf("    ");
      else if (i == mid)
         printf("%3d*", b[i]);  /* mark middle value */
      else
         printf("%3d ", b[i]);

   printf("\n");
}
```

Fig. 6.19 Búsqueda binaria de un arreglo ordenado (parte 2 de 3).

Cada uno de los elementos en el arreglo **a** está identificado en la figura 6.20 por un nombre de elemento de la forma **a[i][j]**; **a** es el nombre del arreglo, e **i** y **j** son los subíndices que identifican en forma única a cada elemento dentro de **a**. Note que los nombres de los elementos en el primer renglón, todos tienen un primer subíndice de **0**; los nombres de los elementos en la cuarta columna, todos tienen un segundo subíndice de **3**.

Error común de programación 6.9

Referenciar un elemento de arreglo de doble subíndice como **a[x][y]** *en vez de* **a[x, y]**.

Un arreglo de múltiple subíndice puede ser inicializado en su declaración en forma similar a un arreglo de un subíndice. Por ejemplo, un arreglo de doble subíndice **b[2][2]** podría ser declarado e inicializado con

```
int b[2][2] = {{1, 2}, {3, 4}};
```

```
Enter a number between 0 and 28: 25

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
 ---------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
                              16  18  20  22* 24  26  28
                                          24  26* 28
                                          24*

25 not found
```

```
Enter a number between 0 and 28: 8

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
 ---------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12
                  8  10* 12
                  8*

8 found in array element 4
```

```
Enter a number between 0 and 28: 6

Suscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
 ---------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12

6 found in array element 3
```

Fig. 6.19 Búsqueda binaria de un arreglo ordenado (parte 3 de 3).

Los valores se agrupan por renglones entre llaves. Por lo tanto, `1` y `2` inicializan `b[0][0]` y `b[0][1]`, `3` y `4` inicializan `b[1][0]` y `b[1][1]`. Si para un renglón dado no se proporcionan suficientes inicializadores, los elementos restantes de dicho renglón se inicializarán a `0`. Por lo tanto, la declaración

```
int b[2][2] = {{1}, {3, 4}};
```

inicializaría `b[0][0]` a `1`, `b[0][1]` a `0`, `b[1][0]` a `3` y `b[1][1]` a `4`.

En la figura 6.21 se demuestra la inicialización de arreglos de doble subíndice en declaraciones. El programa declara tres arreglos de dos renglones y tres columnas (de seis elementos cada

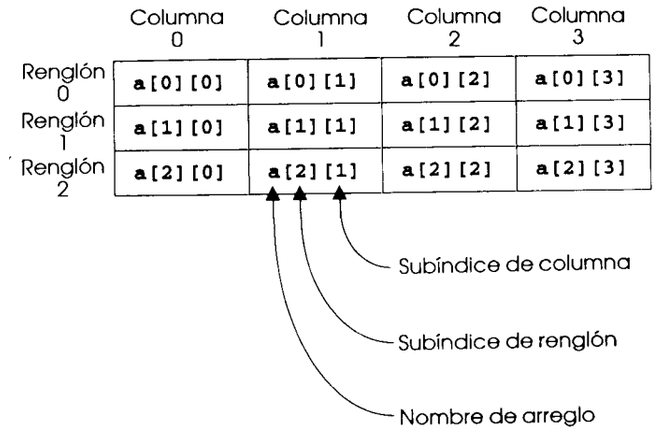


Fig. 6.20 Un arreglo de doble subíndice con tres renglones y cuatro columnas.

uno de ellos). La declaración de `array1` proporciona seis inicializadores en dos sublistas. La primera sublista incializa el primer renglón del arreglo a los valores 1, 2 y 3; y la segunda sublista inicializa el segundo renglón del arreglo a los valores 4, 5, y 6. Si de la lista de inicialización `array1` se retiran los parentesis ( ) de alrededor de cada sublista, el compilador de forma automática inicializa los elementos del primer renglón seguido por los elementos del segundo. La declaración de `array1` proporciona cinco inicializadores. Los inicializadores son asignados al primer renglón y a continuación al segundo. Cualquier elemento que no tenga un inicializador explícito es inicializado en forma automática a cero, por lo que `array2[1][2]` es inicializado a 0. La declaración de `array3` proporciona tres inicializadores en dos sublistas. La sublista para el primer renglón inicializa de manera explícita los dos primeros elementos del primer renglón a 1 y a 2. El tercer elemento automáticamente queda inicializado a cero. La sublista para el segundo renglón inicializa explícitamente el primer elemento a 4. Los dos elementos siguientes automáticamente se inicializan a cero.

El programa llama a la función `printArray` para que se impriman los elementos de cada arreglo. Note que la definición de la función especifica el parámetro de arreglo como `int a[][3]`. Cuando recibimos un arreglo de un subíndice como argumento a una función, los corchetes del arreglo están vacíos en la lista de parámetros de la función. El primer subíndice de un arreglo de múltiples subíndices tampoco se requiere, pero todos los demás subíndices son requeridos. El compilador utiliza estos subíndices para determinar las localizaciones en memoria de los elementos en arreglos de múltiples subíndices. Todos los elementos del arreglo son almacenados en memoria de forma consecutiva, independiente del número de subíndices. En un arreglo de doble subíndice, el primer renglón se almacena en memoria y a continuación se almacena el segundo.

Proporcionar los valores de subíndices en una declaración de parámetros le permite al compilador indicarle a la función cómo localizar un elemento del arreglo. En un arreglo de doble subíndice, cada renglón es básicamente un arreglo de un solo subíndice. Para localizar un elemento en un renglón en particular, el compilador debe saber con exactitud cuántos elementos existen en cada renglón, de tal forma que pueda saltar la cantidad apropiada de posiciones de memoria para llegar al arreglo. Por lo tanto, al tener acceso a `a[1][2]` en nuestro ejemplo, el compilador sabe

```
/* Initializing multidimensional arrays */
#include <stdio.h>

void printArray(int [][3]);

main()
{
   int array1[2][3] = { {1, 2, 3}, {4, 5, 6} },
       array2[2][3] = { 1, 2, 3, 4, 5 },
       array3[2][3] = { {1, 2}, {4} };

   printf("Values in array1 by row are:\n");
   printArray(array1);

   printf("Values in array2 by row are:\n");
   printArray(array2);

   printf("Values in array3 by row are:\n");
   printArray(array3);

   return 0;
}

void printArray(int a[][3])
{
   int i, j;

   for (i = 0; i <= 1; i++) {

      for (j = 0; j <= 2; j++)
         printf("%d ", a[i][j]);;

      printf("\n");
   }
}
```

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Fig. 6.21 Cómo inicializar arreglos multidimensionales.

que debe de saltarse los tres elementos del primer renglón en la memoria para obtener el segundo renglón (renglón 1). A continuación, el compilador llega al tercer elemento de dicho renglón (elemento 2).

Muchas manipulaciones comunes con arreglos utilizan estructuras de repetición **for**. Por ejemplo, la siguiente estructura define todos los elementos en el tercer renglón del arreglo **a** en la figura 6.20 a cero:

```
for (column = 0; column  3; column++)
   a[2][column] = 0;
```

Especificamos el *tercer* renglón, por lo tanto, sabemos que el primer subíndice será siempre **2** (**0** es el primer renglón y **1** el segundo). El ciclo **for** varía sólo en el segundo subíndice (es decir, el subíndice de columnas). La estructura **for** anterior es equivalente a los enunciados de asignación siguientes:

```
a[2][0] = 0;
a[2][1] = 0;
a[2][2] = 0;
a[2][3] = 0;
```

En el arreglo **a**, la siguiente estructura **for** anidada determina el total de todos los elementos.

```
total = 0;

for (row = 0; row <= 2; row++)

for (column = 0; column <= 3; column++)
   total += a [row][column];
```

La estructura **for** totaliza los elementos del arreglo, renglón por renglón. La estructura externa **for** empieza estableciendo **row** (es decir, el subíndice de renglón) a **0**, de tal forma que los elementos del primer renglón puedan ser totalizados por la estructura interna **for**. La estructura externa **for** incrementa **row** a **1**, de tal forma que los elementos del segundo renglón sean totalizados. Por último, la estructura externa **for** incrementa **row** a **2**, para que se totalicen los elementos del tercer renglón. El resultado se imprime cuando la estructura **for** anidada termina.

El programa de la figura 6.22 ejecuta otras manipulaciones comunes de arreglos en un arreglo de 3 por 4 **studentGrades**, utilizando estructuras **for**. Cada renglón del arreglo representa un alumno y cada columna representa una calificación, en uno de los cuatro exámenes que los alumnos pasaron durante el semestre. Las manipulaciones de arreglo se ejecutan mediante cuatro funciones. La función **minimum** determina el grado más bajo de cualquier alumno para el semestre. La función **maximum** determina la calificación más alta de cualquier alumno en el semestre. La función **average** determina el promedio para el semestre de un alumno en particular. La función **printArray** extrae la salida del arreglo de doble subíndice en un formato tabular nítido.

Las funciones **minimum**, **maximum** y **printArray** cada una de ellas recibe tres argumentos —el arreglo **studentGrades** (denominadas **grades** en cada función), el número de alumnos (renglones del arreglo), y el número de exámenes (columnas del arreglo). Cada función cicla a través del arreglo **grades**, utilizando estructuras **for** anidadas. La siguiente estructura **for** anidada corresponde a la función de definición **minimum**:

```
for (i = 0; i <= pupils - 1; i++)
   for (j = 0; j <= tests - 1; j++)
      if (grades[i][j] < lowGrade)
         lowGrade = grades [i][j];
```

```
/* Double-subscripted array example */
#include <stdio.h>
#define STUDENTS 3
#define EXAMS 4

int minimum(int [][EXAMS], int, int);
int maximum(int [][EXAMS], int, int);
float average(int [], int);
void printArray(int [][EXAMS], int, int);

main()
{
   int student,
       studentGrades[STUDENTS][EXAMS] = {{77, 68, 86, 73},
                                         {96, 87, 89, 78},
                                         {70, 90, 86, 81}};

   printf("The array is:\n");
   printArray(studentGrades, STUDENTS, EXAMS);
   printf("\n\nLowest grade: %d\nHighest grade: %d\n",
          minimum(studentGrades, STUDENTS, EXAMS),
          maximum(studentGrades, STUDENTS, EXAMS));

   for (student = 0; student <= STUDENTS - 1; student++)
      printf("The average grade for student %d is %.2f\n",
             student, average(studentGrades[student], EXAMS));

   return 0;
}

/* Find the minimum grade */
int minimum(int grades[][EXAMS], int pupils, int tests)
{
   int i, j, lowGrade = 100;

   for (i = 0; i <= pupils - 1; i++)
      for (j = 0; j <= tests - 1; j++)
         if (grades[i][j] < lowGrade)
            lowGrade = grades[i][j];

   return lowGrade;
}

/* Find the maximum grade */
int maximum(int grades[][EXAMS], int pupils, int tests)
{
   int i, j, highGrade = 0;

   for (i = 0; i <= pupils - 1; i++)
      for (j = 0; j <= tests - 1; j++)
         if (grades[i][j] > highGrade)
            highGrade = grades[i][j];

   return highGrade;
}
```

Fig. 6.22 Ejemplo del uso de arreglos con doble subíndice (parte 1 de 2).


```
/* Determine the average grade for a particular exam */
float average(int setOfGrades[], int tests)
{
   int i, total = 0;

   for (i = 0; i <= tests - 1; i++)
      total += setOfGrades[i];

   return (float) total / tests;
}

/* Print the array */
void printArray(int grades[][EXAMS], int pupils, int tests)
{
   int i, j;

   printf("                  [0]  [1]  [2]  [3]");

   for (i = 0; i <= pupils - 1; i++) {
      printf("\nstudentGrades[%d] ", i);

      for (j = 0; j <= tests - 1; j++)
         printf("%-5d", grades[i][j]);
   }
}
```

```
The array is:
                  [0]  [1]  [2]  [3]
studentGrades[0] 77   68   86   73
studentGrades[1] 96   87   89   78
studentGrades[2] 70   90   86   81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75
```

Fig. 6.22 Ejemplo del uso de arreglos con doble subíndice (parte 2 de 2).

La estructura externa **for** empieza definiendo a **i** (es decir, el subíndice de renglón) en **0**, de tal forma que los elementos del primer renglón puedan ser comparados a la variable **lowGrade** en el cuerpo de la estructura interna **for**. La estructura interna **for** cicla a través de las cuatro calificaciones de un renglón en particular y compara cada una de las calificaciones con **lowGrade**. Si una calificación es menor que **lowGrade**, **lowGrade** se define igual a tal calificación. A continuación la estructura externa **for** incrementa el subíndice de renglón a **1**. El elemento del segundo renglón se compara a la variable **lowGrade**. Después la estructura **for** externa incrementa el subíndice de renglón a **2**. Los elementos del tercer renglón son comparados con la variable **lowGrade**. Cuando la ejecución de la estructura anidada está completa, **lowGrade** contiene la calificación más pequeña del arreglo de doble índice. La función **maximum** funciona de manera similar a la función **minimum**.

La función **average** toma dos argumentos —un arreglo de un solo índice de los resultados de las pruebas para un alumno en particular, llamada **setOfGrades** y el número de resultados de pruebas del arreglo. Cuando se llama a **average**, el primer argumento que se pasa es **studentGrades[student]**. Esto hace que la dirección de un renglón del arreglo de doble subíndice sea pasado a **average**. El argumento **studentGrades[1]** es la dirección inicial del segundo renglón del arreglo. Recuerde que un arreglo de doble subíndice es básicamente un arreglo de arreglos de un índice, y que el nombre de un arreglo de un solo índice es la dirección del arreglo en memoria. La función **average** calcula la suma de los elementos del arreglo, divide el total por el número de resultados de pruebas, y regresa un resultado en punto flotante.

Resumen

- C almacena listas de valores en arreglos. Un arreglo es un grupo de posiciones relacionadas en memoria. Estas posiciones están relacionadas por el hecho de que todas tienen el mismo nombre y son del mismo tipo. Para referirse a una posición particular o algún elemento dentro del arreglo, especificamos el nombre del arreglo y el subíndice.
- Un subíndice puede ser un entero o una expresión de enteros. Si un programa utiliza como subíndice una expresión, entonces la expresión se evalúa para determinar el elemento particular del arreglo.
- Es importante notar la diferencia entre referirse al séptimo elemento del arreglo y referirse al elemento siete del arreglo. El séptimo elemento tiene un subíndice de **6**, en tanto que el elemento siete del arreglo tiene un subíndice de **7** (y de hecho dentro del arreglo es el octavo elemento). Esto es una fuente de errores "por diferencia de uno".
- Los arreglos ocupan espacio en memoria. Para reservar 100 elementos para el arreglo entero **b** y 27 elementos para el arreglo entero **x**, el programador escribe

```
int b[100], x[27];
```

- Un arreglo del tipo **char** puede ser utilizado para almacenar una cadena de caracteres.
- Los elementos de un arreglo pueden ser inicializados de tres formas distintas: por declaración, por asignación y por entrada.
- Si existen menos inicializadores que elementos en el arreglo, C inicializará de forma automática a cero los elementos restantes.
- C no evita la referenciación de elementos más allá de los límites de un arreglo.
- Un arreglo de caracteres puede ser inicializado utilizando una literal de cadena.
- Todas las cadenas en C terminan con un carácter nulo. La representación de la constante de carácter nulo, es **'\0'**.
- Los arreglos de caracteres pueden ser inicializados en la lista de inicialización con constantes de caracteres.
- Los caracteres individuales en una cadena almacenada en un arreglo pueden ser accesibles de forma directa utilizando la notación de subíndice de arreglos.
- Se puede introducir de manera directa una cadena en un arreglo de caracteres desde el teclado utilizando **scanf** y la especificación de conversión **%s**.

- Un arreglo de caracteres que represente una cadena puede ser extraído utilizando **printf** y el especificador de conversión **%s**.
- Aplique **static** a una declaración de arreglo local a fin de que el arreglo no tenga que ser creado cada vez que se llame a la función y cada vez que la función termine no sea destruido el arreglo.
- Los arreglos declarados **static** se inicializan de forma automática una vez en tiempo de compilación. Si el programador no inicializa explícitamente un arreglo **static**, será inicializado a cero por el compilador.
- Para pasar un arreglo a una función, se pasa el nombre del arreglo. Para pasar un elemento de un arreglo a una función, sólo pase el nombre del arreglo seguido por el subíndice (contenido entre corchetes) del elemento particular.
- C pasa arreglo a las funciones utilizando simulación de llamada por referencias —las funciones llamadas pueden modificar los valores de los elementos en los arreglos originales de los llamadores. ¡El nombre del arreglo es de hecho la dirección del primer elemento del arreglo! Dado que es pasada la dirección de inicio del arreglo, la función llamada sabe precisamente donde dicho arreglo está almacenado.
- Para recibir un argumento de arreglo, la lista de parámetros de la función debe especificar que un arreglo será recibido. En los corchetes del arreglo no es requerido el tamaño del mismo.
- La especificación de conversión **%p** por lo regular extrae direcciones en números hexadecimales.
- C proporciona un calificador especial de tipo **const** para impedir la modificación de los valores de arreglos en una función. Cuando un parámetro de arreglo está precedido de un calificador **const**, los elementos del arreglo se convierten en constantes dentro del cuerpo de la función, y cualquier intento para modificar un elemento del arreglo en el cuerpo de la función, dan como resultado un error en tiempo de compilación.
- Un arreglo puede ser ordenado utilizando la técnica de oredenación tipo burbuja. Se hacen varias pasadas del arreglo. En cada una de las pasadas, se comparan pares sucesivos de elementos. Si un par está en orden (o si sus valores son idénticos), se deja tal como está. Si el par está fuera de orden, los valores son intercambiados. Para pequeños arreglos, la ordenación tipo burbuja es aceptable, pero es ineficiente en caso de arreglos mayores, y en comparación con otros algoritmos de ordenación más avanzados.
- La búsqueda lineal compara cada elemento del arreglo con el valor buscado. Dado que el arreglo no está en ningún orden en particular, existe igual probabilidad de que el valor sea encontrado tanto en el primer elemento como en el último. En promedio, por lo tanto, el programa tendrá que comparar el valor buscado con la mitad de los elementos del arreglo. El método de búsqueda lineal funciona bien para arreglos pequeños o para arreglos no clasificados.
- El algoritmo de búsqueda binaria elimina la mitad de los elementos del arreglo bajo búsqueda después de cada comparación. El algoritmo localiza el elemento medio del arreglo y lo compara con el valor buscado. Si son iguales, se ha encontrado el valor buscado y el subíndice del arreglo de dicho elemento se regresa. Si no son iguales, el problema queda reducido a volver a buscar en una de las mitades del arreglo.
- En el peor escenario, la búsqueda mediante la búsqueda binaria de un arreglo de 1024 elementos tomará sólo 10 comparaciones. Un arreglo de 1048576(2^{20}) elementos toma un máximo de 20 comparaciones para encontrar el valor buscado. Un arreglo de mil millones de elementos toma un máximo de 30 comparaciones para encontrar el valor buscado.

- Los arreglos pueden ser utilizados para representar tablas de valores, consistiendo de información arreglada en renglones y columnas. Para identificar un elemento particular de una tabla, se especifican dos subíndices: el primero (por regla convencional) identifica el renglón en el cual se contiene el elemento, y el segundo (por regla convencional) identifica la columna en el cual el elemento está contenido. Las tablas de arreglo que requieran de dos subíndices para identificar un elemento en particular, se conocen como arreglos de doble subíndice.
- El estándar indica que un sistema ANSI C debe de soportar por lo menos 12 subíndices de arreglo.
- Un arreglo de múltiples subíndices puede ser inicializado en su declaración utilizando una lista inicializadora.
- Cuando recibimos un arreglo de un subíndice como argumento de una función, los corchetes del arreglo están vacíos en la lista de parámetros de la función. Tampoco se requiere el primer subíndice de un arreglo de múltiples subíndices, pero todos los subíndices subsecuentes son requeridos. El compilador utilizará estos subíndices para determinar las posiciones en memoria de los elementos en los arreglos de múltiples subíndices.
- Para pasar un renglón de un arreglo de doble subíndice a una función que recibe un arreglo de un subíndice, sólo pase el nombre del arreglo, seguido por el primer subíndice.

Terminología

a[i]
a[i][j]
arreglo
lista de inicialización de arreglo
gráfica de barras
verificación de límites
ordenación tipo burbuja
subíndice de columna
declarar un arreglo
directiva de preprocesador `# define`
doble precisión
arreglo de doble subíndice
elemento de un arreglo
expresión como un subíndice
histograma
inicializar un arreglo
búsqueda lineal
arreglo m por n
promedio
mediana
modo
arreglo de múltiples subíndices
nombre de un arreglo
carácter nulo `'\0'`
error por diferencia de uno
pasar por referencia
pasar arreglos a funciones
especificación de conversión %p
número de posición
texto de remplazo
subíndice de renglón
dimensionabilidad
escalar
cantidad escalar
valor buscado
búsqueda en un arreglo
arreglo de un subíndice
ordenación por hundimiento
ordenación
pasada de ordenación
ordenación de los elementos de un arreglo
corchetes
cadena
subíndice
análisis de datos de encuesta
constante simbólica
tabla de valores
formato tabular
área temporal para intercambio de valores
totalizar los elementos de un arreglo
arreglo de triple subíndice
valor de un elemento
salirse de un arreglo
elemento de orden cero

Errores comunes de programación

6.1 Es importante notar la diferencia entre "el séptimo elemento del arreglo" "y el elemento siete del arreglo". Dado que los subíndices de los arreglos empiezan en 0, "el séptimo elemento del arreglo" tiene un subíndice de 6, en tanto que el elemento 7 del arreglo "tiene un subíndice de siete" y de hecho es el octavo elemento del arreglo. Este es una fuente de un error por "diferencia de uno".
6.2 Olvidar inicializar los elementos de un arreglo cuyos elementos deban de estar inicializados.
6.3 El proporcionar más inicializadores en una lista inicializadora de arreglo que elementos existan dentro del mismo constituye un error de sintaxis.
6.4 Terminar una directiva de preprocesador `#define`, o bien `#include` con un punto y coma. Recuerde que las directivas de preprocesador no son enunciados C.
6.5 Asignar un valor a una constante simbólica en un enunciado ejecutable es un error de sintaxis. Una constante simbólica no es una variable. El compilador no reserva espacio para ella por como hace con las variables que contienen valores durante la ejecución.
6.6 Referirse a un elemento exterior a los límites del arreglo.
6.7 No proporcionar, en un programa, a `scanf` un arreglo de caracteres lo suficiente grande para almacenar una cadena escrita en el teclado, puede dar como resultado una pérdida de datos, así como otros errores en tiempo de ejecución.
6.8 Suponer que los elementos de un arreglo local, que está declarado como `static`, están inicializados a cero, cada vez que la función sea llamada donde se declara el arreglo.
6.9 Referenciar un elemento de arreglo de doble subíndice como `a[x][y]` en vez de `a[x, y]`.

Prácticas sanas de programación

6.1 Utilice sólo letras mayúsculas para los nombres de constantes simbólicas. Esto hace que estas constantes resalten en un programa y le recuerden al programador que las constantes simbólicas no son variables.
6.2 Busque claridad en el programa. Algunas veces será preferible sacrificar la utilización más eficiente de memoria o de tiempo de procesador en aras de escribir programas más claros.
6.3 Al ciclar a través de un arreglo, el subíndice de un arreglo no debe de pasar nunca por debajo de 0 y siempre tiene que ser menor que el número total de elementos del arreglo (tamaño -1). Asegúrese que la condición de terminación del ciclo impide el acceso a elementos fuera de este rango.
6.4 Mencione el subíndice alto del arreglo en una estructura `for`, a fin de ayudar a eliminar los errores por diferencia de uno.
6.5 Los programas deberían verificar la corrección de todos los valores de entrada, para impedir que información errónea afecte los cálculos del programa.
6.6 Algunos programadores incluyen nombres de variables en las funciones prototipo, para que los programas resulten más claros. El compilador ignorará estos nombres.

Sugerencias de rendimiento

6.1 Algunas veces las consideraciones de rendimiento tienen mucho mayor importancia que las consideraciones de claridad.
6.2 Los efectos (por lo regular serios) de referirse a elementos fuera de los límites del arreglo, son dependientes del sistema.
6.3 En funciones que contengan arreglos automáticos donde la función entra y sale de alcance con frecuencia, haga `static` dicho arreglo, de tal forma que no sea necesario crearlo cada vez que la función sea llamada.
6.4 Tiene sentido pasar arreglos simulando llamadas por referencia por razones de rendimiento. Si los arreglos fueran pasados en llamadas por valor, se pasaría una copia de cada uno de los elementos. En el caso de arreglos grandes pasados con frecuencia, esto tomaría mucho tiempo y consumiría gran cantidad de espacio de almacenamiento para las copias de los arreglos.

6.5 A menudo, los algoritmos más sencillos tienen rendimientos pobres. Su virtud estriba en que son fáciles de escribir, de probar y de depurar. Sin embargo, para obtener rendimiento máximo a menudo se requiere de algoritmos más complejos.

Observaciones de ingeniería de software

6.1 Definir el tamaño de cada arreglo como una constante simbólica hace a los programas más dimensionables.

6.2 Es posible pasar un arreglo por valor utilizando una triquiñuela sencilla que explicaremos en el capítulo 10.

6.3 El calificador de tipo `const` puede ser aplicado a un parámetro de arreglo en una definición de función, para impedir que en el cuerpo de la función el arreglo original sea modificado. Este es otro ejemplo del principio de mínimo privilegio. Las funciones no deben tener, ni se les dará, capacidad de modificar un arreglo, a menos de que sea en lo absoluto necesario.

Ejercicios de autoevaluación

6.1 Llene cada uno de los siguientes espacios vacíos:

a) Las listas y las tablas de valores se almacenan en _____.
b) Los elementos de un arreglo están relacionados entre sí por el hecho de que tienen el mismo _____ y _____.
c) El número utilizado para referirnos a un elemento particular de un arreglo se llama su _____.
d) Debe utilizarse un _____ para declarar el tamaño de un arreglo porque el programa hace más dimensionable.
e) El proceso de colocar en orden los elementos de un arreglo se conoce como _____ el arreglo.
f) El proceso de determinar si un arreglo contiene un cierto valor buscado, se conoce como _____ el arreglo
g) Un arreglo que utilice dos subíndices se conoce como un arreglo de _____.

6.2 Declare si lo siguiente es verdadero o falso. Si la respuesta es falsa, explique por qué.

a) Un arreglo puede almacenar muchos tipos diferentes de valores.
b) Un subíndice de arreglo puede ser del tipo de datos `float`.
c) Si existen menos inicializadores en una lista inicializadora que el número de elementos dentro del arreglo, se inicializarán automáticamente C los elementos faltantes con el último valor en la lista de inicializadores.
d) Es un error si la lista de inicializadores contiene más inicializadores que existan elementos dentro del arreglo.
e) Un elemento individual de un arreglo que se pasa a una función y que fue modificado en la función llamada contendrá el valor modificado en la función llamada.

6.3 Conteste las preguntas siguientes relacionadas con un arreglo conocido como `fractions`.

a) Defina una constante simbólica `SIZE`, misma que será remplazada con el texto de remplazo 10.
b) Declare un arreglo con elementos `SIZE` del tipo `float`, e inicialice los elementos a `0`.
c) Dele nombre al cuarto elemento a partir del principio del arreglo.
d) Refiérase al elemento del arreglo 4.
e) Asigne el valor de `1.667` al elemento nueve del arreglo.
f) Asigne el valor `3.333` al séptimo elemento del arreglo.
g) Imprima los elementos 6 y 9 del arreglo con dos dígitos de precisión a la derecha del punto decimal, y muestre la salida que en realidad se despliega en pantalla.
h) Imprima todos los elementos del arreglo utilizando una estructura de repetición `for`. Suponga que la variable entera `x` ha sido definida como variable de control para el ciclo. Muestre la salida.

6.4 Conteste las siguientes preguntas en relación con un arreglo denominado `table`.

a) Declare el arreglo como un arreglo entero con 3 renglones y 3 columnas. Suponga que se ha definido la constante simbólica `SIZE` como de valor 3.
b) ¿Cuántos elementos contiene el arreglo?
c) Utilice una estructura de repetición `for` para inicializar cada elemento del arreglo a la suma de sus subíndices. Suponga que las variables enteras `x` e `y` han sido declaradas como variables de control.
d) Imprima los valores de cada elemento del arreglo `table`. Suponga que el arreglo fue inicializado con la declaración,

```
int table[SIZE][SIZE] = {{1, 8}, {2, 4, 6}, {5}};
```

y las variables enteras `x` e `y` han sido declaradas como variables de control. Muestre la salida.

6.5 Encuentre el error en cada uno de los siguientes segmentos de programa y corrija el error.

a)
```
#define SIZE 100;
```
b)
```
SIZE = 10
```
c) Suponga
```
int b[10] = {0}, i;
for (i = 0; i <= 10; i++)
  b[i] = 1;
```
d)
```
#include <stdio.h>;
```
e) Suponga
```
int a[2][2] = {{1, 2}, {3, 4}};
a[1, 1] = 5;
```

Respuestas a los ejercicios de autoevaluación

6.1 a) Arreglos. b) Nombre, tipo. c) Subíndice. d) Constante simbólica. e) Ordenación. f) Búsqueda. g) Doble subíndice.

6.2 a) Falso. Un arreglo puede sólo almacenar valores del mismo tipo.
b) Falso. El subíndice de un arreglo debe ser un entero o una expresión entera.
c) Falso. C inicializa a cero en forma automática los elementos restantes.
d) Verdadero.
e) Falso. Los elementos individuales de un arreglo son pasados en llamada por valor. Si todo el arreglo se pasa a una función, entonces cualesquiera modificaciones se verán reflejadas en el original.

6.3 a)
```
#define SIZE 10
```
b)
```
Float fractions[SIZE] = {0};
```
c)
```
fractions[3]
```
d)
```
fractions[4]
```
e)
```
fractions[9] = 1.667;
```
f)
```
fractions[6] = 3.333;
```
g)
```
printf("%2f %.2f\n", fractions[6], fractions[9]);
```
Salida: `3.33 1.67.`

h)
```
for (x = 0; x <= SIZE - 1; x++)
  printf("fractions[%d] = %f\n", x, fractions[x]);
```
Salida:
```
fractions[0] = 0.000000
fractions[1] = 0.000000
fractions[2] = 0.000000
fractions[3] = 0.000000
fractions[4] = 0.000000
fractions[5] = 0.000000
fractions[6] = 3.333000
fractions[7] = 0.000000
```

```
fractions[8] = 0.000000
fractions[9] = 1.667000
```

6.4 a) `int table[SIZE][SIZE];`

b) Nueve elementos

c)
```
for (x = 0; x <= SIZE - 1; x++);
   for (y = 0; y <= SIZE - 1; y++)
      table[x][y] = x + y;
```

d)
```
for (x = 0; x <= SIZE - 1; x++)
   for (y = 0; y <= SIZE - 1; y++)
      printf("table[%d][%d] = %d\n", x, y, table[x][y]);0
```

Salida:
```
table[0][0] = 1
table[0][1] = 8
table[0][2] = 0
table[1][0] = 2
table[1][1] = 4
table[1][2] = 6
table[2][0] = 5
table[2][1] = 0
table[2][2] = 0
```

6.5 a) Error: punto y coma al final de la directiva de preprocesador `#define`.
Corrección: Eliminar el punto y coma.

b) Error: asignar un valor a una constante simbólica mediante un enunciado de asignación.
Corrección: asignar un valor a una constante simbólica en una directiva de preprocesador `#define` sin utilizar el operador de asignación como en `#define SIZE 10`.

c) Error: referenciar cualquier elemento del arreglo fuera de los límites del mismo (`b[10]`).
Corrección: cambie el valor final de la variable de control a `9`.

d) Error: punto y coma al final de la directiva de preprocesador `#include`.
Corrección: eliminar el punto y coma.

e) Error: la subindexación del arreglo se ha hecho en forma incorrecta.
Corrección: cambie el enunciado a `a[1][1] = 5;`

Ejercicios

6.6 Llene cada uno de los siguientes espacios vacíos:

a) C almacena listas de valores en _____.

b) Los elementos de un arreglo están relacionados por el hecho de que son _____.

c) Al referirse a un elemento de un arreglo, el número de posición contenido entre paréntesis se conoce como _____.

d) Los nombres de los cinco elementos de un arreglo `p` son _____, _____, _____, _____, y _____.

e) El contenido de un elemento particular de un arreglo se conoce como el _____ de dicho elemento.

f) El ponerle nombre a un arreglo, indicar su tipo, y especificar el número de elementos dentro del mismo, se conoce como _____ el arreglo.

g) El proceso de colocar los elementos de un arreglo en orden ascendente o descendente, se conoce como _____.

h) En un arreglo de doble subíndice, el primer subíndice (por regla convencional) identifica el _____ del elemento, y el segundo subíndice (por regla convencional), identifica el _____ de un elemento.

i) Un arreglo m por n contiene _____ renglones, _____ columnas y _____ elementos.

j) El nombre del elemento en el renglón 3 y en la columna 5 del arreglo `d` es _____.

6.7 Indique cuál de los siguientes es verdadero y cuál es falso; para aquellos que son falsos, explique por qué lo son.

a) Para referirse a una posición o elemento particular dentro de un arreglo, especificamos el nombre del arreglo y el valor del elemento particular.

b) Una declaración de arreglo reserva espacio para el mismo.

c) Para indicar que 100 localizaciones deberán de ser reservadas para el arreglo entero `p`, el programador escribe la declaración

```
p[100];
```

d) Un programa C que inicializa a cero los elemento de un arreglo de 15 elementos debe contener un enunciado `for`.

e) Un programa en C que totaliza los elementos de un arreglo de doble subíndice debe contener enunciados `for` anidados.

f) El promedio, media y modo del siguiente conjunto de valores son 5, 6 y 7 respectivamente: 1, 2, 5, 6, 7, 7, 7.

6.8 Escriba enunciados en C que ejecuten cada uno de los siguientes:

a) Muestre el valor del séptimo elemento del arreglo de caracteres `f`.

b) Introduzca un valor en el elemento 4 de un arreglo de punto flotante de un solo subíndice `b`.

c) Inicialice a `8` cada uno de los 5 elementos de un arreglo entero de un solo subíndice `g`.

d) Totalice los elementos del arreglo de punto flotante `c` de 100 elementos.

e) Copie el arreglo `a` en la primera porción del arreglo `b`. Suponga `float a[11], b[34];`

f) Determine e imprima los valores más pequeños y más grandes contenidos en un arreglo de punto flotante `w` de 99 elementos.

6.9 Suponga un arreglo entero de 2 por 5 de nombre `t`.

a) Escriba una declaración para `t`.

b) ¿Cuántos renglones tiene `t`?

c) ¿Cuántas columnas tiene `t`?

d) ¿Cuántos elementos tiene `t`?

e) Escriba los nombres de todos los elementos del segundo renglón de `t`.

f) Escriba los nombres de todos los elementos de la tercera columna de `t`.

g) Escriba un solo enunciado en C que defina a cero el elemento de `t` en el renglón 1 y columna 2.

h) Escriba una serie de enunciados en C que inicialicen a cero cada elemento de `t`. No utilice una estructura de repetición.

i) Escriba una estructura `for` anidada que inicialice a cero cada elemento de `t`.

j) Escriba un enunciado de C que introduzca los valores para los elementos de `t` desde la terminal.

k) Escriba una serie de enunciados en C que determinen e impriman el valor más pequeño en el arreglo `t`.

l) Escriba un enunciado en C que muestre los elementos del primer renglón de `t`.

m) Escriba un enunciado en C que totalice los elementos de la cuarta columna de `t`.

n) Escriba una serie de enunciados en C que impriman el arreglo `t` en formato tabular nítido. Enliste los subíndices de columna como encabezados en la parte superior, y enliste los subíndices de renglones en la parte izquierda de cada renglón.

6.10 Utilice un arreglo de un subíndice para resolver el siguiente problema. Una empresa le paga a su personal de ventas en base a comisión. Los vendedores reciben $200 por semana más 9 % de sus ventas brutas de dicha semana. Por ejemplo, un vendedor que vende $3000 en ventas brutas en una semana recibe $200 más 9% de 3000, o sea un total de $470. Escriba un programa en C (utilizando un arreglo de contadores) que determine cuántos de los vendedores ganaron salarios en cada uno de los rangos siguientes (suponiendo que el salario de cada vendedor se trunca a una cantidad entera):

1. $200-$299
2. $300-$399
3. $400-$499
4. $500-$599
5. $600-$699
6. $700-$799
7. $800-$899
8. $900-$999
9. $1000 o superior

6.11 La ordenación por el método de burbuja presentada en la figura 6.15, es ineficiente en el caso de arreglos grandes. Haga las siguientes modificaciones simples para mejorar el rendimiento de este tipo de ordenación.

a) Después de la primera pasada el número más alto está garantizado que deberá aparecer en el elemento numerado más alto dentro del arreglo; después de la segunda pasada, los dos números más altos estarán "en su lugar", y así en lo sucesivo. En vez de hacer nueve comparaciones en cada pasada, modifique la ordenación tipo burbuja para llevar a cabo ocho comparaciones en la segunda pasada, 7 en la tercera, y así sucesivamente.

b) Los datos en el arreglo pudieran estar ya en el orden apropiado o en un orden casi apropiado, por lo tanto, ¿por qué hacer nueve pasadas si menos pudieran ser suficientes? Modifique la ordenación para verificar, al final de cada pasada, si se han hecho intercambios. Si no ha habido intercambios, entonces los datos deben de estar ya en el orden apropiado y, por lo tanto, el programa debe darse por terminado. Si ha habido intercambios, entonces por lo menos se requiere de una pasada adicional.

6.12 Escriba enunciados individuales que ejecuten cada una de las operaciones siguientes, de arreglos de un subíndice:

a) Inicialice a ceros los 10 elementos del arreglo entero `counts`.

b) Añada uno a cada uno de los 15 elementos del arreglo entero `bonus`.

c) Lea los 12 valores del arreglo de punto flotante `monthlyTemperatures` desde el teclado.

d) Imprima los 5 valores del arreglo entero `bestScores` en formato de columna.

6.13 Encuentre el error o los errores en cada uno de los enunciados siguientes:

a) Suponga: `char str[5];`
```
scanf("%s", str);      /* User types hello */
```

b) Suponga: `int a [3];`
```
printf("$d %d %d\n", a[1], a[2], a[3]);
```

c)
```
float f[3] = {1.1, 10.01, 100.001, 1000.0001};
```

d) Suponga: `double d[2][10];`
```
d[1, 9] = 2.345;
```

6.14 Modifique el programa de la figura 6.16 para que la función `mode` sea capaz de manejar un empate para el valor modo. También modifique la función `median` de tal forma que en un arreglo que tenga un número par de elementos, los dos elementos del medio sean promediados.

6.15 Utilice un arreglo de un subíndice para resolver el problema siguiente. Lea 20 números, cada uno de los cuales este entre 10 y 100 inclusive. Conforme cada número es leído, imprímalo sólo si no es un duplicado de un número ya leído. Provea para un "caso peor" en el cual los 20 números resulten diferentes. Utilice el arreglo más pequeño posible para resolver este problema.

6.16 Etiquete los elementos de un arreglo de doble subíndice de 3 por 5, de nombre `sales`, para indicar el orden en el cual se definen a cero, mediante el siguiente segmento de programa:

```
for (row = 0; row <= 2; row++)
   for (column = 0; column <= 4; column++)
      sales[row][column] = 0;
```


6.17 ¿Qué es lo que ejecuta el siguiente programa?

```
#include <stdio.h>
#define SIZE 10

int whatIsThis(int [], int);

main()
{
   int total, a[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

   total = whatIsThis(a, SIZE);
   printf("Total of array element values is %d\n", total);
   return 0;
}

int whatIsThis(int b[], int size)
{
   if (size == 1)
      return b[0];
   else
      return b[size - 1] + whatIsThis(b, size - 1);
}
```

6.18 ¿Qué es lo que lleva a cabo el siguiente programa?

```
#include <stdio.h>
#define SIZE 10

void someFunction(int [], int);

main()
{
   int a[SIZE] = {32, 27, 64, 18, 95, 14, 90, 70, 60, 37};

   printf("The values in the array are:\n");
   someFunction(a, SIZE);
   printf("\n");
   return 0;
}

void someFunction(int b[], int size)
{
   if (size > 0) {
      someFunction(&b[1], size - 1);
      printf("%d ",b[0]);
   }
}
```

6.19 Escriba un programa en C que simule el tirar dos dados. El programa deberá utilizar `rand` para tirar el primer dado, y después volverá a utilizar `rand` para tirar el segundo. La suma de los dos valores deberá entonces ser calculada. *Nota:* en vista de que cada dado puede mostrar un valor entero de 1 a 6, entonces la suma de los dos valores variará desde 2 hasta 12, siendo 7 la suma más frecuente y 2 y 12 las menos frecuentes. En la figura 6.23 se muestran las 36 combinaciones posibles de los dos dados. Su programa deberá tirar 36,000 veces los dos dados. Utilice un arreglo de un subíndice para llevar cuenta del número de veces que aparece cada suma posible. Imprima los resultados en un formato tabular. También, determine si los totales son razonables, es decir, existen seis formas de llegar a un 7, por lo que aproximadamente una sexta parte de todas las tiradas deberán ser 7.

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Fig. 6.23 Las 36 posibles combinaciones de tirar dos dados.

6.20 Escriba un programa que ejecute 1000 juegos de craps y que responda a cada una de las preguntas siguientes:

a) ¿Cuántos juegos son ganados en la primera tirada, segunda tirada, ..., tirada número veinte y después de la tirada número veinte?.
b) ¿Cuántos juegos han sido perdidos en la primera tirada, segunda tirada, ..., tirada número veinte y después de la tirada número veinte?.
c) ¿Cuales son las probabilidades de ganar en el craps?. (*Nota:* deberá llegar a la conclusión que el craps es uno de los juegos de casino más justos). ¿Qué es lo que supone que esto significa?
d) ¿Cuál es la longitud o duración promedio de un juego de craps?.
e) ¿Al aumentar la duración del juego aumentan las probabilidades de ganar?

6.21 (*Sistema de reservaciones de aerolínea*). Una pequeña aerolínea acaba de adquirir una computadora para su sistema automatizado de reservaciones. El presidente le ha solicitado a usted que programe el nuevo sistema en C. Usted debe escribir un programa que asigne asientos en cada vuelo del único avión de la aerolínea (capacidad: 10 asientos).

Su programa deberá mostrar el siguiente menú de alternativas:

```
Please type 1 for "smoking"
Please type 2 for "nonsmoking"
```

Si la persona escribe 1, entonces su programa deberá asignar un asiento en la sección de fumar (asientos 1 al 5) si la persona escribe 2, entonces su programa deberá de asignar un asiento en la sección de no fumar (asientos 6 al 10). Su programa a continuación deberá imprimir un pase de abordaje, indicando el número de asiento de la persona y si está en la sección o de no fumar del aeroplano.

Utilice un arreglo de un subíndice para representar el diagrama de asientos del avión. Inicialice todos los elementos del arreglo a cero para indicar que todos los asientos están vacíos. Conforme se asigne cada asiento, defina los elementos correspondientes del arreglo a 1 para indicar que dicho asiento ya no está disponible.

Su programa no deberá, naturalmente, asignar nunca un asiento que ya haya sido asignado. Cuando esté llena la sección de fumar, su programa deberá solicitar a la persona, si le parece aceptable ser colocada en la sección de no fumar (o viceversa). Si dice que sí, entonces efectúe la asignación apropiada de asiento. Si dice que no, entonces imprima el mensaje `"Next flight leaves in 3 hours".`

6.22 Utilice una arreglo de doble subíndice para resolver el problema siguiente. Una empresa tiene cuatro vendedores (1 a 4) que venden cinco productos diferentes (1 a 5). Una vez al día, cada vendedor emite un volante para cada tipo distinto de producto vendido. Cada volante contiene:

1. El número del vendedor.
2. El número del producto.
3. El valor total en dólares del producto vendido ese día.

Por lo tanto, cada vendedor entrega por día entre 0 y 5 volantes de ventas. Suponga que está disponible la información de todos los volantes correspondientes al mes anterior. Escriba un programa que lea toda esta información correspondiente a las ventas del mes anterior, y que resuma las ventas totales por vendedor y por producto. Todos los totales deberán almacenarse en un arreglo de doble subíndice `sales`. Después de procesar toda la información correspondiente al mes anterior, imprima los resultados en forma tabular, con cada una de las columnas representando a un vendedor en particular y cada uno de los renglones representando un producto en particular. Totalice en forma cruzada cada renglón, para obtener las ventas totales de cada producto del mes pasado; totalice cada columna para obtener las ventas totales por vendedor correspondiente al mes pasado. Su impresión en forma tabular deberá incluir estos totales a la derecha de los renglones totalizados y en la parte inferior de las columnas totalizadas.

6.23 (*Gráficos tipo tortuga*). El lenguaje Logo, que es en particular popular entre usuarios de computadoras personales, hizo famoso el concepto de los *gráficos tipo tortuga*. Imagine una tortuga mecánica, que camina por la habitación bajo el control de un programa de C. La tortuga sujeta una pluma en dos posiciones posibles, arriba o abajo. Cuando la pluma está abajo, la tortuga traza formas conforme se mueve; cuando la pluma está arriba, la tortuga se mueve a su antojo libremente, sin escribir nada. En este problema simularemos la operación de la tortuga y además crearemos un bloque de notas computarizado.

Utilice un arreglo de 50 por 50 de nombre `floor`, que se inicializa a ceros. Lea los comandos partiendo de un arreglo que los contenga. Lleve control en todo momento de la posición actual de la tortuga, así como de si la pluma en ese momento está arriba o abajo. Suponga que la tortuga siempre empieza a partir de la posición 0,0 en el piso, con su pluma arriba. El conjunto de comandos de la tortuga que su programa debe procesar, son como sigue:

Comando	Significado
1	Pluma arriba
2	Pluma abajo
3	Giro a la derecha
4	Giro a la izquierda
5,10	Moverse hacia adelante 10 espacios (o un número distinto que 10)
6	Imprima el arreglo de 50 por 50
9	Fin de los datos (valor centinela)

Suponga que la tortuga está en algún lugar cerca del centro del piso. El siguiente "programa" dibujaría e imprimiría un cuadrado de 12 por 12:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

Conforme la tortuga se mueve con la pluma abajo, defina los elementos apropiados del arreglo `floor` al valor **1**. Cuando se da el comando **6** (imprimir), siempre que exista en el arreglo un **1**, despliegue un asterisco, o cualquier otro carácter que seleccione. Siempre que aparezca un **0**, despliegue un espacio vacío. Escriba un programa en C para poner en operación las capacidades gráficas de la tortuga discutidas aquí. Escriba varios programas gráficos de tortuga para dibujar formas interesantes. Añada otros comandos para incrementar el poder del lenguaje gráfico de su tortuga.

6.24 (*Recorrido del caballo*). Uno de los acertijos más interesantes para los aficionados al ajedrez, es el problema del recorrido del caballo, propuesto originalmente por el matemático Euler. La pregunta es la siguiente: ¿puede la pieza de ajedrez conocida como el caballo moverse en el interior de un tablero de ajedrez vacío y entrar en contacto con cada una de las 64 casillas, una vez y sólo una vez? Estudiamos aquí este problema interesante en profundidad.

El caballo hace movimientos en forma de L (dos en una dirección y a continuación uno en una dirección perpendicular). Entonces, a partir de una casilla en la mitad de un tablero de ajedrez vacío, el caballo puede efectuar ocho movimientos diferentes (numerados desde 0 hasta 7) tal y como se muestra en la figura 6.24.

a) Dibuje un tablero de ajedrez de 8 por 8 en una hoja de papel e intente a mano un recorrido de caballo. Coloque un **1** en la primera casilla a la cual se mueva, un **2** sobre la segunda, un **3** sobre la tercera, etcétera. Antes de iniciar el recorrido, estime qué tan lejos piensa que pueda llegar, recordando que un recorrido completo consistiría de 64 movimientos. ¿Qué tan lejos llegó? ¿Se aproximó en su estimación?

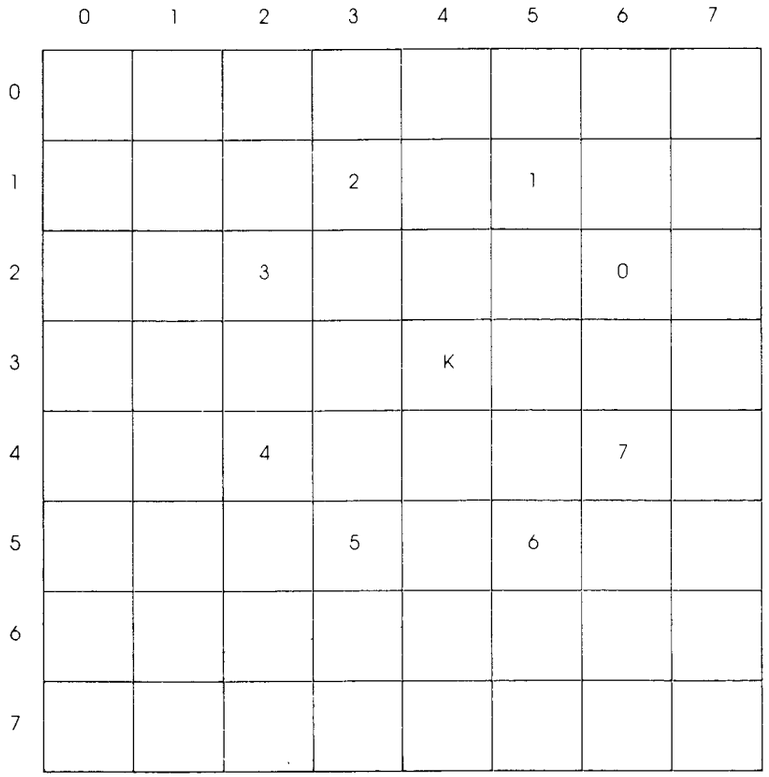


Fig. 6.24 Los ocho movimientos posibles del caballo.

b) Ahora desarrollemos un programa que mueva al caballo sobre el tablero. El tablero mismo se representa por un arreglo de doble subíndice de 8 por 8 de nombre `board`. Cada una de las casillas se inicializa a cero. Describimos cada uno de los ocho movimientos posibles en términos tanto de sus componentes horizontales como verticales. Por ejemplo, un movimiento del tipo 0, tal y como se muestra en la figura 6.24, consiste en mover horizontalmente dos casillas a la derecha y una casilla de forma vertical hacia arriba. El movimiento 2 consiste en mover de manera horizontal una casilla hacia la derecha y dos casillas verticalmente hacia arriba. Los movimientos horizontales a la izquierda y los movimientos verticales hacia abajo se indicarán con números negativos. Los ocho movimientos pueden ser descritos en dos arreglos de un solo subíndice, `horizontal` y `vertical`, como sigue:

```
horizontal[0] = 2
horizontal[1] = 1
horizontal[2] = -1
horizontal[3] = -2
horizontal[4] = -2
horizontal[5] = -1
horizontal[6] = 1
horizontal[7] = 2

vertical[0] = -1
vertical[1] = -2
vertical[2] = -2
vertical[3] = -1
vertical[4] = 1
vertical[5] = 2
vertical[6] = 2
vertical[7] = 1
```

Hagamos las variables `currentRow` y `currentColumn` indicar el renglón y la columna de la posición actual del caballo. Para hacer un movimiento del tipo `moveNumber`, donde `moveNumber` quede entre 0 y 7, su programa utiliza los enunciados

```
currentRow += vertical[moveNumber];
currentColumn += horizontal[moveNumber];
```

Lleve un contador que varíe desde **1** hasta **64**. Registre la última cuenta en cada casilla movida por el caballo. Recuerde en probar cada movimiento potencial, para asegurarse que el caballo no haya visitado ya esa casilla. Y, naturalmente, pruebe cada movimiento potencial, para asegurarse que el caballo no se ha salido del tablero de ajedrez. Ahora escriba un programa para mover el caballo por el teclado. Ejecute el programa. ¿Cuántos movimientos hizo el caballo?

c) Después de intentar escribir y ejecutar un programa del recorrido del caballo, probablemente usted ha desarrollado algunos conocimientos valiosos. Los utilizaremos para desarrollar una *heurística* (o estrategia) para el movimiento del caballo. La heurística no garantiza el éxito, pero una heurística cuidadosamente desarrollada mejora en forma importante la oportunidad de éxito. Quizás haya observado que las casillas externas son en cierta forma más difíciles que las casillas más cercanas al centro del tablero. De hecho, las más difíciles o inaccesibles de las casillas son las de las cuatro esquinas.

La intuición le puede sugerir que debería intentar primero mover el caballo a las casillas más problemáticas, y dejar abiertas aquellas que son más fáciles de llegar, a fin de que conforme se vaya congestionando el tablero cerca del fin del recorrido, existan mayores oportunidades de éxito.

Pudiéramos desarrollar una "heurística de accesibilidad" ordenando cada una de las casillas de acuerdo a su accesibilidad, y a partir de eso, moviendo siempre el caballo a la casilla (dentro de los movimientos aceptados del caballo, naturalmente) que sea más inaccesible.

Etiquetamos un arreglo de doble subíndice `accessibility`, con números que indiquen desde cuántas casillas es accesible una casilla en particular. En un tablero en blanco o vacío, las casillas centrales serán, por lo tanto, tasadas o valuadas como `8`, las casillas de esquina como `2`, y las demás casillas tendrán números de accesibilidad de `3`, `4`, o `6` como sigue:

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 6 3
2 3 4 4 4 4 3 2
```

Ahora escriba una versión del programa del recorrido del caballo utilizando la heurística de accesibilidad. En todo momento, el caballo deberá moverse a la casilla que tenga el número de accesibilidad más bajo. En caso de empate, el caballo deberá moverse a cualquiera de las casillas de empate. Por lo tanto, el recorrido pudiera empezar en cualquiera de las cuatro esquinas. (*Nota:* conforme el caballo se mueva por el tablero, su programa debería de reducir los números de accesibilidad conforme más y más casillas se van ocupando. De esta forma, en cualquier momento dado durante el recorrido, cada número de accesibilidad de las casillas disponibles se conservará igual precisamente el número de casillas que pueden ser alcanzadas). Ejecute esta versión de su programa. ¿Obtuvo un recorrido completo? Ahora modifique el programa para ejecutar 64 recorridos, uno partiendo de cada una de las casillas del tablero. ¿Cuántos recorridos completos obtuvo?

d) Escriba una versión del programa del recorrido del caballo en la cual, al encontrar un empate entre dos o más casillas, el programa decida cual casilla seleccionar mirando hacia adelante cuáles son las casillas alcanzables a partir de las casillas de "empate". Su programa deberá moverse a la casilla a partir de la cual el siguiente movimiento llegaría a una casilla con el número de accesibilidad menor.

6.25 (*Recorrido del caballo: enfoques de fuerza bruta*). En el ejercicio 6.24 desarrollamos una solución al programa del recorrido del caballo. El enfoque utilizado, conocido como "heurística de accesibilidad", genera muchas soluciones y se ejecuta con eficiencia.

Conforme las computadoras siguen aumentando en potencia, seremos capaces de resolver muchos problemas a base del simple poder de la computadora y utilizando algoritmos relativamente poco complicados. Llamémosle a este enfoque la solución de problemas de "fuerza bruta".

a) Utilice la generación de números aleatorios para permitir que el caballo se desplace sobre el tablero de ajedrez (en movimientos legítimos en forma de L, naturalmente) en forma aleatoria. Su programa debería ejecutar un recorrido e imprimir el tablero de ajedrez final. ¿Qué tan lejos llegó el caballo?

b) Lo más probable es que el programa anterior produjo un recorrido relativamente corto. Ahora modifique su programa para intentar 1000 recorridos. Utilice un arreglo de un solo subíndice para llevar control del número de recorridos de cada longitud. Cuando su programa termine el intento de 1000 recorridos, deberá imprimir esta información en un formato tabular nítido. ¿Cuál fue el resultado mejor?

c) Lo más probable, es que el programa anterior le dió algunos recorridos "respetables" pero ningún recorrido completo. Ahora "quite todas las amarras" y deje sólo que su programa se ejecute hasta que produzca un recorrido completo. (*Precaución.* Esta versión del programa podría ejecutarse durante horas en una computadora poderosa). Aquí otra vez, lleve una tabla del número de recorridos de cada longitud, e imprima esta tabla cuando se encuentre el primer recorrido completo. ¿Cuántos recorridos tuvo que intentar su programa antes de producir un recorrido completo? ¿Cuánto tiempo le tomó?

d) Compare la versión de fuerza bruta del recorrido del caballo con la versión de la heurística de accesibilidad. ¿Cuál requirió un estudio más cuidadoso del problema? ¿Qué algoritmo fue más difícil de desarrollar? ¿Cuál requirió de más potencia de la computadora? ¿Podríamos estar seguros (por anticipado) de poder llegar a obtener un recorrido completo con el enfoque heurístico de accesibilidad? ¿Podíamos estar seguros (con anticipación) de obtener un recorrido completo con el enfoque de fuerza bruta? Discuta los pros y los contras de la resolución en general de problemas por fuerza bruta.

6.26 (*Las ocho reinas*). Otro acertijo para aficionados al ajedrez es el problema de las ocho reinas. Dicho simplemente: es o no posible colocar ocho reinas en un tablero de ajedrez vacío, de tal forma que ninguna reina esté atacando a ninguna otra, esto es de tal forma que ¿ningún par de reinas estén en el mismo renglón, la misma columna o a lo largo de la misma diagonal? Utilice el tipo de proceso mental desarrollado en el ejercicio 6.24 para formular una heurística para resolver el problema de las ocho reinas. Ejecute su programa. (*Sugerencia:* es posible asignar un valor numérico a cada casilla del tablero, indicando cuántas casillas son "eliminadas" de un tablero vacío, una vez que se ha colocado una reina en dicha casilla. Por ejemplo, a cada una de las cuatro esquinas deberá asignársele el valor 22, como en la figura 6.25).

Una vez que esos "números de eliminación" son colocados en las 64 casillas, la heurística apropiada podría ser: coloque la reina siguiente en la casilla con el número de eliminación más pequeño. ¿Por qué es de forma intuitiva atrayente esta estrategia?

6.27 (*Ocho reinas: enfoques de fuerza bruta*). En este problema desarrollará varios enfoques de fuerza bruta para resolver el problema de las ocho reinas introducido en el ejercicio 6.26.

a) Resuelva el problema de las ocho reinas, utilizando la técnica de fuerza bruta al azar desarrollada en el problema 6.25.

b) Utilice una técnica exhaustiva, es decir, pruebe todas las combinaciones posibles de las ocho reinas sobre el tablero.

c) ¿Por qué supone que el enfoque exhaustivo de fuerza bruta pudiera no ser apropiado para la resolución del problema del recorrido del caballo?

d) Compare y resalte las diferencias en general de los enfoques de fuerza bruta aleatoria y fuerza bruta exhaustiva.

6.28 (*Eliminación de duplicados*). En el capítulo 12 exploramos la estructura de datos del árbol de búsqueda binaria de alta velocidad. Una característica del árbol de búsqueda binaria, es que los valores duplicados son descartados al hacer inserciones en el árbol. Esto se conoce como eliminación de duplicados. Escriba un programa que produzca 20 números al azar entre 1 y 20. El programa deberá almacenar en un arreglo todos los valores no duplicados. Utilice para esta tarea el arreglo más pequeño posible.

6.29 (*Recorrido del caballo: prueba del recorrido cerrado*). En el recorrido del caballo, un recorrido completo es aquel en que el caballo efectúa 64 movimientos tocando cada casilla del tablero de ajedrez una vez y sólo una vez. Un recorrido cerrado ocurre cuando el 64º movimiento queda a un movimiento de distancia de la posición en la cual el caballo inició el recorrido. Modifique el programa del recorrido del caballo, que escribió en el ejercicio 6.24, para probar para un recorrido cerrado, cuando haya ocurrido un recorrido completo.

Fig. 6.25 Las 22 casillas eliminadas al colocar una reina en la esquina superior izquierda.

6.30 (*La criba de Erastostenes*). Un entero primo es cualquier entero que puede ser sólo dividido entre sí mismo y entre 1. La criba de Erastostenes es un método para encontrar los números primos. Funciona como sigue:

1) Instituya un arreglo con todos los elementos inicializados a 1 (verdadero). Los elementos del arreglo con subíndices primos se conservarán en 1. Todos los otros elementos del arreglo eventualmente se quedarán en cero.
2) Empezando con el subíndice 2 del arreglo (el subíndice 1 debe ser primo), cada vez que se encuentre un elemento de arreglo cuyo valor sea 1, cicle a través del resto del arreglo y defina como cero cualquier elemento cuyo subíndice resulte un múltiplo del subíndice correspondiente al elemento con valor 1. Para el subíndice 2 del arreglo, todos los elementos más allá de 2 dentro del arreglo que sean múltiplos de 2, serán valuados en cero (subíndices 4, 6, 8, 10, etcétera). Para el subíndice del arreglo 3, todos los elementos más allá de 3 en el arreglo que sean múltiples de 3, serán valuados en cero (subíndices 6, 9, 12, 15, etcétera).

Cuando se haya terminado este proceso, los elementos del arreglo que aún estén definidos como 1, indicarán que el subíndice es un número primo. Entonces esos subíndices pueden ser impresos. Escriba un programa que utilice un arreglo de 1000 elementos para determinar e imprimir los números primos entre 1 y 999. Ignore el elemento cero del arreglo.

6.31 (*Ordenación tipo cubeta*). Una ordenación tipo cubeta empieza con un arreglo de un índice de enteros positivos a ordenar, un arreglo de doble subíndice de enteros con renglones con subíndices desde 0 hasta 9 y con columnas con subíndices desde 0 hasta *n* - 1, donde *n* es el número de valores dentro del arreglo a ordenar. Cada renglón del arreglo de doble subíndice se conoce como una cubeta. Escriba una función `bucketSort`, que tome un arreglo de enteros y el tamaño del arreglo como argumentos.

El algoritmo es como sigue:

1) Cicle a través del arreglo de un subíndice y coloque cada uno de sus valores en un renglón del arreglo de cubeta basado en sus dígitos uno. Por ejemplo, 97 se coloca en el renglón 7, 3 se coloca en el renglón 3, y 100 se coloca en el renglón 0.
2) Cicle a través del arreglo de cubeta y copie los valores de regreso al arreglo original. El nuevo orden de los valores anteriores en el arreglo de un solo subíndice es 100, 3 y 97.
3) Repita este proceso para cada posición digital subsecuente (decenas, centenas, miles, etcétera), y deténgase cuando se haya procesado el dígito más a la izquierda del número mayor.

En la segunda pasada del arreglo, 100 se coloca en el renglón 0, 3 se coloca en el renglón 0 (sólo tenía un dígito), y 97 se coloca en el renglón 9. El orden de los valores en el arreglo de un solo subíndice es 100, 3 y 97. En la tercera pasada, 100 se coloca en el renglón 1, 3 se coloca en el renglón 0 y 97 se coloca en el renglón 0 (después de 3). La ordenación por cubeta garantiza tener todos los valores de forma correcta clasificados, una vez procesado el dígito más a la izquierda del número más grande. La ordenación por cubeta sabe que ha terminado, cuando todos los valores se copian en el renglón cero del arreglo de doble subíndice.

Advierta que el arreglo de doble subíndice de cubetas es diez veces del tamaño del arreglo entero ordenándose. Esta técnica de ordenación permite un mayor rendimiento que una ordenación tipo burbuja, pero requiere de una capacidad de almacenamiento mucho mayor. La ordenación tipo burbuja sólo requiere de una localización de memoria adicional para el tipo de datos ordenándose. La ordenación por cubeta es un ejemplo de intercambio, espacio-tiempo. Utiliza más memoria, pero da un mejor rendimiento. Esta versión de la ordenación de cubeta requiere el copiado de todos los datos de regreso al arreglo original en cada una de las pasadas. Otra posibilidad es la creación de un segundo arreglo de cubeta de doble subíndice, y mover repetidamente los datos entre los dos arreglos de cubeta, hasta que todos los datos quedan copiados en el renglón cero de uno de ellos. El renglón cero, contendrá entonces el arreglo ordenado.

Ejercicios de recursión

6.32 (*Ordenación de selección*). Una ordenación de selección recorre un arreglo buscando el elemento más pequeño del mismo. Cuando encuentra el más pequeño, es intercambiado con el primer elemento del

arreglo. El proceso a continuación se repite para el subarreglo que empieza con el segundo elemento del arreglo. Cada pasada del arreglo resulta en un elemento colocado en su posición correcta. Esta ordenación requiere de capacidades de procesamientos similares a la ordenación tipo burbuja para un arreglo de *n* elementos, deberán de hacerse *n* - 1 pasadas, y para cada subarreglo, se harán *n* - 1 comparaciones para encontrar el valor más pequeño. Cuando el subarreglo bajo proceso contenga un solo elemento, el arreglo habrá quedado terminado y ordenado. Escriba una función recursiva `selectionSort` para ejecutar este algoritmo.

6.33 (*Palíndromos*). Un palíndromo es una cadena que se escribe de la misma forma hacia adelante y hacia atrás. Algunos ejemplos de palíndromos son "radar", "able was i ere i saw elba", y "a man a plan a canal panama". Escriba una función recursiva `testPalindrome`, que devuelva uno si la cadena almacenada en el arreglo es un palíndromo y 0 de lo contrario. La función deberá ignorar espacios y puntuaciones incluidas en la cadena.

6.34 (*Búsqueda lineal*). Modifique el programa de la figura 6.18 para utilizar una función recursiva `linearSearch` para ejecutar la búsqueda lineal del arreglo. La función deberá recibir como argumentos un arreglo entero y el tamaño del arreglo. Si se encuentra el valor buscado, devuelva el subíndice del arreglo; de lo contrario devuelva -1.

6.35 (*Búsqueda binaria*). Modifique el programa de la figura 6.19 para utilizar una función recursiva `binarySearch` para ejecutar la búsqueda binaria del arreglo. La función deberá recibir como argumentos un arreglo entero y el subíndice inicial y final. Si el valor buscado es hallado, devuelva el subíndice del arreglo; de lo contrario, devuelva -1.

6.36 (*Ocho reinas*). Modifique el programa ocho reinas creado en el ejercicio 6.26 para resolver el problema en forma recursiva.

6.37 (*Imprima un arreglo*). Escriba una función recursiva `printArray` que toma un arreglo y el tamaño del arreglo como argumentos y que no devuelva nada. La función deberá dejar de procesar y regresar cuando reciba un arreglo de tamaño cero.

6.38 (*Imprimir una cadena de atrás para adelante*). Escriba una función recursiva `stringReverse` que tome un arreglo de caracteres como argumento y que no regrese nada. La función deberá dejar de procesar y regresar cuando se encuentre el carácter nulo de terminación de la cadena.

6.39 (*Encontrar el valor mínimo en un arreglo*). Escriba la función recursiva `recursiveMinimum`, que toma un arreglo entero y el tamaño del arreglo como argumentos y regresa el elemento más pequeño del mismo. La función deberá detener su proceso y regresar cuando reciba un arreglo de un solo elemento.

UNIVERSIDAD DE LA REPUBLICA
FACULTAD DE INGENIERIA
DEPARTAMENTO DE
DOCUMENTACION Y BIBLIOTECA
MONTEVIDEO - URUGUAY

7

Apuntadores

Objetivos

- Ser capaz de utilizar apuntadores.
- Ser capaz de utilizar apuntadores para pasar argumentos a las funciones en llamada por referencia.
- Comprender las relaciones íntimas entre apuntadores, arreglos y cadenas.
- Comprender la utilización de apuntadores a funciones.
- Ser capaz de declarar y utilizar arreglos de cadenas.

Se nos asignan direcciones para ocultar nuestra ubicación.
Saki (H.H. Munro)

Averigüe las instrucciones por medio de la falta de indicaciones.
William Shakespeare
Hamlet

Muchas cosas, con plena referencia
A nuestra complacencia, podrían funcionar en forma inversa.
William Shakespeare
King Henry V

¡Encontrará que resulta excelente el verificar siempre sus referencias, señor!
Dr. Routh

No es posible confiar en un código que no haya usted creado por sí mismo.
(En especial códigos que provengan de empresas que emplean a personas como yo).
Ken Thompson
1983 Turing Award Lecture Association for Computing Machinery, Inc.

Sinopsis

Resumen • Terminología • Errores comunes de programación • Prácticas sanas de programación • Sugerencias de rendimiento • Sugerencias de portabilidad • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios • Sección especial: cómo construir su propia computadora.

7.1 Introducción

En este capítulo, analizamos una de las características más poderosas del lenguaje de programación en C, el *apuntador*. Los apuntadores son las capacidades más difíciles de dominar en C. Los apuntadores le permiten a los programas simular llamadas por referencia, crear y manipular estructuras de datos, es decir, estructuras de datos que pueden crecer o encogerse, como son listas enlazadas, colas de espera, pilas y árboles. Este capítulo explica conceptos básicos de los apuntadores. En el capítulo 10 se examina el uso de apuntadores junto con las estructuras. En el capítulo 12 se presentan las técnicas de administración dinámica de la memoria y se presentan ejemplos de creación y uso de estructuras dinámicas de datos.

7.2 Declaraciones e inicialización de variables de apuntadores

Los apuntadores son variables que contienen direcciones de memoria como sus valores. Por lo regular una variable contiene directamente un valor específico. Un apuntador, por otra parte, contiene la dirección de una variable que contiene un valor específico. En este sentido, un nombre de variable se refiere *directamente* a un valor y un apuntador se refiere *indirectamente* a un valor (figura 7.1). El referirse a un valor a través de un apuntador se conoce como *indirección.*

Los apuntadores, como cualquier otra variable, deben ser declarados antes de que puedan ser utilizados. La declaración

```
int *countPtr, count;
```

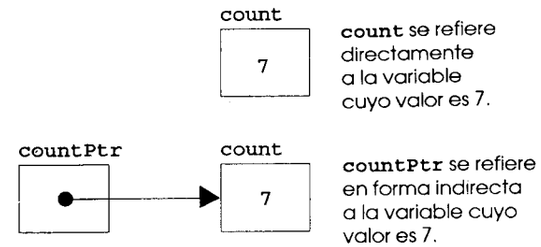



Fig. 7.1 Referenciación directa e indirecta de una variable.

declara la variable `countPtr` siendo del tipo `int*` (es decir, un apuntador a un valor entero) y se lee, "`countPtr` es un apuntador a `int`", o bien "`countPtr` apunta a un objeto del tipo entero". También, la variable `count` se declara como un entero, no un apuntador a un entero. El `*` sólo se aplica a `countPtr` en la declaración. Cuando el `*` se utiliza de esta forma en una declaración, indica que la variable que se está declarando es un apuntador. Los apuntadores pueden ser declarados para apuntar a objetos de cualquier tipo de datos.

Error común de programación 7.1

El operador de indirección `*` no se distribuye a todos los nombres de variables de una declaración. Cada apuntador debe de ser declarado con el `*` prefijo al nombre.

Práctica sana de programación 7.1

En un nombre de variable de apuntador incluya las letras `ptr` para que quede claro que estas variables son apuntadores y deben ser manejadas de forma apropiada.

Los apuntadores deben ser inicializados cuando son declarados o en un enunciado de asignación. Un apuntador puede ser inicializado a `0`, `NULL`, o a una dirección. Un apuntador con el valor `NULL` apunta a nada. `NULL` es una constante simbólica, definida en el archivo de cabecera `<stdio.h>` (y en varios otros archivos de cabecera). Inicializar un apuntador a `0` es equivalente a inicializar un apuntador a `NULL`, pero es preferible `NULL`. Cuando se asigna `0`, primero se convierte a un apuntador del tipo apropiado. El valor `0` es el único valor entero que puede ser directamente asignado a una variable de apuntador. En la sección 7.3 se analiza cómo asignar la dirección de una variable a un apuntador.

Práctica sana de programación 7.2

Inicialice los apuntadores para evitar resultados inesperados.

7.3 Operadores de apuntador

El `&`, u *operador de dirección,* es un operador unario que regresa la dirección de su operando. Por ejemplo, suponiendo las declaraciones

```
int y = 5;
int *yPtr;
```

el enunciado

```
yPtr = &y;
```

asigna la dirección de la variable **y** a la variable de apuntador **yPtr**. La variable **yPtr** se dice entonces que "apunta a" **y**. La figura 7.2 muestra una representación esquemática de la memoria, después de que se ejecuta la asignación anterior.

La figura 7.3 muestra la representación del apuntador en memoria, suponiendo que la variable entera **y** se almacena en la posición **600000**, y la variable de apuntador **yPtr** se almacena en la posición **500000**. El operando del operador de dirección debe ser una variable; el operador de dirección no puede ser aplicado a constantes, a expresiones, o a variables declaradas con la clase de almacenamiento **register**.

El operador *****, conocido comúnmente como el *operador de indirección* o de *desreferencia*, regresa el valor del objeto hacia el cual su operando apunta (es decir, un apuntador). Por ejemplo, el enunciado

```
printf("%d", *yPtr);
```

imprime el valor de la variable **y**, es decir 5. Utilizar a ***** de esta forma se conoce como *desreferenciar a un apuntador*.

Error común de programación 7.2

Desreferenciar un apuntador que no haya sido correctamente inicializado, o que no haya sido asignado para apuntar a una posición específica en memoria. Esto podría causar un error fatal en tiempo de ejecución, o podría modificar de forma accidental datos importantes y permitir que el programa se ejecute hasta su terminación proporcionando resultados incorrectos.

El programa en la figura 7.4 demuestra los operadores de apuntador. La especificación de conversión de **printf** **%p** extrae la localización de memoria en forma de entero hexadecimal (vea el Apéndice E, Sistemas numéricos, para mayor información sobre enteros hexadecimales).

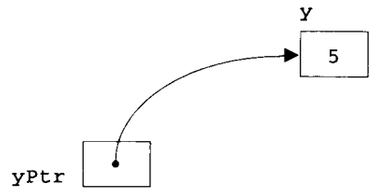


Fig. 7.2 Representación gráfica de un apuntador apuntando a una variable entera en memoria.

Fig. 7.3 Representación en memoria de **y** y **yPtr**.

Note que la dirección de **a** y el valor de **aPtr** son idénticos en la salida, confirmando así que de hecho la dirección de **a** ha sido asignada a la variable de apuntador **aPtr**. Los operadores **&** y ***** son complementos el uno del otro —cuando se aplican ambos de manera consecutiva a **aPtr**, en cualquier orden, el mismo resultado será impreso. La gráfica en la figura 7.5, muestra la precedencia y asociatividad de los operadores presentados hasta este momento.

7.4 Cómo llamar funciones por referencia

Existen dos formas de pasar argumentos a un función llamada por valor y llamada por referencia. En C todas las llamadas de función son llamadas por valor. Como vimos en el capítulo 5, se puede utilizar **return** para regresar un valor de una función llamada hacia el llamador (o para regresar el control de una función llamada sin regresar un valor). Muchas funciones requieren la capacidad de modificar una o más variables del llamador, o de pasar un apuntador a un objeto de datos grande, para evitar la sobrecarga de pasar el objeto en llamada por valor (lo que, naturalmente,

```
/* Using the & and * operators */
#include <stdio.h>

main()
{
   int a;        /* a is an integer */
   int *aPtr;    /* aPtr is a pointer to an integer */

   a = 7;
   aPtr = &a;    /* aPtr set to address of a */

   printf("The address of a is %p\n"
          "The value of aPtr is %p\n\n", &a, aPtr);

   printf("The value of a is %d\n"
          "The value of *aPtr is %d\n\n", a, *aPtr);

   printf("Proving that * and & are complements of "
          "each other.\n&*aPtr = %p\n*&aPtr = %p\n",
          &*aPtr, *&aPtr);
   return 0;
}
```

```
The address of a is FFF4
The value of aPtr is FFF4

The value of a is 7
The value of *aPtr is 7

Proving that * and & are complements of each other.
&*aPtr = FFF4
*&aPtr = FFF4
```

Fig. 7.4 Los operadores de apuntador **&** y *****.

Operadores								Asociatividad	Tipo
()	[]							de izquierda a derecha	oculto
+	-	++	--	!	*	&	(type)	de izquierda a derecha	unario
*	/	%						de izquierda a derecha	multiplicativo
+	-							de izquierda a derecha	aditivo
<	<=	>	>=					de izquierda a derecha	relación
==	!=							de izquierda a derecha	igualdad
&&								de izquierda a derecha	y lógico
||								de izquierda a derecha	o lógico
?:								de izquierda a derecha	condicional
=	+=	-=	*=	/=	%=			de izquierda a derecha	asignación
,								de izquierda a derecha	coma

Fig. 7.5 Precedencia de operadores.

requiere el hacer una copia del objeto). Para estos fines, C proporciona las capacidades de simulación de llamadas por referencia.

En C, los programadores utilizan apuntadores y el operador de indirección para simular llamadas por referencia. Cuando se llama a una función con argumentos que deban ser modificados, se pasan las direcciones de los argumentos. Esto se lleva a cabo normalmente aplicando el operador de dirección (**&**), a la variable cuyo valor deberá ser modificado. Como se vió en el capítulo 6, los arreglos no son pasados mediante el operador **&** porque C pasa de forma automática la posición inicial en memoria del arreglo (el nombre de un arreglo es equivalente a **&arrayName[0]**). Cuando se pasa a una función la dirección de una variable, el operador de indirección (*****), puede ser utilizado en la función para modificar el valor de esa posición en la memoria del llamador.

Los programas de las figuras 7.6 y 7.7 presentan dos versiones de una función que eleva un entero al cubo —**cubeByValue** y **cubeByReference**. El programa de la figura 7.6 pasa la variable **number** a la función **cubeByValue**, utilizando llamada por valor. La función **cubeByValue** eleva al cubo su argumento y pasa su nuevo valor de regreso a **main**, utilizando el enunciado **return**. El nuevo valor se asigna al **number** en **main**.

El programa de la figura 7.7 pasa la variable **number** utilizando llamada por referencia —se pasa la dirección de **number**— a la función **cubeByReference**. La función **cubeByReference** toma un apuntador a **int** conocido como **nPtr** como argumento. La función desreferencia el apuntador y eleva al cubo el valor hacia el cual apunta **nPtr**. Esto cambia el valor de **number** dentro de **main**. Las figuras 7.8 y 7.9 analizan de forma gráfica los programas de las figuras 7.6 y 7.7, respectivamente.

Error común de programación 7.3

No desreferenciar un apuntador, cuando es necesario hacerlo para obtener el valor hacia el cual el apuntador señala.


```
/* Cube a variable using call by value */
#include <stdio.h>

int cubeByValue(int);

main()
{
   int number = 5;

   printf("The original value of number is %d\n", number);
   number = cubeByValue(number);
   printf("The new value of number is %d\n", number);
   return 0;
}

int cubeByValue(int n)
{
   return n * n * n;   /* cube local variable n */
}
```

```
The original value of number is 5
The new value of number is 125
```

Fig. 7.6 Elevación al cubo de una variable, utilizando llamada por valor.

```
/* Cube a variable using call by reference */
#include <stdio.h>

void cubeByReference(int *);

main()
{
   int number = 5;

   printf("The original value of number is %d\n", number);
   cubeByReference(&number);
   printf("The new value of number is %d\n", number);
   return 0;
}

void cubeByReference(int *nPtr)
{
   *nPtr = *nPtr * *nPtr * *nPtr;  /* cube number in main */
}
```

```
The original value of number is 5
The new value of number is 125
```

Fig. 7.7 Elevación al cubo de una variable, utilizando llamada por referencia.

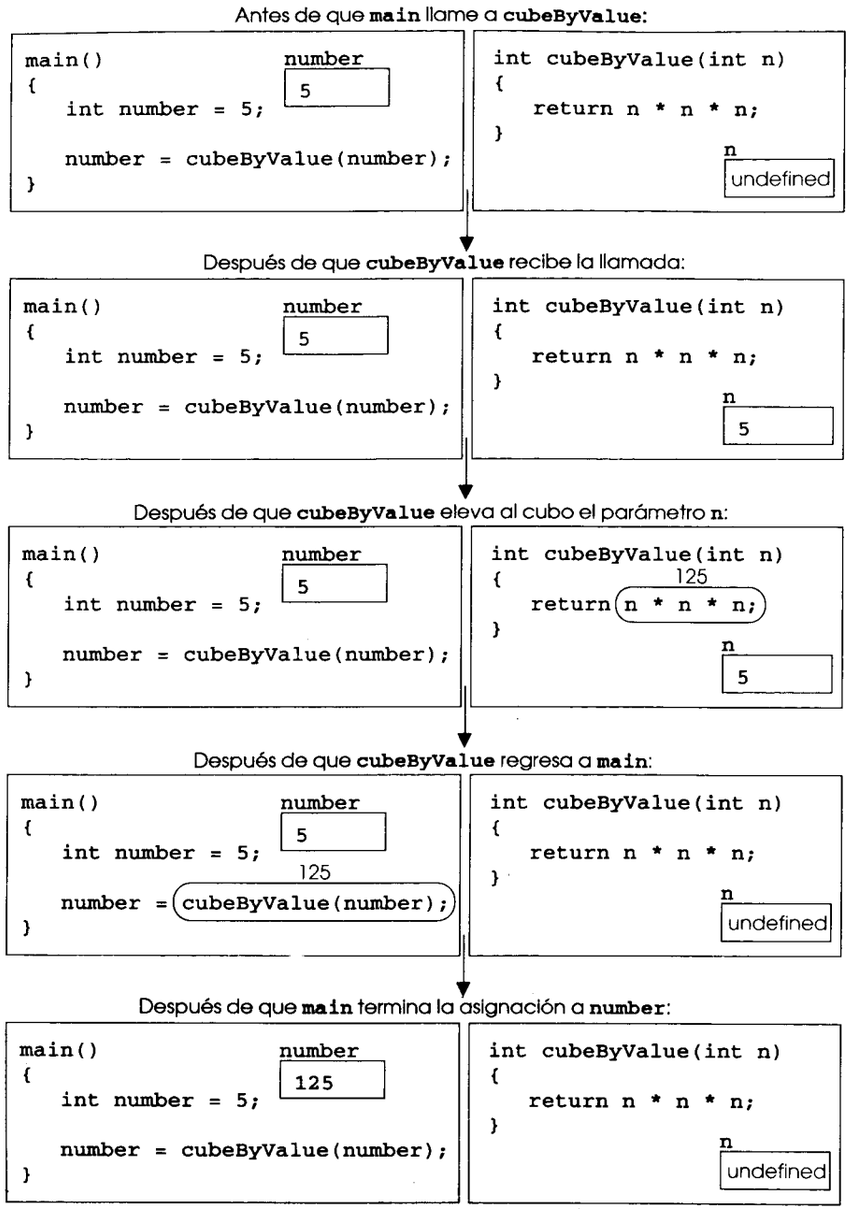


Fig. 7.8 Análisis de una llamada por valor típica.

Una función que recibe una dirección como argumento debe definir un parámetro de un apuntador para recibir la dirección. Por ejemplo, el encabezado para la función **cubeByReference** es

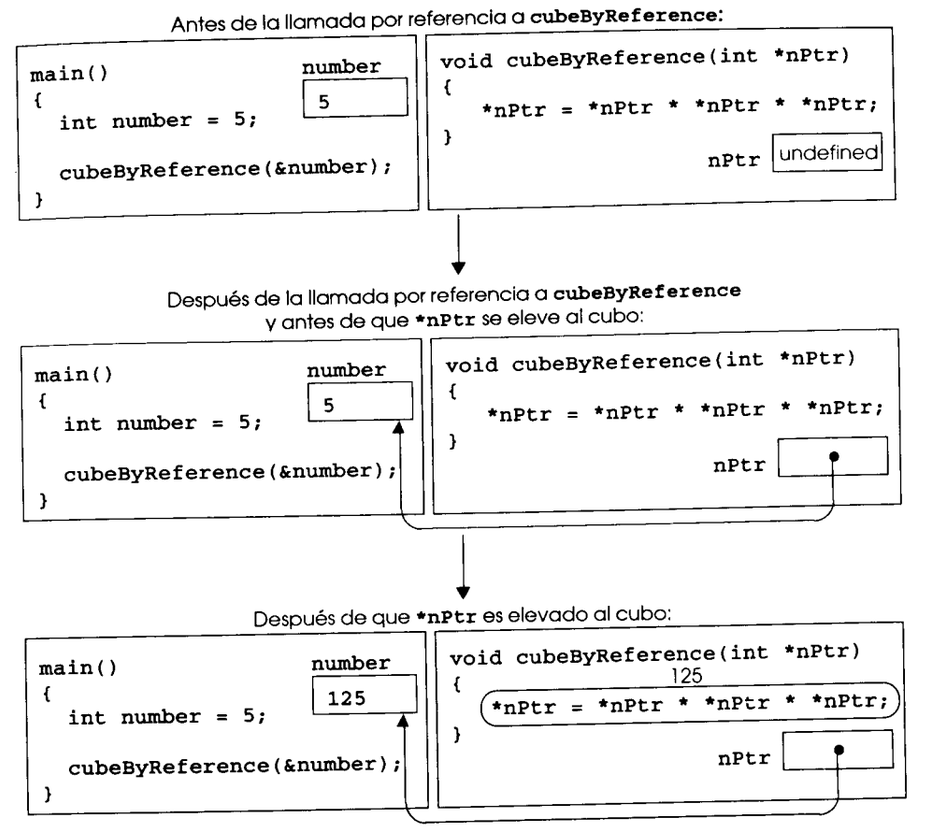


Fig. 7.9 Análisis de una llamada por referencia típica.

```
void cubeByReference(int *nPtr)
```

El encabezado especifica que **cubeByReference** recibe la dirección de una variable entera como argumento, almacena la dirección local en **nPtr**, y no regresa un valor.

El prototipo de función para **cubeByReference** contiene **int *** entre paréntesis. Como en el caso de otros tipos de variables, en los prototipos de función no es necesario incluir los nombres de los apuntadores. Los nombres, que se incluyan para fines de documentación, serán ignorados por el compilador de C.

En el encabezado de función y en el prototipo para una función que espera un arreglo de un subíndice como argumento, puede ser utilizada la notación de apuntador en la lista de parámetros de **cubeByReference**. El compilador no diferencia entre una función que recibe un apuntador y una función que recibe un arreglo de un solo subíndice. Esto, naturalmente, significa que la función debe "saber" cuando está recibiendo un arreglo o una variable para la cual deberá llevar a cabo una llamada por referencia. Cuando el compilador encuentre un parámetro de función correspondiente a un arreglo de un subíndice de la forma **int b []**, el compilador convertirá el parámetro a la notación de apuntador **int *b**. Ambas formas son intercambiables.

Práctica sana de programación 7.3

Utilice llamadas por valor para pasar argumentos a una función, a menos de que en forma explícita el llamador requiera que la función llamada modifique el valor de la variable del argumento en el entorno de llamador. Esto es otro ejemplo del principio del mínimo privilegio.

7.5 Cómo utilizar el calificador Const con apuntadores

El *calificador* **const** permite al programador informarle al compilador que el valor de una variable particular no deberá ser modificado. En las primeras versiones de C no existía el calificador **const**; fue añadido al lenguaje por el comité de ANSI C.

Observación de ingeniería de software 7.1

El calificador **const** *puede ser utilizado para forzar el principio del mínimo privilegio. La utilización del principio de mínimo privilegio para diseñar con propiedad el software reduce en forma importante el tiempo de depuración y efectos inadecuados colaterales, y hace un programa más fácil de modificar y mantener.*

Tip de portabilidad 7.1

A pesar de que en ANSI C **const** *está bien definido, algunos sistemas no lo tienen incorporado.*

A través de los años, una gran base de código heredado quedó escrita en las primeras versiones de C, que no utilizan a **const** porque no estaba disponible. Por esta razón, existen grandes oportunidades de mejoría en la ingeniería de software del código existente de C. También, muchos programadores, que en la actualidad utilizan ANSI C, en sus programas no utilizan **const**, porque empezaron a programar usando las primeras versiones de C. Estos programadores están perdiendo muchas oportunidades de buena ingeniería de software.

Existen seis posibilidades para el uso (o el no uso) de **const** con parámetros de función, dos —al pasar parámetros en llamadas por valor y cuatro al pasar parámetros con llamadas por referencia. ¿Cómo escogerá usted una de las seis posibilidades? Deje que el principio del mínimo privilegio sea su guía. Siempre dele a una función suficiente acceso a los datos en sus parámetros para llevar a cabo su tarea especificada, pero no más.

En el capítulo 5, explicamos que todas las llamadas en C son llamadas por valor, en la llamada de función se efectúa una copia del argumento y se pasa a la función. Si en la función la copia se modifica, el valor original en el llamador se mantiene sin cambio. En muchos casos, el valor pasado a la función es modificado para que ésta pueda llevar a cabo su tarea. Sin embargo, en algunas ocasiones, el valor no deberá ser alterado en la función llamada, aun cuando la función llamada manipule una copia del valor original.

Considere una función que toma como argumentos un arreglo de un subíndice y su tamaño e imprime el arreglo. Una función como ésta deberá ciclar a través del arreglo y extraer en forma individual cada elemento del arreglo. El tamaño del arreglo se utiliza en el cuerpo de la función para determinar el subíndice alto del mismo, de tal forma que el ciclo pueda terminar cuándo se termine la impresión. El tamaño del arreglo no se modifica en el cuerpo de la función.

Observación de ingeniería de software 7.2

Si un valor no se modifica (o no debería modificarse) en el cuerpo de una función al cual es pasado, el valor deberá declararse **const**, *para asegurarse que no se modifica de forma accidental.*

Si se hace un intento de modificar un valor declarado **const**, el compilador lo detectará y emitirá ya sea una advertencia, o un error, dependiendo del compilador particular.

Observación de ingeniería de software 7.3

En una función llamadora sólo un valor puede ser alterado cuando se utiliza llamada por valor. Este valor debe ser asignado a partir del valor de regreso de la función. Para modificar varios valores en una función llamadora, debe utilizarse llamada por referencia.

Práctica sana de programación 7.4

Antes de utilizar una función, verifique el prototipo de función correspondiente a esta función, a fin de determinar si la función es capaz de modificar los valores que se le pasan.

Error común de programación 7.4

No estar consciente que una función está esperando apuntadores como argumentos por llamadas por referencia, y está pasando argumentos en llamadas por valor. Algunos compiladores toman los valores, suponiendo que son apuntadores y desreferencian los valores como apuntadores. En tiempo de ejecución, a menudo se generan violaciones de acceso a la memoria o fallas de segmentación. Otros compiladores detectan falta de coincidencia en tipo entre argumentos y parámetros, y generan mensajes de error.

Existen cuatro formas para pasar un apuntador a una función: un apuntador no constante a datos no constantes, un apuntador constante a datos no constantes, un apuntador no constante a datos constantes, y un apuntador constante a datos constantes. Cada una de las cuatro combinaciones proporciona un nivel distinto de privilegios de acceso.

El nivel más alto de acceso de datos se consigue mediante un apuntador no constante a datos no constantes. En este caso, los datos pueden ser modificados a través de un apuntador desreferenciado, y el apuntador puede ser modificado para señalar a otros elementos de datos. Una declaración para un apuntador no constante a datos no constantes no incluye **const**. Tal apuntador pudiera ser utilizado para recibir una cadena como argumento a una función que utiliza aritmética de apuntador para procesar (y posiblemente para modificar) cada carácter dentro de la cadena. La función **convertToUppercase** de la figura 7.10 declara como su argumento un apuntador no constante a datos no constantes, llamado **s (char*s)**. La función procesa la cadena **s**, un carácter a la vez, utilizando aritmética de apuntador. Si un carácter está en el rango **a** a **z**, se convierte a su letra en mayúsculas correspondiente, **A** a **Z** utilizando un cálculo basado en su código ASCII; de lo contrario es pasado por alto, y es procesado el siguiente carácter en la cadena. Note que todas las letras mayúsculas en el conjunto de caracteres ASCII tienen valores enteros que son equivalentes en valores ASCII a sus letras correspondientes en minúsculas menos 32 (vea la tabla de valores de caracteres ASCII en el Apéndice D). En el capítulo 8, presentaremos la función **toupper** de la biblioteca estándar de C para la conversión de letras a mayúsculas.

Un apuntador no constante a datos constantes es un apuntador que puede ser modificado para apuntar a cualquier elemento de datos del tipo apropiado, pero no pueden ser modificados los datos hacia los cuales apunta. Tal apuntador pudiera ser utilizado para recibir un argumento de arreglo a una función, que procesaría cada elemento del arreglo, sin modificar los datos. Por ejemplo, la función **printCharacters** de la figura 7.11 declara los parámetros s del tipo **const char***. La declaración se lee de derecha a izquierda de la forma "**s** es un apuntador a una constante de carácter". El cuerpo de la función utiliza una estructura **for**, para extraer cada carácter de la cadena, hasta que encuentre el carácter NULL. Después de haber impreso cada carácter, el apuntador s es incrementado, para que apunte al siguiente carácter dentro de la cadena.

```
/* Converting lowercase letters to uppercase letters */
/* using a non-constant pointer to non-constant data */
#include <stdio.h>

void convertToUppercase(char *);

main()
{
   char string[] = "characters";

   printf("The string before conversion is: %s\n", string);
   convertToUppercase(string);
   printf("The string after conversion is:  %s\n", string);
   return 0;
}

void convertToUppercase(char *s)
{
   while (*s != '\0') {

      if (*s >= 'a' && *s <= 'z')
         *s -= 32;  /* convert to ASCII uppercase letter */

      ++s;  /* increment s to point to the next character */
   }
}
```

```
The string before conversion is: characters
The string after conversion is: CHARACTERS
```

Fig. 7.10 Cómo convertir una cadena a mayúsculas, utilizando un apuntador no constante a datos no constantes.

En la figura 7.12 se demuestran los mensajes de error producidos por el compilador Borland C++ al intentar compilar una función que recibe un apuntador no constante a datos constantes, y la función utiliza el apuntador a fin de modificar datos.

Como sabemos, los arreglos son conjuntos de tipos de datos, que almacenan bajo un nombre muchos elementos de datos relacionados del mismo tipo. En el capítulo 10, analizaremos otra forma de tipo de conjunto de datos llamada una *estructura* (llamado a veces en otros lenguajes un *registro*). Una estructura es capaz de almacenar muchos elementos de datos relacionados, de distintos tipos de datos, bajo un nombre (por ejemplo, almacenar información sobre cada uno de los empleados de una empresa). Cuando se llama una función con un arreglo como argumento, el arreglo se pasa de forma automática a la función en llamada por referencia. Sin embargo, las estructuras son siempre pasadas en llamada por valor —se pasa una copia de toda la estructura. Esto requiere de sobrecarga en tiempo de ejecución, para efectuar una copia de cada elemento de dato en la estructura y almacenarla en la pila de llamada de la función, de la computadora. Cuando los datos de la estructura deban ser pasados a una función, podemos utilizar apuntadores hacia datos constantes, para conseguir el rendimiento de una llamada por referencia y la protección de una llamada por valor. Cuando se pasa un apuntador a una estructura, sólo debe de ser efectuada una copia de la dirección en donde está almacenada la estructura. En una máquina con direcciones de 4 bytes, se efectúa una copia de 4 bytes de memoria, en vez de una copia de la estructura, de posiblemente cientos o miles de bytes.


```
/* Printing a string one character at a time using */
/* a non-constant pointer to constant data         */
#include <stdio.h>

void printCharacters(const char *);

main()
{
   char string[] = "print characters of a string";

   printf("The string is:\n");
   printCharacters(string);
   putchar('\n');
   return 0;
}

void printCharacters(const char *s)
{
   for ( ; *s != '\0'; s++)   /* no initialization */
      putchar(*s);
}
```

```
The string is:
print characters of a string
```

Fig. 7.11 Cómo imprimir una cadena, un carácter a la vez, utilizando un apuntador no constante a datos constantes.

Sugrencia de rendimiento 7.1

Pase grandes objetos como son estructuras utilizando apuntadores a datos constantes para obtener los beneficios de rendimiento de llamadas por referencia y la seguridad de llamadas por valor.

Utilizar de esta forma apuntadores a datos constantes es un ejemplo de *cambiar tiempo por espacio*. Si la memoria es reducida y la eficiencia de ejecución es una preocupación importante, deberán utilizarse apuntadores. Si la memoria es abundante y la eficiencia no es una preocupación importante, los datos deberán ser pasados en llamada por valor, para forzar el principio del mínimo privilegio. Recuerde que algunos sistemas no manejan bien **const**, por lo que la llamada por valor sigue siendo la mejor forma de evitar que los datos sean modificados.

Un apuntador constante a datos no constantes es un apuntador que siempre apunta a la misma posición de memoria, y los datos en esa posición pueden ser modificados a través del apuntador. Esta es la forma por omisión de un nombre de arreglo. Un nombre de arreglo es un apuntador constante al inicio de dicho arreglo. Todos los datos en el arreglo son accesibles y modificados, utilizando el nombre del arreglo y los subíndices del mismo. Un apuntador constante a datos no constantes puede ser utilizado para recibir un arreglo como argumento de una función, que tiene acceso a elementos de arreglo utilizando sólo notaciones de subíndices del arreglo. Los apuntadores declarados **const** deben ser inicializados al ser declarados (si el apuntador es un parámetro

```
/* Attempting to modify data through a    */
/* non-constant pointer to constant data */
#include <stdio.h>

void f(const int *);

main()
{
   int y;

   f(&y);      /* f attempts illegal modification */
   return 0;
}

void f(const int *x)
{
   *x = 100;  /* cannot modify a const object */
}
```

```
Compiling FIG7_12.C:
Error FIG7_12.C 17: Cannot modify a const object
Warning FIG7_12.C 18: Parameter 'x' is never used
```

Fig. 7.12 Intento de modificación de datos a través de un apuntador no constante a datos constantes.

de función, será inicializado con un apuntador que se pasa a la función). El programa de la figura 7.13 intenta modificar un apuntador constante. El apuntador **ptr** se declara ser del tipo **int * const**. La declaración se lee de derecha a izquierda como "**ptr** es un apuntador constante a un entero". El apuntador se inicializa con la dirección de la variable entera **x**. El programa intenta asignar la dirección de **y** a **ptr**, pero se genera un mensaje de error.

El privilegio de mínimo acceso se concede mediante un apuntador constante a datos constantes. Un apuntador de este tipo siempre apunta a la misma posición de memoria, y los datos en esa posición de memoria no pueden ser modificados. Esta es la forma en que debería pasarse un arreglo a una función que sólo ve al arreglo utilizando notación de subíndices del mismo y no modifica dicho arreglo. El programa de la figura 7.14 declara a la variable de apuntador **ptr** ser del tipo **const int * const**. Esta declaración se lee de derecha a izquierda como "**ptr** es un apuntador constante a un entero constante". La figura muestra los mensajes de error generados cuando se intenta modificar los datos a los cuales apunta **ptr**, y cuando se intenta modificar la dirección almacenada en la variable del apuntador.

7.6 Ordenación de tipo burbuja utilizando llamadas por referencia

Modifiquemos el programa de ordenación de tipo burbuja de la figura 6.15 para utilizar dos funciones **bubbleSort** y **swap**. La función **bubbleSort** ejecuta la ordenación del arreglo. Llama a la función **swap** para intercambiar los elementos del arreglo **array[j]** y **array[j +1]** (vea la figura 7.15). Recuerde que C obliga al ocultamiento de la información entre funciones,


```
/* Attempting to modify a constant pointer to */
/* non-constant data                          */
#include <stdio.h>

main()
{
   int x, y;
   int * const ptr = &x;

   ptr = &y;
   return 0;
}
```

```
Compiling FIG7_13.C:
Error FIG7_13.C 10 Cannot modify a const object
Warning FIG7_13.C 12: 'ptr' is assigned a value that is
   never used
Warning FIG7_13.C 12: 'y' is declared but never used
```

Fig. 7.13 Intento de modificación de un apuntador constante a datos no constantes.

```
/* Attempting to modify a constant pointer to */
/* constant data                              */
#include <stdio.h>

main()
{
   int x = 5, y;
   const int *const ptr = &x;

   *ptr = 7;
   ptr = &y;
   return 0;
}
```

```
Compiling FIG7_14.C:
Error FIG7_14.C 10: Cannot modify a const object
Error FIG7_14.C 11 Cannot modify a const object
Warning FIG7_14.C 13: 'ptr' is assigned a value that is
   never used
Warning FIG7_14.C 13: 'y' is declared but never used
```

Fig. 7.14 Intento de modificación de un apuntador constante a datos constantes.

por lo que **swap** no tiene acceso a los elementos individuales del arreglo en **bubbleSort**. Dado que **bubbleSort** *desea* que **swap** tenga acceso a los elementos del arreglo a ser intercambiados, **bubbleSort** pasa cada uno de estos elementos por llamada por referencia a **swap** la dirección

de cada elemento del arreglo se pasa en forma explícita. Aunque arreglos completos de manera automática se pasan en llamada por referencia, los elementos individuales del arreglo son escalares y, por lo regular, son pasados en llamadas por valor. Por lo tanto, **bubbleSort** utiliza el operador de dirección (**&**) en cada uno de los elementos del arreglo de la llamada de **swap**, como sigue

```
swap(&array[j], &array[j + 1]);
```

para que ocurra la llamada por referencia. La función **swap** recibe **&array[j]** en la variable de apuntador **element1Ptr**. Aun cuando **swap** debido al ocultamiento de información no tiene permitido saber el nombre de **array[j]**, **swap** puede utilizar ***element1Ptr** como un sinónimo de **array[j]**. Por lo tanto, cuando **swap** hace referencia a ***element1Ptr**, de hecho está referenciando a **array[j]** de **bubbleSort**. Similarmente, cuando **swap** hace referencia a ***element2Ptr**, de hecho está referenciando a **array[j + 1]** de **bubbleSort**. Aun cuando **swap** no se le tiene permitido decir

```
temp = array[j];
array[j] = array[j + 1];
array[j + 1] = temp;
```

precisamente el mismo efecto se consigue mediante

```
temp = ;*element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = temp;
```

en la función **swap** de la figura 7.15.

Deben de hacerse notar ciertas características de la función **bubbleSort**. El encabezado de función declara **array** como **int *array** en vez de **int array[]**, para indicar que **bubbleSort** recibe como argumento un arreglo de un subíndice (otra vez, estas notaciones son intercambiables). El parámetro **size** se declara **const**, a fin de obligar al principio del mínimo privilegio. Aunque el parámetro **size** recibe una copia de un valor en **main**, y modificar dicha copia no puede cambiar el valor en **main**, para llevar a cabo su tarea **bubbleSort** no necesita modificar **size**. Durante la ejecución de **bubbleSort** el tamaño del arreglo se conserva fijo. Por lo tanto, **size** se declara **const**, para asegurarse de que no se modifique. Si durante el proceso de ordenación se llegara a modificar el tamaño del arreglo, sería posible que el algoritmo de ordenamiento no se ejecutase de forma correcta.

El prototipo para la función **swap** se incluye en el cuerpo de la función **bubbleSort**, porque es la única función que llama a **swap**. El colocar el prototipo en **bubbleSort** restringe llamadas directas a **swap** únicamente a las que **bubbleSort** ejecuta. Otras funciones que intenten llamar a **swap** no tienen acceso a un prototipo de función apropiada, por lo que el compilador genera en forma automática una. Esto por lo regular resulta un prototipo que no coincide con el encabezado de función (y genera un error de compilación), porque el compilador supone **int** para el tipo de regreso, y para los tipos de parámetros.

Observación de ingeniería de software 7.4

Colocar prototipos de función en las definiciones de otras funciones obliga al principio del mínimo privilegio al restringir las llamadas correctas de función sólo a aquellas funciones en las cuales dichos prototipos aparecen.


```
/* This program puts values into an array, sorts
   the values into ascending order, and prints the
   resulting array */
#include <stdio.h>
#define SIZE 10

void bubbleSort(int *, const int);

main()
{
   int i, a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};

   printf("Data items in original order\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%4d", a[i]);

   bubbleSort(a, SIZE);          /* sort the array */
   printf("\nData items in ascending order\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%4d", a[i]);

   printf("\n");
   return 0;
}

void bubbleSort(int *array, const int size)
{
   int pass, j;
   void swap(int *, int *);

   for (pass = 1; pass <= size - 1; pass++)

      for (j = 0; j <= size - 2; j++)

         if (array[j] > array[j + 1])
            swap(&array[j], &array[j + 1]);
}

void swap(int *element1Ptr, int *element2Ptr)
{
   int temp;

   temp = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = temp;
}
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Fig. 7.15 Ordenación de tipo burbuja con llamada por referencia.

Note que la función **bubbleSort** recibe como parámetro el tamaño del arreglo. La función debe saber el tamaño del arreglo, para poder clasificarlo. Cuando se pasa un arreglo a una función, la dirección en memoria del primer elemento del arreglo es recibido por la función. La dirección no proporciona ninguna información a la función en relación con el número de elementos del arreglo. Por lo tanto, el programador debe proporcionar a la función el tamaño de dicho arreglo.

En el programa, la función **bubbleSort** recibió de forma explícita el tamaño del arreglo. Existen dos ventajas principales en este enfoque —la reutilización del software y una adecuada ingeniería de software. Al definir la función de tal forma que reciba el tamaño del arreglo en forma de argumento, permitimos que se utilice la función en cualquier programa, que ordene arreglos enteros de un subíndice, así como que los arreglos puedan ser de cualquier tamaño.

Observación de ingeniería de software 7.5

Al pasar un arreglo a una función, pase también su tamaño. Esto ayuda a generalizar la función. Las funciones generales son a menudo reutilizables.

Podríamos haber almacenado el tamaño del arreglo en una variable global, accesible a todo el programa. Esto hubiera sido más eficiente, porque una copia del tamaño no se hubiera hecho para pasársela a la función. Sin embargo, otros programas que requieren de una capacidad de clasificación de arreglos enteros, pudieran no tener la misma variable global y, por lo tanto, la función no podría ser utilizada en dichos programas.

Observación de ingeniería de software 7.6

Las variables globales violan el principio del mínimo privilegio y son un ejemplo de ingeniería de software pobre.

Sugerencia de rendimiento 7.2

Pasar el tamaño de un arreglo a una función ocupa tiempo y espacio de pilas adicional, porque debe ejecutarse una copia del tamaño para ser pasada a la función. Las variables globales, sin embargo, no requieren de tiempo adicional o de espacio, porque son accesibles directamente por cualquier función.

El tamaño del arreglo pudiera haberse programado en directo dentro de la función. Esto restringe el uso de la función, a un arreglo de un tamaño específico, y reduce en forma importante su reutilización. Sólo programas que procesen arreglos enteros de un subíndice, del tamaño específico codificado dentro de la función, podrían utilizar esta función.

C proporciona el *operador unario* especial **sizeof** para determinar el tamaño de un arreglo en bytes (o de cualquier otro tipo de datos) durante la compilación de un programa. Cuando se aplica al nombre de un arreglo, como en la figura 7.16, el operador **sizeof** regresa como un entero el número total de bytes del arreglo. Note que las variables del tipo **float** están almacenadas por lo regular en 4 bytes de memoria, y **array** está declarado que tiene 20 elementos. Por lo tanto, existe en **array** un total de 80 bytes.

El número de elementos de un arreglo también puede ser determinado en tiempo de compilación. Por ejemplo, considere la siguiente declaración de arreglo:

```
double real[22];
```



```
/* sizeof operator when used on an array name */
/* returns the number of bytes in the array   */
#include <stdio.h>

main()
{
   float array[20];

   printf("The number of bytes in the array is %d\n",
          sizeof(array));

   return 0;
}
```

```
The number of bytes in the array is 80
```

Fig. 7.16 El operador **sizeof** cuando se aplica a un nombre de arreglo, regresa el número de bytes en el mismo.

Las variables **double** están normalmente almacenadas en 8 bytes de memoria. Por lo tanto, el arreglo **real** contiene un total de 176 bytes. Para determinar el número de elementos en el arreglo, puede utilizarse la siguiente expresión:

```
sizeof(real) / sizeof(double)
```

La expresión determina el número de bytes en el arreglo **real**, y divide dicho valor por el número de bytes utilizados en memoria para almacenar un valor **double**.

El programa de la figura 7.17 calcula el número de bytes utilizados para almacenar cada uno de los tipos de datos estándar, en una PC compatible.

Sugerencia de portabilidad 7.2

El número de bytes utilizados para almacenar un tipo particular de datos, pudiera variar de sistema a sistema. Al escribir programas que dependan de tamaños de tipos de datos, y que se ejecutarán en diversos sistemas de computación, utilice **sizeof** *para determinar el número de bytes utilizados para almacenar los tipos de datos.*

El operador **sizeof** puede ser aplicado a cualquier nombre de variable, tipo o constante. Al ser aplicado a un nombre de variable (que no sea un nombre de arreglo),o a una constante, será regresado el número de bytes utilizados para almacenar el tipo específico de variable o de constante. Note que son requeridos los paréntesis utilizados junto con **sizeof**, si el nombre del tipo se proporciona como su operandó. Si se omiten los paréntesis ocurrirá un error de sintaxis. No son requeridos los paréntesis si como su operando se proporciona el nombre de la variable.

7.7 Expresiones y aritmética de apuntadores

Los apuntadores son operandos válidos en expresiones aritméticas, en expresiones de asignación y en expresiones de comparación. Sin embargo, no todos los operadores, normalmente utilizados en estas expresiones son válidos, en conjunción con las variables de apuntador. En esta sección se describen los operadores que pueden tener apuntadores como operandos, y como se utilizan dichos operadores.

```
/* Demonstrating the sizeof operator */
#include <stdio.h>

main()
{
   printf("        sizeof(char) = %d\n"
          "       sizeof(short) = %d\n"
          "         sizeof(int) = %d\n"
          "        sizeof(long) = %d\n"
          "       sizeof(float) = %d\n"
          "      sizeof(double) = %d\n"
          "sizeof(long double) = %d\n",
          sizeof(char), sizeof(short), sizeof(int),
          sizeof(long), sizeof(float), sizeof(double),
          sizeof(long double));
   return 0;
}
```

```
sizeof(char)        = 1
sizeof(short)       = 2
sizeof(int)         = 2
sizeof(long)        = 4
sizeof(float)       = 4
sizeof(double)      = 8
sizeof(long double) = 10
```

Fig. 7.17 Cómo utilizar el operador **sizeof** para determinar los tamaños de tipo de datos estándar.

Con apuntadores se pueden ejecutar un conjunto limitado de operaciones aritméticas. Un apuntador puede ser incrementado (++) o decrementado (—), se puede añadir un entero a un apuntador (+ o +=), un entero puede ser restado de un apuntador (- o -=), o un apuntador puede ser sustraído o restado de otro.

Suponga que ha sido declarado el arreglo **int v[10]** y su primer elemento está en memoria en la posición **3000**. Suponga que el apuntador **vPtr** ha sido inicializado para apuntar a **v[0]**, es decir, el valor de **vPtr** es **3000**. En la figura 7.18 se diagrama esta situación para una máquina con enteros de 4 bytes. Note que **vPtr** puede ser inicializado para apuntar al arreglo **v** con cualquiera de los enunciados

```
vPtr = v;
vPtr = &v[0];
```

Sugerencia de portabilidad 7.3

La mayor parte de las computadoras de hoy día tienen enteros de 2 o de 4 bytes. Algunas de las máquinas más modernas utilizan enteros de 8 bytes. Dado que los resultados de la aritmética de apuntadores depende del tamaño de los objetos a los cuales el apuntador señala, la aritmética de los apuntadores depende de la máquina.

En aritmética convencional, la adición **3000 + 2** da como resultado el valor **3002**. Por lo regular, este no es el caso en la aritmética de apuntadores. Cuando se añade o se resta un entero de un apuntador, el apuntador no se incrementa o decrementa sólo por el valor de dicho entero,

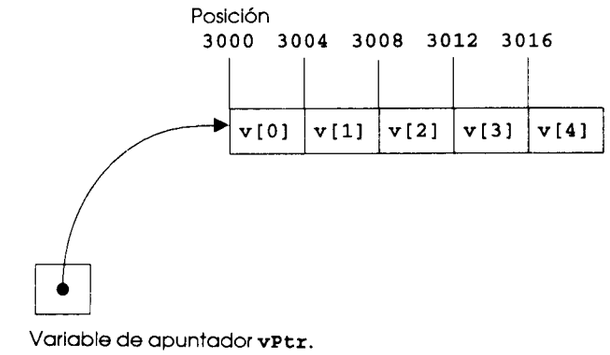


Fig. 7.18 El arreglo **v** y una variable de apuntador **vPtr**, que señala a **v**.

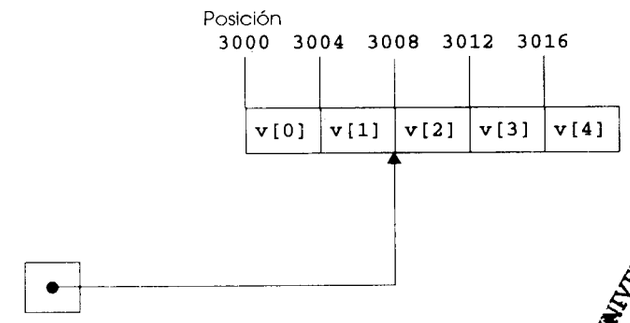
sino por el entero, multiplicado por el tamaño del objeto al cual el apuntador se refiere. El número de bytes depende del tipo de datos del objeto. Por ejemplo, el enunciado

```
vPtr += 2;
```

produciría **3008** (**3000 + 2 * 4**), suponiendo un entero almacenado en 4 bytes de memoria. En el arreglo **v**, **vPtr**, ahora señalaría a **v[2]** (figura 7.19). Si un entero se almacena en 2 bytes de memoria, entonces el cálculo anterior resultaría en una posición de memoria **3004** (**3000 + 2 * 2**). Si el arreglo fuera de un tipo de datos diferente, el enunciado precedente incrementaría el apuntador por dos veces el número de bytes que toma para almacenar un objeto de dicho tipo de datos. Al ejecutar aritmética de apuntadores en un arreglo de caracteres, los resultados serán consistentes con la aritmética normal, porque cada carácter tiene una longitud de un byte.

Si **vPtr** ha sido incrementado a **3016**, lo que señala a **v[4]**, el enunciado

```
vPtr - = 4;
```



UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE
DOCUMENTACIÓN Y BIBLIOTE
MONTEVIDEO - URUGU

definiría a **vPtr** de vuelta a **3000** lo que es decir al principio del arreglo. Si un apuntador está siendo incrementado o decrementado por uno, pueden ser utilizados los operadores de incremento (++)y de decremento (–). Cualquiera de los enunciados

```
++vPtr;
vPtr++;
```

incrementan el apuntador, para que apunte a la siguiente posición dentro del arreglo. Cualquiera de los enunciados

```
--vPtr;
vPtr--;
```

decrementan el apuntador, para que apunte al elemento anterior del arreglo.

Las variables de apuntador pueden ser restadas una de otra. Por ejemplo, si **vPtr** contiene la posición **3000**, y **v2Ptr** contiene la dirección **3008**, el enunciado

```
x = v2Ptr - vPtr;
```

asignaría a **x** el número de los elementos del arreglo de **vPtr** hasta **v2Ptr**, en este caso, **2**. La aritmética de apuntadores no tiene significado, a menos de que se ejecute en un arreglo. No podemos suponer que dos variables del mismo tipo estén almacenados de manera contigua en memoria, a menos de que sean elementos adyacentes de un arreglo.

Error común de programación 7.5

Utilizar aritmética de apuntadores en un apuntador que no se refiere a un arreglo de valores.

Error común de programación 7.6

Restar o comparar dos apuntadores que no se refieren al mismo arreglo.

Error común de programación 7.7

Salirse de cualquiera de los dos extremos de un arreglo al utilizar aritmética de apuntadores.

Un apuntador puede ser asignado a otro apuntador, si ambos son del mismo tipo. De lo contrario, deberá utilizarse un operador cast para convertir el apuntador a la derecha de la asignación al tipo de apuntador de la izquierda de la asignación. La excepción a esta regla es el apuntador a **void** (es decir, **void ***) que es un apuntador genérico, que puede representar cualquier tipo de apuntador. Todos los tipos de apuntador pueden ser asignados a un apuntador **void** y un apuntador **void** puede ser asignado a un apuntador de cualquier tipo. En ambos casos no se requiere de una operación cast.

Un apuntador a **void** no puede ser desreferenciado. Por ejemplo, el compilador sabe que un apuntador a **int** se refiere a 4 bytes, en memoria en una máquina con enteros de 4 bytes, pero un apuntador a **void** sólo contiene una posición de memoria para un tipo de datos desconocido —el número preciso de bytes a los que se refiere el apuntador no es conocido por el compilador. Para un apuntador en particular, el compilador debe saber el tipo de datos, para determinar el número de bytes a desreferenciar. En el caso de un apuntador **void**, el número de bytes no puede ser determinado partiendo del tipo.

Error común de programación 7.8

Se generará un error de sintaxis asignar un apuntador de un tipo a un apuntador de otro tipo, si ninguno de los dos es del tipo **void ***.

Error común de programación 7.9

Desreferenciar un apuntador **void ***.

Los apuntadores pueden ser comparados mediante operadores de igualdad y relacionales, pero dichas comparaciones no tendrán sentido, a menos que los apuntadores señalen a miembros del mismo arreglo. Las comparaciones de apuntadores comparan las direcciones almacenadas en los mismos. Una comparación de dos apuntadores que señalen al mismo arreglo podría mostrar, por ejemplo, que un apuntador señale a un elemento de numeración más alta en el arreglo que el otro. Un uso común de una comparación de apuntadores es determinar si un apuntador es **NULL**.

7.8 Relaciones entre apuntadores y arreglos

Los arreglos y los apuntadores en C están relacionados en forma íntima y pueden ser utilizados *casi* en forma indistinta. Un nombre de arreglo puede ser considerado como un apuntador constante. Los apuntadores pueden ser utilizados para hacer cualquier operación que involucre subíndices de arreglos.

Sugerencia de rendimiento 7.3

Durante la compilación la notación de subíndices de arreglo se convierte a notación de apuntador, por lo que escribir expresiones de subíndices de arreglo con notación de apuntadores, puede ahorrar tiempo de compilación.

Práctica sana de programación 7.5

Al manipular arreglos utilice notación de arreglos en vez de notación de apuntadores. Aunque el programa pudiera tomar más tiempo para su compilación, es probable que sea mucho más claro.

Suponga que han sido declarados el arreglo entero **b[5]** y la variable de apuntador entera **bPtr**. Dado que el nombre del arreglo (sin subíndice) es un apuntador al primer elemento del arreglo, podemos definir **bPtr** igual a la dirección del primer elemento en el arreglo **b**, mediante el enunciado

```
bPtr = b;
```

Este enunciado es equivalente a tomar la dirección del primer elemento del arreglo, como sigue

```
bPtr = &b[0];
```

Alternativamente el elemento del arreglo **b[3]** puede ser referenciado con la expresión de apuntador

```
*(bPtr + 3)
```

El **3** en la expresión arriba citada es el *desplazamiento* del apuntador. Cuando el apuntador apunta al principio de un arreglo, el desplazamiento indica qué elemento del arreglo debe ser referenciado, y el valor de desplazamiento es idéntico al subíndice del arreglo. La notación anterior se conoce como *notación apuntador/desplazamiento*. Son necesarios los paréntesis porque la precedencia de * es más alta que la de +. Sin los paréntesis, la expresión arriba citada sumaría **3** al valor de la expresión ***bPtr** (es decir, se añadiría **3** a **b[0]**, suponiendo que **bPtr** apunta al principio del arreglo). Al igual que el elemento del arreglo puede ser referenciado con una expresión de apuntador, la dirección

```
&b[3]
```

puede ser escrita con la expresión de apuntador

```
bPtr + 3
```

El arreglo mismo puede ser tratado como un apuntador, y utilizado en aritmética de apuntador. Por ejemplo, la expresión

```
*(b + 3)
```

también se refiere al elemento del arreglo `b[3]`. En general, todas las expresiones de arreglos con subíndice pueden ser escritas mediante un apuntador y un desplazamiento. En este caso se utilizó notación apuntador/desplazamiento junto con el nombre del arreglo como un apuntador. Note que el enunciado anterior no cambia de forma alguna el nombre del arreglo; `b` aún apunta al primer elemento del arreglo.

Los apuntadores pueden tener subíndices exactamente de la misma forma que los arreglos. Por ejemplo, la expresión

```
bPtr[1]
```

se refiere al elemento del arreglo `b[1]`. Esto se conoce como *notación apuntador/subíndice.*

Recuerde que el nombre de un arreglo es, en esencia, un apuntador constante; siempre apunta al principio del arreglo. Por lo tanto, la expresión

```
b += 3
```

resulta inválida, porque intenta modificar el valor del nombre de un arreglo con aritmética de apuntador.

Error común de programación 7.10

Es un error de sintaxis intentar modificar un nombre de arreglo con aritmética de apuntador.

El programa de la figura 7.20 utiliza los cuatro métodos analizados aquí para referirse a los elementos del arreglo —arreglos con subíndices, apuntador/desplazamiento con el nombre del arreglo como un apuntador, subíndice de apuntador, y apuntador/desplazamiento con un apuntador para imprimir los cuatro elementos del arreglo entero `b`.

Para ilustrar más aún la intercambiabilidad de arreglos y apuntadores, veamos las dos funciones de copia de cadenas —`copy1` y `copy2`— del programa de la figura 7.21. Ambas funciones copian una cadena (posiblemente un arreglo de caracteres) en un arreglo de caracteres. Después de una comparación de los prototipos de función de `copy1` y `copy2`, las funciones parecen idénticas. Llevan a cabo la misma tarea; sin embargo, se ponen en operación de forma distinta.

La función `copy1` utiliza notación de subíndices de arreglo para copiar la cadena en `s2` al arreglo de caracteres `s1`. La función declara una variable de contador entera `i`, para usarla como subíndice del arreglo. El encabezado de estructura `for` ejecuta toda la operación de copia —su cuerpo es el enunciado vacío. El encabezado especifica que `i` se inicializa a cero y se incrementa en uno en cada iteración del ciclo. La condición en la estructura `for`, `s1[i] = s2[i]`, ejecuta la operación de copia, carácter por carácter, desde `s2` a `s1`. Cuando se encuentra el carácter null en `s2`, se asigna a `s1`, y el ciclo termina porque el valor entero del carácter null es cero (falso). Recuerde que el valor de un enunciado de asignación es el valor asignado al argumento izquierdo.


```
/* Using subscripting and pointer notations with arrays */
#include <stdio.h>

main()
{
   int i, offset, b[] = {10, 20, 30, 40};
   int *bPtr = b;   /* set bPtr to point to array b */

   printf("Array b printed with:\n"
          "Array subscript notation\n");

   for (i = 0; i <= 3; i++)
      printf("b[%d] = %d\n", i, b[i]);

   printf("\nPointer/offset notation where \n"
          "the pointer is the array name\n");

   for (offset = 0; offset <= 3; offset++)
      printf("*(b + %d) = %d\n", offset, *(b + offset));

   printf("\nPointer subscript notation\n");

   for (i = 0; i <= 3; i++)
      printf("bPtr[%d] = %d\n", i, bPtr[i]);

   printf("\nPointer/offset notation\n");

   for (offset = 0; offset <= 3; offset++)
      printf("*(bPtr + %d) = %d\n", offset, *(bPtr + offset));

   return 0;
}
```

Fig. 7.20 Cómo usar los cuatro métodos de referenciar los elementos de arreglo (parte 1 de 2).

La función `copy2` utiliza apuntadores y aritmética de apuntadores para copiar la cadena de `s2` al arreglo de caracteres `s1`. Otra vez, el encabezado de estructura `for` ejecuta toda la operación de copia. El encabezado no incluye ninguna inicialización de variables. Como en la función `copy1`, la condición (`*s1 = *s2`) ejecuta la operación de copia. El apuntador `s2` se desreferencia y el carácter resultante se asigna al apuntador desreferenciado `s1`. Después de la asignación en la condición, los apuntadores se incrementan para señalar al siguiente elemento del arreglo `s1`, y el siguiente carácter de la cadena `s2`, respectivamente. Cuando en `s2` se encuentra el carácter null, se le asigna al apuntador desreferenciado `s1` y el ciclo se termina.

Note que el primer argumento, tanto de `copy1` como de `copy2` debe ser un arreglo lo suficiente grande para contener la cadena del segundo argumento. De lo contrario, podría ocurrir un error cuando se intente escribir en una posición de memoria que no forme parte del arreglo. También, note que el segundo parámetro de cada función se declara como `const char *` (una cadena constante). En ambas funciones, el segundo argumento se copia en el primer argumento —los caracteres se leen a partir de él uno por uno, pero los caracteres nunca se modifican. Por lo tanto, se declara el segundo parámetro para señalar a un valor constante, y que se cumpla el

```
Array b printed with:
Arry subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notion where
the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notion
*(bPtr + 9) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Fig. 7.20 Cómo usar los cuatro métodos de referenciar los elementos de arreglo (parte 2 de 2).

principio del mínimo privilegio. Ninguna de las funciones requiere de capacidad de modificar el segundo argumento, por lo que ninguna de ellas la tiene.

7.9 Arreglos de apuntadores

Los arreglos pueden contener apuntadores. Un uso común para una estructura de datos como ésta, es formar un arreglo de cadenas, conocida como un *arreglo de cadenas*. Cada entrada en el arreglo es una cadena, pero en C una cadena es esencial un apuntador a su primer carácter. Por lo que en un arreglo de cadenas cada entrada es de hecho un apuntador al primer carácter de una cadena. Veamos la declaración del arreglo de cadenas **suit**, que pudiera ser útil para representar un mazo de naipes.

```
char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};
```

La porción **suit[4]** de la declaración indica un arreglo de 4 elementos. La porción **char*** de la declaración indica que cada elemento del arreglo **suit** es del tipo "apuntador a **char**". Los cuatro valores a colocarse en el arreglo son **"Hearts"**, **"Diamonds"**, **"Clubs"** y **"Spades"**. Cada una de estas está almacenada en memoria como una cadena de caracteres terminada por NULL, de una longitud de un carácter más largo que el número de caracteres entre las comillas. Las cuatro cadenas son de 7, 9, 6 y 7 caracteres de longitud, respectivamente. Aunque pareciera como si estas cadenas están colocadas en el arreglo **suit**, de hecho en el arreglo sólo están almacenados los apuntadores (figura 7.22).


```
/* Copying a string using array notation
   and pointer notation */
#include <stdio.h>

void copy1(char *, const char *);
void copy2(char *, const char *);

main()
{
   char string1[10], *string2 = "Hello",
        string3[10], string4[] = "Good Bye";

   copy1(string1, string2);
   printf("string1 = %s\n", string1);

   copy2(string3, string4);
   printf("string3 = %s\n", string3);
   return 0;
}

/* copy s2 to s1 using array notation */
void copy1(char *s1, const char *s2)
{
   int i;

   for (i = 0; s1[i] = s2[i]; i++)
      ;   /* do nothing in body */
}

/* copy s2 to s1 using pointer notation */
void copy2(char *s1, const char *s2)
{
   for ( ; *s1 = *s2; s1++, s2++)
      ;   /* do nothing in body */
}
```

```
string1 = Hello
string3 = Good Bye
```

Fig. 7.21 Como copiar una cadena utilizando notación de arreglo y notación de apuntador.

Cada apuntador señala al primer carácter de su cadena correspondiente. Por lo tanto, aunque el arreglo **suit** es de tamaño fijo, permite el acceso a cadenas de caracteres de cualquier longitud. Esta flexibilidad es un ejemplo de las capacidades poderosas de estructuración de datos de C.

Los palos de la baraja podrían haber sido colocados en un arreglo doble, en el cual cada renglón podría representar un palo, y cada columna representaría una de las letras del nombre de dicho palo. Una estructura de datos como ésta tendría que tener un número fijo de columnas por renglón, y dicho número debería ser igual de grande que la cadena más larga. Por lo tanto, se desperdiciaría gran cantidad de memoria, cuando se tuviera que almacenar un gran número de

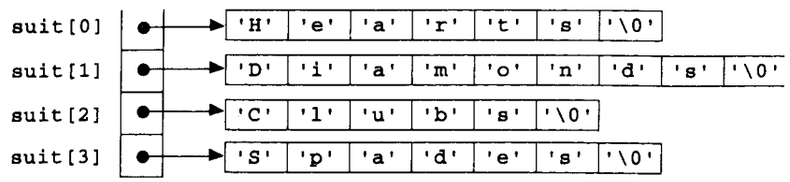


Fig. 7.22 Un ejemplo gráfico del arreglo **suit**.

cadenas, con la mayoría de ellas más cortas que la más larga. En la siguiente sección usamos arreglos de cadenas para representar un mazo de naipes.

7.10 Estudio de caso: simulación de barajar y repartir naipes

En esta sección, utilizamos la generación de números aleatorios para desarrollar un programa de simulación de barajar y repartir naipes. Este programa podrá ser después utilizado como base para formar programas, que jueguen juegos específicos de naipes. A fin de revelar algunos problemas sutiles de rendimiento, con intención hemos utilizado algoritmos subóptimos de barajar y de repartir. En los ejercicios y en el capítulo 10, desarrollaremos algoritmos más eficaces.

Utilizando el enfoque de refinación descendente, paso a paso, desarrollamos un programa que barajará un mazo de 52 naipes, y a continuación repartirá cada uno de dichos 52 naipes. El enfoque descendente es en particular útil para atacar problemas más grandes y más complejos que los que hemos visto en capítulos anteriores.

Utilizamos un arreglo de doble subíndice de 4 por 13 de nombre **deck** para representar el mazo de naipes (figura 7.23). Los renglones corresponden a los palos —el renglón 0 corresponde a los corazones, el renglón 1 a los diamantes, el renglón 2 a los tréboles y el renglón 3 a las espadas. Las columnas corresponden a los valores nominales de los naipes —las columnas del 0 al 9 corresponden a los valores del as al diez respectivamente, y las columnas 10 a la 12 corresponden al jack, reina y rey. Cargaremos el arreglo de cadenas **suit**, con cadenas de caracteres que representen los cuatro palos, y el arreglo de cadenas **face**, con cadenas de caracteres que representen los trece valores nominales.

Este mazo simulado de naipes puede ser barajado como sigue. Primero el arreglo **deck** se iguala a cero. A continuación, se selecciona aleatoriamente un **renglón** (0-3) y una **columna** (0-12). El número 1 se inserta en el elemento del arreglo **deck[row][column]** para indicar que este naipe será el primero que se repartirá del mazo barajado. Este proceso continúa con los números 2, 3, ..., 52 que se insertarán al azar en el arreglo **deck**, para indicar cuáles son los naipes que se le colocarán segundo, tercero, ..., y cincuenta y dos en el mazo barajado. Conforme el arreglo **deck** se empiece a llenar con números de naipes, es posible que un naipe quede seleccionado dos veces, es decir **deck[row][column]** al ser seleccionado resulte en no cero. Esta selección se ignora simplemente y se vuelve a repetir la selección de otros **rows** y **columns** en forma aleatoria, hasta que se encuentre un naipe no seleccionado. Eventualmente los números 1 hasta el 52 ocuparán los 52 renglones del arreglo **deck**. Llegado a este punto, el mazo de naipes ha sido totalmente barajado.

Si los naipes que ya han sido barajados se seleccionasen repetidamente al azar, este algoritmo de barajar se ejecutaría en forma indefinida. Este fenómeno se conoce como *posposición indefinida*. En los ejercicios analizaremos un mejor algoritmo de barajar, que elimina la posibilidad de posposición indefinida.

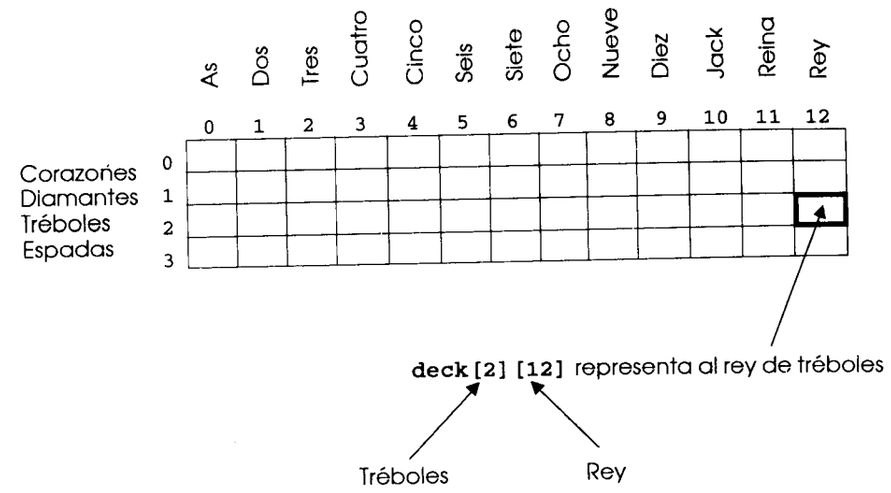


Fig. 7.23 Representación de arreglo de doble subíndice de un mazo de naipes.

Sugerencia de rendimiento 7.4

Algunas veces un algoritmo que aparece de forma "natural" puede contener problemas sutiles de rendimiento, como es la posposición indefinida. Busque algoritmos que eviten la posposición indefinida.

Para repartir el primer naipe, buscaremos en el arreglo a **deck[row][column] = 1**. Esto se lleva a cabo con una estructura anidada **for**, que varía **row** desde 0 hasta 3 y a **column** desde 0 hasta 12. ¿A qué naipe corresponde esta posición dentro del arreglo? El arreglo **suit** ha sido precargado con los cuatro palos, por lo que para obtener el palo, imprimimos la cadena de caracteres **suit[row]**. Similarmente, para obtener el valor nominal del naipe, imprimimos la cadena de caracteres **face[column]**. También imprimimos la cadena de caracteres **" of "**. Al imprimir esta información en el orden adecuado, nos permite imprimir cada naipe de la forma **"King of Clubs"**, **"Ace of Diamonds"**, y así en lo sucesivo.

Sigamos con el proceso de refinación descendente paso a paso. Lo general es simplemente

Shuffle and deal 52 cards

Nuestro primer refinamiento da como resultado:

Initialize the suit array
Initialize the face array
Initialize the deck array
Shuffle the deck
Deal 52 cards

"Shuffle the deck" pudiera expandirse, como sigue:

For each of the 52 cards
  Place card number in randomly selected unoccupied slot of deck

"Deal 52 cards" puede expandirse como sigue:

For each of the 52 cards
Find card number in deck array print face and suit of card

La incorporación de estas expansiones da como resultado nuestro segundo refinamiento completo:

Initialize the suit array
Initialize the face array
Initialize the deck array
For each of the 52 cards
Place card number in randomly selected unoccupied slot of deck

For each of the 52 cards
Find card number in deck array and print face and suit of card

"Place card number in randomly selected unoccupied slot of deck" se puede expandir como sigue:

Choose slot of deck randomly
While chosen slot of deck has been previously chosen
Choose slot of deck randomly
Place card number in chosen slot of deck

"Find card number in deck array and print face and suit of card" puede ser expandido como sigue:

For each slot of the deck array
If slot contains card number
Print the face and suit of the card

Al incorporar estas expansiones da como resultado nuestro tercer refinamiento:

Initialize the suit array
Initialize the face array
Initialize the deck array

For each of the 52 cards
Choose slot of deck randomly
While slot of deck has been previously chosen
Choose slot of deck randomly
Place card number in chosen slot of deck

For each of the 52 cards
For each slot of deck array
If slot contains desired card number
Print the face and suit of the card

Esto completa el proceso de refinamiento. Note que este programa es más eficiente si se combinan las porciones de barajar y de repartir del algoritmo, de tal forma que cada naipe sea repartido conforme se coloca en el mazo. Hemos decidido programar estas operaciones por separado, ya que por lo regular los naipes se reparten después de que han sido barajados (y no mientras se barajan).

El programa de barajar y repartir naipes se muestra en la figura 7.24 y una ejecución de muestra aparece en la figura 7.25. Note en las llamadas a **printf** el uso del especificador de conversión **%s** para imprimir cadenas de caracteres. El argumento correspondiente en la llamada **printf** debe ser un apuntador a **char** (o un arreglo **char**). En la función **deal**, la especificación de formato **"%5s of%-8s"** imprime una cadena de caracteres, justificado a la derecha, en un campo de cinco caracteres, seguido por **" of "** y una cadena de caracteres, justificado a la izquierda, en un campo de ocho caracteres. El signo menos en **%-8s** significa que la cadena está justificada a la izquierda, en un campo de ancho 8.

```
/* Card dealing program */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void shuffle(int [][13]);
void deal(const int [][13], const char *[], const char *[]);

main()
{
   char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};
   char *face[13] = {"Ace", "Deuce", "Three", "Four",
                     "Five", "Six", "Seven", "Eight",
                     "Nine", "Ten", "Jack", "Queen", "King"};
   int deck[4][13] = {0};

   srand(time(NULL));

   shuffle(deck);
   deal(deck, face, suit);

   return 0;
}

void shuffle(int wDeck[][13])
{
   int card, row, column;

   for(card = 1; card <= 52; card++) {
      row = rand() % 4;
      column = rand() % 13;

      while(wDeck[row][column] != 0) {
         row = rand() % 4;
         column = rand() % 13;
      }

      wDeck[row][column] = card;
   }
}
```

Fig. 7.24 Programa de repartición de naipes (parte 1 de 2).

```
void deal(const int wDeck[][13], const char *wFace[],
          const char *wSuit[])
{
   int card, row, column;

   for (card = 1; card <= 52; card++)

      for (row = 0; row <= 3; row++)

         for (column = 0; column <= 12; column++)

            if (wDeck[row][column] == card)
               printf("%5s of %-8s%c",
                      wFace[column], wSuit[row],
                      card % 2 == 0 ? '\n' : '\t');
}
```

Fig. 7.24 Programa de repartición de naipes (parte 2 de 2).

```
  Six of Clubs        Seven of Dimonds
  Ace of Spades         Ace of Diamonds
  Ace of Hearts       Queen of Diamonds
Queen of Clubs        Seven of Hearts
  Ten of Hearts       Deuce of Clubs
  Ten of Spades       Three of Spades
  Ten of Diamonds      Four of Spades
 Four of Diamonds       Ten of Clubs
  Six of Diamonds       Six of Spades
Eight of Hearts       Three of Diamonds
 Nine of Hearts       Three of Hearts
Deuce of Spades         Six of Hearts
 Five of Clubs        Eight of Clubs
Deuce of Diamonds     Eight of Spades
 Five of Spades        King of Clubs
 King of Diamonds      Jack of Spades
Deuce of Hearts       Queen of Hearts
  Ace of Clubs         King of Spades
Three of Clubs         King of Hearts
 Nine of Clubs         Nine of Spades
 Four of Hearts       Queen of Spades
Eight of Diamonds      Nine of Diamonds
 Jack of Diamonds     Seven of Clubs
 Five of Hearts        Five of Diamonds
 Four of Clubs         Jack of Hearts
 Jack of Clubs        Seven of Spades
```

Fig. 7.25 Ejecución de muestra del programa de repartición de naipes.

Existe una debilidad en el algoritmo de distribución. Una vez que se encuentra una coincidencia, aún si se encuentra en el primer intento, las dos estructuras internas **for** continúan su búsqueda de los elementos restantes de **deck**, buscando coincidencia. En los ejercicios y en el estudio de caso del capítulo 10, corregiremos esta deficiencia.

7.11 Apuntadores a funciones

Un apuntador a una función contiene la dirección de la función en memoria. En el capítulo 6, vimos que el nombre de un arreglo es en realidad la dirección de memoria del primer elemento de dicho arreglo. Similarmente, el nombre de una función es realmente la dirección inicial en memoria del código que ejecuta la tarea de dicha función. Los apuntadores a las funciones pueden ser pasados a las funciones, regresado de las funciones, almacenados en arreglos, y asignados a otros apuntadores de función.

Para ilustrar el uso de apuntadores a funciones, hemos modificado el programa de clasificación de tipo burbuja de la figura 7.15 para formar el programa de la figura 7.26. Nuestro nuevo programa consiste de **main**, y de las funciones **bubble**, **swap**, **ascending**, y **descending**. La función **bubbleSort** recibe un apuntador a una función ya sea a la función **ascending** o a la función **descending** como un argumento, en adición a un arreglo entero y el tamaño del arreglo. El programa le solicita al usuario que escoja si el arreglo deberá de ser ordenado en orden ascendente o en orden descendente. Si el usuario escribe 1, un apuntador a la función **ascending** se pasa a la función **bubble**, haciendo que el arreglo sea ordenado en orden ascendente. Si el usuario introduce 2, un apuntador a la función **descending** se pasa a la función **bubble**, causando que el arreglo sea ordenado en orden descendente. La salida del programa se muestra en la figura 7.27.

El parámetro siguiente aparece en el encabezado de función correspondiente **a bubble**:

```
int (*compare)(int, int)
```

Esto le indica a **bubble** que espere un parámetro, que es un apuntador a una función, que recibe dos parámetros enteros y regresa un resultado entero. Dado que * tiene una precedencia inferior que los paréntesis que encierran a los parámetros de función, se requieren paréntesis alrededor de ***compare**. Si no se hubieran incluido los paréntesis, la declaración habría sido

```
int *compare(int, int)
```

lo que declara una función que recibe dos enteros como parámetros, y que regresa un apuntador como un entero.

El parámetro correspondiente a la función prototipo de **bubble** es

```
int (*)(int, int)
```

Note que sólo se han incluído tipos, pero para fines de documentación el programador puede incluir nombres, mismos que serán ignorados por el compilador.

La función pasada a **bubble** es llamada en un enunciado **if**, como sigue

```
if ((*compare)(work[count], work[count + 1]))
```

Igual que un apuntador a una variable es desreferenciado para poder tener acceso al valor de la variable, un apuntador a una función es desreferenciado, para utilizar la función.

La llamada a la función podría haber sido efectuada sin desreferenciar el apuntador, como en

```
if (compare(work[count], work[count + 1]))
```

```
/* Multipurpose sorting program using function pointers */
#include <stdio.h>
#define SIZE 10

void bubble(int *, const int, int (*)(int, int));
int ascending(const int, const int);
int descending(const int, const int);

main()
{
   int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
   int counter, order;

   printf("Enter 1 to sort in ascending order,\n");
   printf("Enter 2 to sort in descending order: ");
   scanf("%d", &order);

   printf("\nData items in original order\n");

   for (counter = 0; counter <= SIZE - 1; counter++)
      printf("%4d", a[counter]);

   if (order == 1) {
      bubble(a, SIZE, ascending);
      printf("\nData items in ascending order\n");
   }
   else {
      bubble(a, SIZE, descending);
      printf("\nData items in descending order\n");
   }

   for (counter = 0; counter <= SIZE - 1; counter++)
      printf("%4d", a[counter]);

   printf("\n");

   return 0;
}

void bubble(int *work, const int size, int (*compare)(int, int))
{
   int pass, count;
   void swap(int *, int *);

   for (pass = 1; pass <= size - 1; pass++)

      for (count = 0; count <= size - 2; count++)

         if ((*compare)(work[count], work[count + 1]))
            swap(&work[count], &work[count + 1]);
}
```

Fig. 7.26 Programa de ordenación de uso múltiple, utilizando apuntadores de función (parte 1 de 2).


```
void swap(int *element1Ptr, int *element2Ptr)
{
   int temp;

   temp = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = temp;
}

int ascending(const int a, const int b)
{
   return b < a;
}

int descending(const int a, const int b)
{
   return b > a;
}
```

Fig. 7.26 Programa de ordenación de uso múltiple, utilizando apuntadores de función (parte 2 de 2).

mismo que usa el apuntador directamente como nombre de función. Para llamar una función a través de un apuntador preferimos el primer método, porque de forma explícita ilustra que **compare** es un apuntador a una función, mismo que está desreferenciado para llamar la función. El segundo método de llamar una función a través de un apuntador, lo hace aparecer como si **compare** fuera en realidad una función. Esto pudiera resultar confuso a un usuario del programa que desease ver la definición de la función **compare**, encontrándose que nunca se define en el archivo.

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in descending order
  89  68  45  37  12  10   8   6   4   2
```

Fig. 7.27 Despliegues del programa de ordenación tipo burbuja de la figura 7.26.

Un uso común de los apuntadores de función ocurre en los denominados sistemas manejados por menú. Al usuario se le pide que seleccione una opción de un menú (posiblemente de 1 a 5). Cada opçión está servida por una función diferente. En un arreglo de apuntador a funciones se almacenan apuntadores a cada función. La selección del usuario se utiliza en el arreglo como un subíndice, y el apuntador en el arreglo se utiliza para llamar a la función.

El programa de la figura 7.28 da un ejemplo genérico de la mecánica de la declaración y uso de un arreglo de apuntadores a funciones. Se definen tres funciones —**function1**, **function2** y **function3**— donde cada una de ellas toma un argumento entero y no regresa nada. Los apuntadores a estas tres funciones se almacenan en un arreglo **f**, que se declara como sigue:

```
void *f[3])(int) = {function1, function2, function3};
```

La declaración se lee empezando en el conjunto de paréntesis más a la izquierda, "**f** es un arreglo de tres apuntadores a funciones que toman como argumento un **int** y que regresan **void**". El arreglo se inicializa con los nombres de las tres funciones. Cuando el usuario introduce un valor entre 0 y 2, el valor se utiliza como subíndice en el arreglo de apuntadores a funciones. La llamada de función se efectúa como sigue:

```
(*f[choice])(choice);
```

En la llamada, **f[choice]** selecciona el apuntador en la posición **choice** del arreglo. El apuntador es desreferenciado para llamar la función, y **choice** es pasado como el argumento a la función. Cada función imprime su valor de argumento y su nombre de función, para indicar que la función ha sido correctamente llamada. En los ejercicios, usted desarrollará un sistema manejado por menú.

Resumen

- Los apuntadores son variables que contienen como sus valores direcciones de otras variables.
- Los apuntadores deben ser declarados, antes de que puedan ser usados.
- La declaración

```
int *ptr;
```

- declara a **ptr** como un apuntador a un objeto del tipo **int**, y se lee, "**ptr** es un apuntador a **int**". El * como se utiliza aquí en una declaración, indica que la variable es un apuntador.
- Existen tres valores que pueden ser utilizados para inicializar un apuntador; **0**, **NULL**, o una dirección. Es lo mismo inicializar un apuntador a **0** e inicializar el mismo apuntador a **NULL**.
- El único entero que puede ser asignado a un apuntador es 0.
- El operador **&** (de dirección) regresa la dirección de su operando.
- El operando del operador de dirección debe ser una variable; el operador de dirección no puede ser aplicado a constantes, a expresiones, o a variables declaradas con la clase de almacenamiento **register**.
- El operador *, conocido como operador de indirección o de desreferenciación, regresa el valor del objeto al cual apunta su operando en memoria. Esto se llama desreferenciar el apuntador.


```
/* Demonstrating an array of pointers to functions */
#include <stdio.h>

void function1(int);
void function2(int);
void function3(int);

main()
{
   void (*f[3])(int) = {function1, function2, function3};
   int choice;

   printf("Enter a number between 0 and 2, 3 to end: ");
   scanf("%d", &choice);

   while (choice >= 0 && choice < 3) {
      (*f[choice])(choice);
      printf("Enter a number between 0 and 2, 3 to end: ");
      scanf("%d", &choice);
   }

   printf("You entered 3 to end\n");
   return 0;
}

void function1(int a)
{
   printf("You entered %d so function1 was called\n\n", a);
}

void function2(int b)
{
   printf("You entered %d so function2 was called\n\n", b);
}

void function3(int c)
{
   printf("You entered %d so function3 was called\n\n", c);
}
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
You entered 3 to end
```

Fig. 7.28 Cómo demostrar un arreglo de apuntadores a funciones.

- Al llamar una función con un argumento, que el llamador desea que la función llamada modifique, se pasa la dirección del argumento. A continuación la función llamada utiliza el operador de indirección (*) para modificar el valor del argumento de la función llamadora.
- Una función que recibe una dirección como un argumento, debe incluir un apuntador como su parámetro formal correspondiente.
- En los prototipos de función no es necesario incluir los nombres de los apuntadores; sólo es necesario incluir los tipos de apuntador. Los nombres de apuntador pueden ser incluídos por razones de documentación, pero el compilador los ignorará.
- El calificador **const** permite al programador informarle al compilador que el valor de una variable particular no debe ser modificado.
- Si se intenta modificar un valor declarado **const**, el compilador lo detectará y emitirá una advertencia o un error, dependiendo del compilador particular.
- Existen cuatro formas para pasar un apuntador a una función: un apuntador no constante a datos no constantes, un apuntador constante a datos no constantes, un apuntador no constante a datos constantes y un apuntador constante a datos constantes.
- Los arreglos son pasados por referencia en forma automática, porque el valor del nombre del arreglo es la dirección del mismo.
- Para pasar un elemento de un arreglo en llamada por referencia, deberá ser pasada la dirección del elemento específico del arreglo.
- C proporciona el operador unario especial **sizeof**, para determinar el tamaño en bytes de un arreglo (o de cualquier otro tipo de datos) durante la compilación del programa.
- Al aplicarse al nombre de un arreglo, el operador **sizeof** regresa el número total de bytes en el arreglo, como un entero.
- El operador **sizeof** puede ser aplicado a cualquier nombre de variable, tipo o constante.
- Las operaciones aritméticas que pueden ser ejecutadas sobre apuntadores son incrementar (++) un apuntador, decrementar (–) un apuntador, sumar (+ o +=) un apuntador y un entero, restar (- o -=) un apuntador y un entero y restar un apuntador de otro.
- Cuando un entero se añade o se resta de un apuntador, el apuntador se incrementa o se decrementa por dicho entero, multiplicado por el tamaño del objeto al cual se apunta.
- Las operaciones aritméticas de apuntador deberían ser ejecutadas sólo en porciones contiguas de memoria, como existen en un arreglo. Todos los elementos de un arreglo están almacenados en memoria de forma contigua.
- Al ejecutar aritmética de apuntador en un arreglo de caracteres, los resultados son como en aritmética normal, dado que cada carácter está almacenado en un byte de memoria.
- Los apuntadores pueden ser asignados de uno a otro, si ambos apuntadores son del mismo tipo. De lo contrario, deberá de utilizarse una conversión (cambio de tipo). La excepción a lo anterior es un apuntador a **void**, que es un tipo de apuntador genérico que puede contener apuntadores de cualquier tipo. A los apuntadores a **void** se les pueden asignar apuntadores de otros tipos y pueden ser asignados a apuntadores de otros tipos sin necesidad de una conversión.
- Un apuntador **void** no puede ser desreferenciado.
- Los apuntadores pueden ser comparados utilizando los operadores de igualdad y relacionales. Las comparaciones de apuntadores por lo regular son significativas sólo si los apuntadores apuntan a miembros del mismo arreglo.

- Los apuntadores pueden tener subíndices exactamente como los nombres de los arreglos.
- Un nombre de arreglo sin subíndice es un apuntador al primer elemento del arreglo.
- En notación apuntador/desplazamiento, el desplazamiento es el mismo que un subíndice de arreglo.
- Todas las expresiones de arreglos con subíndice pueden ser escritas con un apuntador y un desplazamiento, utilizando ya sea el nombre del arreglo como un apuntador, o un apuntador por separado, que apunte al arreglo.
- Un nombre de arreglo es un apuntador constante, que siempre apunta a la misma posición en memoria. Los nombres de arreglo no pueden ser modificados, como pueden ser modificados los apuntadores convencionales.
- Es posible tener arreglos de apuntadores.
- Es posible tener apuntadores a funciones.
- Un apuntador a una función es la dirección donde reside el código de la función.
- Los apuntadores a las funciones pueden ser pasados a funciones, regresados de funciones, almacenados en arreglos y asignados a otros apuntadores.
- Un uso común de apuntadores de función es en los sistemas conocidos como manejados por menú.

Terminología

como sumar un apuntador y un entero
operador de dirección (&)
arreglo de apuntadores
arreglo de cadenas
llamada por referencia
llamada por valor
apuntador de carácter
const
apuntador constante
apuntador constante a datos constantes
apuntador constante a datos no constantes
decrementar un apuntador
desreferenciar un apuntador
operador de desreferenciar (*)
referencia directa a una variable
asignación dinámica de memoria
apuntador de función
incrementar un apuntador
posposición indefinida
indirección
operador de indirección (*)
referencia indirecta a una variable
cómo inicializar apuntadores
lista enlazada
apuntador no constante a datos constantes
apuntador no constante a datos no constantes
apuntador **NULL**
desplazamiento
apuntador
aritmética de apuntador
asignación de apuntador
comparación de apuntador
expresión de apuntador
indexación de apuntador
notación apuntador/desplazamiento
subscripción de apuntador
apuntador a una función
apuntador a **void** (**void ***)
tipos de apuntador
principio de mínimo privilegio
llamada por referencia simulada
sizeof
arreglo de cadenas
resta de un entero de un apuntador
resta de dos apuntadores
refinación descendente paso a paso
void * (apuntador a **void**)

Errores comunes de programación

7.1 El operador de indirección * no se distribuye a todos los nombres de variables de una declaración. Cada apuntador debe de ser declarado con el * prefijo al nombre.

7.2 Desreferenciar un apuntador que no haya sido correctamente inicializado, o que no haya sido asignado para apuntar a una posición específica en memoria. Esto podría causar un error fatal en tiempo de ejecución, o podría modificar de forma accidental datos importantes y permitir que el programa se ejecute hasta su terminación proporcionando resultados incorrectos.

7.3 No desreferenciar un apuntador, cuando es necesario hacerlo para obtener el valor hacia el cual el apuntador señala.

7.4 No estar consciente que una función está esperando apuntadores como argumentos en llamadas por referencia, y está pasando argumentos en llamadas por valor. Algunos compiladores toman los valores, suponiendo que son apuntadores y desreferencian los valores como apuntadores. En tiempo de ejecución, a menudo se generan violaciones de acceso a la memoria o fallas de segmentación. Otros compiladores detectan falta de coincidencia en tipo entre argumentos y parámetros, que generan mensajes de error.

7.5 Utilizar aritmética de apuntadores en un apuntador que no se refiere a un arreglo de valores.

7.6 Restar o comparar dos apuntadores que no se refieren al mismo arreglo.

7.7 Salirse de cualquiera de los dos extremos de un arreglo al utilizar aritmética de apuntadores.

7.8 Se generará un error de sintaxis asignar un apuntador de un tipo a un apuntador de otro tipo, si ninguno de los dos es del tipo `void *`.

7.9 Desreferenciar un apuntador `void *`.

7.10 Es un error de sintaxis intentar modificar un nombre de arreglo con aritmética de apuntador.

Prácticas sanas de programación

7.1 En un nombre de variable de apuntador incluya las letras `ptr` para que quede claro que estas variables son apuntadores y deben ser manejadas con propiedad.

7.2 Inicialice los apuntadores para evitar resultados inesperados.

7.3 Utilice llamadas por valor para pasar argumentos a una función, a menos de que en forma explícita el llamador requiera que la función llamada modifique el valor de la variable del argumento en el entorno de llamador. Esto es otro ejemplo del principio del mínimo privilegio. Algunas personas prefieren llamadas por referencia por razones de rendimiento ya que el copiado de valores de término medio se le elude.

7.4 Antes de utilizar una función, verifique el prototipo de función correspondiente a esta función, a fin de determinar si la función es capaz de modificar los valores que se le pasan.

7.5 Al manipular arreglos utilice notación de arreglos en vez de notación de apuntadores. Aunque el programa pudiera tomar más tiempo para su compilación, probablemente será mucho más claro.

Sugerencias de rendimiento

7.1 Pase grandes objetos como son estructuras utilizando apuntadores a datos constantes para obtener los beneficios de rendimiento de llamadas por referencia y la seguridad de llamadas por valor.

7.2 Pasar el tamaño de un arreglo a una función ocupa tiempo y espacio de pilas adicional, porque debe ejecutarse una copia del tamaño para ser pasada a la función. Las variables globales, sin embargo, no requieren de tiempo adicional o de espacio, porque son accesibles de forma directa por cualquier función.

7.3 Durante la compilación la notación de subíndices de arreglo se convierte a notación de apuntador, por lo que escribir expresiones de subíndices de arreglo con notación de apuntadores, puede ahorrar tiempo de compilación.

7.4 Algunas veces un algoritmo que aparece de forma "natural" puede contener problemas sutiles de rendimiento, como es la posposición indefinida. Busque algoritmos que eviten la posposición indefinida.

Sugerencias de portabilidad

7.1 A pesar de que en ANSI C `const` está bien definido, algunos sistemas no lo tienen incorporado.

7.2 El número de bytes utilizados para almacenar un tipo particular de datos, pudiera variar de sistema a sistema. Al escribir programas que dependan de tamaños de tipos de datos, y que se ejecutarán en diversos sistemas de computación, utilice `sizeof` para determinar el número de bytes utilizados para almacenar los tipos de datos.

7.3 La mayor parte de las computadoras de hoy día tienen enteros de 2 o de 4 bytes. Algunas de las máquinas más modernas utilizan enteros de 8 bytes. Dado que los resultados de la aritmética de apuntadores depende del tamaño de los objetos a los cuales el apuntador señala, la aritmética de los apuntadores depende de la máquina.

Observaciones de ingeniería de software

7.1 El calificador `const` puede ser utilizado para forzar el principio del mínimo privilegio. La utilización del principio de mínimo privilegio para diseñar apropiadamente el software reduce en forma importante el tiempo de depuración y efectos inadecuados colaterales, y hace un programa más fácil de modificar y mantener.

7.2 Si un valor no se modifica (o no debería modificarse) en el cuerpo de una función al cual es pasado, el valor deberá declararse `const`, para asegurarse que no se modifica accidentalmente.

7.3 En una función llamadora sólo un valor puede ser alterado cuando se utiliza llamada por valor. Este valor debe ser asignado a partir del valor de regreso de la función. Para modificar varios valores en una función llamadora, debe utilizarse llamada por referencia.

7.4 Colocar prototipos de función en las definiciones de otras funciones obliga al principio del mínimo privilegio al restringir las llamadas correctas de función sólo a aquellas funciones en las cuales dichos prototipos aparecen.

7.5 Al pasar un arreglo a una función, pase también su tamaño. Esto ayuda a generalizar la función. Las funciones generales son a menudo reutilizables.

7.6 Las variables globales violan el principio del mínimo privilegio y son un ejemplo de ingeniería de software pobre.

Ejercicios de autoevaluación

7.1 Conteste cada uno de los siguientes:

a) Un apuntador es una variable que contiene como su valor la _____ de otra variable.

b) Los tres valores que se pueden utilizar para inicializar un apuntador son _____, _____, o una _____.

c) El único entero que puede ser asignado a un apuntador es _____.

7.2 Indique si lo siguiente es cierto o falso. Si la respuesta es falso, explique por qué.

a) El operador de dirección `&` puede sólo aplicarse a constantes, a expresiones y a variables declaradas con la clase de almacenamiento `register`.

b) Un apuntador que se declara ser `void` puede ser desreferenciado.

c) Los apuntadores de diferentes tipos no pueden ser asignados uno al otro, sin una operación de conversión.

7.3 Conteste cada una de las siguientes. Suponga que números de una precisión de punto flotante están almacenados en 4 bytes, y que la dirección inicial del arreglo está en memoria en la posición 1002500. Cada parte del ejercicio deberá de utilizar los resultados de las partes anteriores, donde sea apropiado.

a) Declare un arreglo del tipo `float`, llamado `numbers` con 10 elementos, e inicialice los elementos a los valores `0.0, 1.1, 2.2, ....9.9`. Suponga que la constante simbólica `SIZE` ha sido definida como `10`.

b) Declare un apuntador `nPtr`, que apunte a un objeto de tipo `float`.

c) Imprima los elementos del arreglo **numbers**, utilizando notación de subíndice de arreglo. Utilice una estructura **for**, y suponga que se ha declarado la variable de control entera **i**. Imprima cada número con una precisión de una posición a la derecha del punto decimal.
d) Proporcione dos enunciados por separado, que asignen la dirección inicial del arreglo **numbers** a la variable de apuntador **nPtr**.
e) Imprima los elementos del arreglo **numbers**, utilizando la notación apuntador/desplazamiento, con el apuntador **nPtr**.
f) Imprima los elementos del arreglo **numbers**, utilizando notación apuntador/desplazamiento con el nombre del arreglo como el apuntador.
g) Imprima los elementos del arreglo **numbers** mediante subíndices del apuntador **nPtr**.
h) Refiérase al elemento 4 del arreglo **numbers**, utilizando la notación de subíndice de arreglo, la notación de apuntador/desplazamiento y utilizando como el apuntador el nombre del arreglo, la notación de subíndice de apuntador con **nPtr**, y la notación de apuntador/desplazamiento con **nPtr**.
i) Suponiendo que **nPtr** apunta al principio del arreglo **numbers**, ¿cuál es la dirección referenciada por **nPtr + 8**? ¿Cuál es el valor almacenado en esa posición?
j) Suponiendo que **nPtr** apunta a **numbers[5]**, qué dirección es referenciada por **nPtr -= 4**. ¿Cuál es el valor almacenado en esta posición?

7.4 Para cada uno de los siguientes, escriba un enunciado que ejecute la tarea indicada. Suponga que se han declarado las variables de punto flotante **number1** y **number2**, y que **number1** ha sido inicializado a **7.3**.

a) Declare la variable **fPtr** que sea un apuntador a un objeto del tipo **float**.
b) Asigne la dirección de la variable **number1** a una variable de apuntador **fPtr**.
c) Imprima el valor del objeto señalado hacia por **fPtr**.
d) Asigne el valor del objeto al que se señala con **fPtr** a la variable **number2**.
e) Imprima el valor de **number2**.
f) Imprima la dirección de **number1**. Utilice el especificador de conversión **%p**.
g) Imprima la dirección almacenada en **fPtr**. Utilice el especificador de conversión **%p**. ¿Es el valor impreso el mismo al de la dirección de **number1**?

7.5 Haga cada uno de lo siguiente.

a) Escriba el encabezado de función de una función llamada **exchange** que toma como parámetros dos apuntadores a números de punto flotante **x** e **y**, y no regresa un valor.
b) Escriba el prototipo de función para la función en la parte (a).
c) Escriba el encabezado de función para la función llamada **evaluate**, que regresa un entero y que toma como parámetros el entero **x** y un apuntador a la función **poly**. La función **poly** toma un parámetro entero y regresa un entero.
d) Escriba el prototipo de función para la función de la parte (c).

7.6 Encuentre el error en cada uno de los segmentos de programas siguientes. Suponga

```
int *zPtr; /* zPtr will reference array z */
int *aPtr = NULL;
void *sPtr = NULL;
int number, i;
int z[5] = {1, 2, 3, 4, 5}

sPtr = z;
```

a) `++zptr;`
b)
```
/* use pointer to get first value of array */
number = zPtr;
```
c)
```
/* assign array element 2 (the value 3) to number */
number = *zPtr[2];
```


d)
```
/* print entire array z */
for (i = 0; i  5; i++)
printf("%d ", zPtr[i]);
```
e)
```
/* assign the value pointed to by sPtr number */
number = *sPtr;
```
f) `++z;`

Respuestas a los ejercicios de autoevaluación

7.1 a) dirección. b) `0`, `NULL`, una dirección. c) `0`

7.2 a) Falso. El operador de dirección pude ser sólo aplicado a variables y no puede aplicarse a variables declaradas con la clase de almacenamiento **register**.
b) Falso. Un apuntador **void** no puede ser desreferenciado porque no existe manera de saber con exactitud cuantas bytes de memoria deberán ser desreferenciadas.
c) Falso. Apuntadores del tipo **void** pueden ser asignados como apuntadores de otros tipos, y apuntadores del tipo **void** pueden ser asignados a apuntadores de otros tipos.

7.3 a)
```
float numbers[SIZE] = {0.0, 1.1, 2.2, 3.3, 4.4, 5.5,
                       6.6, 7.7, 8.8, 9.9};
```
b) `float *nPtr;`
c)
```
for (i = 0; i <= 0; i  SIZE - 1; i++)
 printf("%.1f ", numbers[i]);
```
d)
```
nPtr = numbers;
nPtr = &numbers[0];
```
e)
```
for (i = 0; i <= SIZE - 1; i++)
  printf("%.1f ", *(nPtr + i));
```
f)
```
for (i = 0; i <= SIZE - 1; i++)
printf("%.1f ", *(numbers + i));
```
g)
```
for (i = 0; i <= SIZE - 1; i++)
 printf("%.1f ", nPtr[i]);
```
h)
```
numbers[4]
*(numbers + 4)
nPtr[4]
*(nPtr + 4)
```
i) La dirección es `1002500 + 8 * 4 = 1002532`. El valor es `8.8`.
j) La dirección de `numbers[5]` es `1002500 + 5 * 4 = 1002520`.
La dirección de `nPtr -= 4` es `1002520 - 4 * 4 = 1002504`.
El valor en dicha posición es `1.1`.

7.4 a)
```
float *fPtr;
```
b)
```
fPtr = &number1;
```
c)
```
printf("The value of *fPtr is %f\n", *fPtr);
```
d)
```
number2 = *fPtr;
```
e)
```
printf("The value of number2 is %f\n", number2);
```
f)
```
printf("The address of number1 is %p\n", &number1);
```
g)
```
printf("The address stored in fptr is %p\n", fPtr);
```
Sí, el valor es el mismo.

7.5 a)
```
void exchange (float*x, float*y)
```
b)
```
voyd exchange (float *,float *)
```
c)
```
int evaluate(int x, int (*poly)(int))
```
d)
```
int evaluate(int, int (*)(int));
```

7.6 a) Error: `zPtr` no ha sido inicializado.
Corrección: inicialice `zPtr` con `zPtr = z;`
b) Error: El apuntador no está desreferenciado.
Corrección: modifique el enunciado a `number = *zPtr;`
c) Error: `zPtr[2]` no es un apuntador y no deberá ser desreferenciado.
Corrección: modifique `*zPtr[2]` a `zPtr[2]`.
d) Error: referirse a un elemento de arreglo exterior a los límites del mismo, utilizando subíndices de apuntador.
Corrección: cambie el valor final de la variable de control en la estructura `for` a `4`.
e) Error: Desreferenciando un apuntador `void`.
Corrección: a fin de desreferenciar el apuntador, primero debe de ser convertido a un apuntador entero. Modifique el enunciado anterior a `number = (int *)sPtr;`
f) Error: Tratar de modificar el nombre de un arreglo utilizando aritmética de apuntador.
Corrección: utilice una variable de apuntador, en vez del nombre del arreglo, para llevar a cabo aritmética de apuntador, o suscriba el nombre del arreglo para referirse a un elemento específico.

Ejercicios

7.7 Conteste cada una de las siguientes:
a) El operador _____ regresa la posición en memoria donde está almacenado su operando.
b) El operador _____ regresa el valor del objeto hacia el cual apunta su operando.
c) Para simular llamada por referencia al pasar una variable no de arreglo a una función, es necesario pasar el _____ de la variable a la función.

7.8 Indique si lo siguiente es verdadero o falso. Si es falso, explique por qué.
a) Dos apuntadores que señalen a diferentes arreglos no pueden ser comparados en forma significativa.
b) Dado que el nombre de un arreglo es un apuntador para el primer elemento del mismo, los nombres de los arreglos pueden ser manipulados de la misma forma que los apuntadores.

7.9 Conteste cada una de los siguientes. Suponga que enteros unsigned están almacenados en 2 bytes, y que las dirección inicial del arreglo es en la posición 1002500 en memoria.
a) Declare un arreglo del tipo `unsigned int` llamado `values` con 5 elementos, e inicie los elementos a los enteros pares del 2 al 10. Suponga la constante simbólica `SIZE` definida como `5`.
b) Declare un apuntador `vPtr` que señale a un objeto del tipo `unsigned int`.
c) Imprima los elementos del arreglo `values` utilizando notación de subíndices de arreglo. Utilice una estructura `for` y suponga que ha sido declarada una variable de control entera `i`.
d) Proporcione dos enunciados separados que asignen la dirección inicial del arreglo `values` a la variable de apuntador `vPtr`.
e) Imprima los elementos del arreglo `values`, utilizando notación apuntador/desplazamiento.
f) Imprima los elementos del arreglo `values`, utilizando notación apuntador/desplazamiento con el nombre del arreglo como el apuntador.
g) Imprima los elementos del arreglo `values`, mediante subíndices del apuntador al arreglo.
h) Refiérase al elemento 5 del arreglo `values` utilizando notación de subíndice de arreglo, notación de apuntador/desplazamiento con el nombre del arreglo como el apuntador, notación de subíndice de apuntador, y notación de apuntador/desplazamiento.
i) ¿Cuál es la dirección referenciada por `vPtr + 3`? ¿Cuál es el valor almacenado en esa posición?
j) Suponiendo que `vPtr` apunta a `values[4]`, ¿cuál es la dirección referenciada por `vPtr -= 4`. Cuál es el valor almacenado en dicha posición?

7.10 Para cada uno de los siguientes, escriba un solo enunciado que ejecute la tarea indicada. Suponga que han sido declaradas las variables enteras long `value1` y `value2`, y que `value1` ha sido inicializado a `200000`.
a) Declare la variable `lPtr` que sea un apuntador a un objeto del tipo `long`.
b) Asigne la dirección de la variable `value1` a la variable de apuntador `lPtr`.
c) Imprima el valor del objeto señalado por `lPtr`.
d) Asigne el valor del objeto señalado por `lPtr` a la variable `value2`.
e) Imprima el valor de `value2`.
f) Imprima la dirección de `value1`.
g) Imprima la dirección almacenada en `lPtr`. ¿Es el valor impreso el mismo que la dirección de `value1`?

7.11 Lleve a cabo cada uno de lo siguiente.
a) Escriba el encabezado de función para la función `zero` que toma un parámetro de arreglo entero long `bigIntegers` y no regresa un valor.
b) Escriba la función prototipo para la función de la parte (a).
c) Escriba el encabezado de función de la función `add1AndSum` que toma un parámetro de arreglo entero `oneTooSmall` y regresa un entero.
d) Escriba el prototipo de función para la función descrita en la parte (c).

Nota: los ejercicios 7.12 al 7.15 tienen un cierto elemento de reto. Una vez que haya llevado a cabo estos problemas, deberá estar en condiciones de resolver con facilidad los juegos de naipes más populares.

7.12 Modifique el programa de la figura 7.24, de tal forma que la función de distribución de naipes reparta una mano de póker de cinco naipes. A continuación escriba las funciones adicionales siguientes:
a) Determine si la mano contiene un par.
b) Determine si la mano contiene dos pares.
c) Determine si la mano contiene tres de un tipo (es decir, por ejemplo, tres jacks).
d) Determine si la mano contiene cuatro de un tipo (por ejemplo, cuatro ases).
e) Determine si la mano contiene un color (es decir, todos los cinco naipes del mismo palo).
f) Determine si la mano contiene una flor imperial (es decir, cinco naipes de valores nominales consecutivos).

7.13 Utilice las funciones desarrolladas en el ejercicio 7.12, para escribir un programa que distribuya dos manos de póker de 5 naipes, evalúe cada una de las manos, y determine cuál es la mejor mano.

7.14 Modifique el programa desarrollado en el ejercicio 7.13, de tal forma que pueda simular al tallador. La mano de 5 naipes del tallador se distribuye "tapada", de tal manera que el jugador no puede verla. El programa deberá entonces evaluar la mano del tallador, y basado en la calidad de la misma, el tallador deberá de solicitar uno, dos o tres naipes adicionales para remplazar el número correspondiente de naipes innecesarios de la mano original. El programa deberá entonces reevaluar la mano del tallador. (*Advertencia:* ¡Este es un problema difícil!)

7.15 Modifique el programa desarrollado en el ejercicio 7.14, de tal forma de que pueda manejar de forma automática la mano del tallador, pero el jugador pueda decidir qué naipes de su mano remplazar. El programa deberá entonces evaluar ambas manos y determinar quien gana. Ahora utilice este nuevo programa para jugar 20 juegos contra la computadora. ¿Quién gana más juegos, usted o la computadora? Haga que uno de sus amigos juegue 20 juegos contra la computadora. ¿Quién gana más juegos? Basado en los resultados de estos juegos, efectúe modificaciones apropiadas para refinar su programa de jugar al póker (esto también es un problema difícil). Juegue otros 20 juegos. ¿Su programa modificado juega un mejor juego?.

7.16 En el programa de barajar y distribuir naipes de la figura 7.24, intencionalmente utilizamos un algoritmo de barajar ineficiente, que introducía la posibilidad de posposición indefinida. En este problema, usted creará un algoritmo de barajar de alto rendimiento, que elimine la posposición indefinida.

Modifique el programa de la figura 7.24 como sigue. Empiece inicializando el arreglo `deck` como se muestra en la figura 7.29. Modifique la función `shuffle` para ciclar, renglón por renglón y columna por

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	14	15	16	17	18	19	20	21	22	23	24	25	26
2	27	28	29	30	31	32	33	34	35	36	37	38	39
3	40	41	42	43	44	45	46	47	48	49	50	51	52

Fig. 7.29 Arreglo **deck**, no barajado.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	19	40	27	25	36	46	10	34	35	41	18	2	44
1	13	28	14	16	21	30	8	11	31	17	24	7	1
2	12	33	15	42	43	23	45	3	29	32	4	47	26
3	50	38	52	39	48	51	9	5	37	49	22	6	20

Fig. 7.30 Arreglo **deck** de muestra, barajado.

columna, a través del arreglo, tocando cada elemento una vez. Cada elemento deberá ser intercambiado con un elemento elegido al azar del arreglo.

Imprima el arreglo resultante para determinar si el mazo ha sido barajado de manera satisfactoria (como en la figura 7.30, por ejemplo). Posiblemente desee que su programa llame varias veces a la función **shuffle**, para asegurarse que se ha barajado correctamente.

Note que, a pesar que el enfoque en este problema mejora el algoritmo de barajar, el algoritmo de repartir todavía requiere buscar en el arreglo **deck** el naipe 1, a continuación el naipe 2, y en seguida el naipe 3, y así en lo sucesivo. Y lo que es aún peor, aun después de que el algoritmo de distribución localiza y reparte el naipe, el algoritmo sigue buscando a través del resto del mazo. Modifique el programa de la figura 7.24 de tal forma de que una vez que se haya distribuido un naipe, no se hagan intentos posteriores para hacer coincidir este número de naipe, y el programa continúe de inmediato con la distribución del naipe siguiente. En el capítulo 10 desarrollamos un algoritmo de reparto de naipes, que requiere de una operación por cada naipe.

7.17 (*Simulación: La liebre y la tortuga*). En este problema recreará uno de los verdaderos grandes momentos de la historia, es decir, la carrera clásica entre la liebre y la tortuga. Utilizará una generación de números aleatorios para desarrollar una simulación de este evento memorable.

Nuestros contendientes empiezan la carrera en el "cuadro 1" de 70 cuadros. Cada cuadro representa una posición posible a lo largo de la carrera. La línea de meta está en el cuadro 70. El primer contendiente que llegue o que pase el cuadro 70 gana una cubeta de zanahorias y lechugas frescas. El desarrollo de la pista transcurre sobre la ladera de una montaña resbalosa, por lo que ocasionalmente los corredores pierden terreno.

Existe un reloj que hace tic una vez por segundo. Con cada tic del reloj, su programa deberá ajustar la posición de los animales, de acuerdo con las reglas de la página siguiente:

Utilice variables para llevar control de las posiciones de los animales (es decir, los números de posición son del 1 al 70). Inicie cada animal en la posición 1 (es decir, en la "línea de arranque"). Si un animal resbala antes del cuadro 1, regrese el animal de vuelta al cuadro 1.

Genere los porcentajes de la tabla anterior mediante la producción de un entero al azar, i, en el rango $1 \le i \le 10$. Para la tortuga, ejecute un "paso rápido" cuando $1 \le i \le 5$, un "resbalón" cuando $6 \le i \le 7$, o un "paso lento" cuando $8 \le i \le 10$. Utilice una técnica similar para mover a la liebre.

Empiece la carrera imprimiendo

```
BANG !!!!!
AND THEY'RE OFF !!!!!
```


Animal	Tipo de movimiento	Porcentaje del tiempo	Movimiento real
Tortuga	Paso rápido	50%	3 cuadros a la derecha
	Resbalón	20%	6 cuadros a la izquierda
	Paso lento	30%	1 cuadro a la derecha
Liebre	Dormido	20%	Ningún movimiento
	Salto grande	20%	9 cuadros a la derecha
	Resbalón grande	10%	12 cuadros a la izquierda
	Salto pequeño	30%	1 cuadro a la derecha
	Pequeño resbalón	20%	2 cuadros a la izquierda

Entonces, por cada pulso del reloj (es decir, para cada repetición del ciclo) imprima una línea de 70 posiciones, que muestre la letra `T` en la posición de la tortuga, y la letra `H` en la posición de la liebre. Ocasionalmente, ambos contendientes coincidirán en el mismo cuadro. En esta condición, la tortuga muerde a la liebre y su programa deberá de imprimir `OUCH!!!` empezando en esta posición. Todas las posiciones de impresión, distintas a la de `T`, la `H`, o la `OUCH!!!` (en caso de un empate) deberán estar vacías.

Después de que se ha impreso cada línea, pruebe si ambos animales han alcanzado o han pasado el cuadro 70. Si es así, entonces imprima el ganador y dé por terminada la simulación. Si la tortuga gana, imprima `TORTOISE WINS!!! YAY!!!`. Si la liebre gana, imprima `Hare wins. Yuch.` Si ambos animales ganan en el mismo pulso del reloj, quizás desee favorecer a la tortuga (el "más débil"), o quizás desee imprimir `It's a tie`. Si ninguno de los animales gana, otra vez vuelva a ejecutar el ciclo, para simular el siguiente pulso del reloj. Cuando esté listo para ejecutar su programa, reúna un grupo de observadores para presenciar la carrera. ¡Se asombrará del interés que su auditorio mostrará!

Sección especial: cómo construir su propia computadora

En los siguientes varios problemas, nos desviamos temporalmente del mundo de la programación en lenguajes de alto nivel. Le "quitamos la cubierta" a una computadora y estudiamos su estructura interna. Presentamos la programación en lenguaje máquina y escribimos varios programas en lenguaje máquina. Para que esto resulte una experiencia en especial valiosa, luego construimos una computadora (mediante la técnica de la *simulación* basada en software) ¡en la cual usted podrá ejecutar sus programas en lenguaje de máquina!

7.18 (*Programación en lenguaje de máquina*). Procedamos a crear una computadora, que llamaremos Simpletron. Como su nombre implica, es una máquina sencilla, pero como veremos pronto, es también poderosa. Simpletron ejecuta programas, escritos en el único lenguaje que comprende en forma directa, esto es, en el lenguaje de máquina Simpletron, o bien, abreviando, en LMS.

Simpletron contiene un *acumulador* —un "registro especial" en el cual se coloca la información ante Simpletron, mismo que utiliza esta información en cálculos o la examina de varias formas. Toda la información en Simpletron se maneja en términos de *palabras*. Una palabra es un número decimal de cuatro dígitos signados, como `+3364`, `-1293`, `+0007`, `-0001`, etcétera. Simpletron está equipado con una memoria de 100 palabras, y estas palabras están referenciadas por sus números de posición `00`, `01`, ..., `99`.

Antes de ejecutar un programa LMS, debemos de *cargar*, es decir colocar, el programa en memoria. La primera instrucción (o enunciado) de cualquier programa LMS siempre se colocará en la posición `00`.

Cada instrucción escrita en LMS ocupa una palabra de la memoria Simpletron (y, por lo tanto, las instrucciones resultan números decimales de cuatro dígitos signados). Supondremos que el signo de una instrucción LMS será siempre más, pero el signo de una palabra de datos podrá ser o más o menos. Cada posición en la memoria de Simpletron pudiera contener instrucción, o un valor de datos usado por un programa, o un área no utilizada (y, por lo tanto, no definida) de memoria. Los dos primeros dígitos de cada instrucción LMS son el *código de operación*, que define la operación a ejecutarse. Los códigos de operación LMS se resumen en la figura 7.31.

Los últimos dos dígitos de una instrucción LMS son el *operando*, es decir la dirección de la posición de memoria que contiene la palabra a la cual se aplica la operación. Ahora consideremos varios programas simples LMS.

Código de operación	Significado
Operaciones de entrada/salida:	
`#define READ 10`	Lee una palabra de la terminal y la coloca en una posición específica en memoria.
`#define WRITE 11`	Escribe una palabra desde una posición específica en memoria a la terminal.
Operaciones de cargar/almacenar:	
`#define LOAD 20`	Carga una palabra de una posición específica en memoria al acumulador.
`#define STORE 21`	Almacena una palabra del acumulador a una posición específica en memoria.
Operaciones aritméticas:	
`#define ADD 30`	Añade una palabra de una posición específica en memoria a la palabra en el acumulador (deja el resultado en el acumulador).
`#define SUBTRACT 31`	Sustrae una palabra de una posición específica en memoria de la palabra existente en el acumulador (deja el resultado en el acumulador).
`#define DIVIDE 32`	Divide la palabra existente en el acumulador entre una palabra de una posición específica en la memoria (deja el resultado en el acumulador).
`#define MULTIPLY 33`	Multiplica una palabra de una localización específica en la memoria por la palabra en el acumulador (deja el resultado en el acumulador).
Operaciones de transferencia de control:	
`#define BRANCH 40`	Se desvía a una posición específica en memoria.
`#define BRANCHNEG 41`	Se desvía a una posición específica en memoria si el acumulador es negativo.
`#define BRANCHZERO 42`	Se desvía a una posición específica en memoria si el acumulador es cero.
`#define HALT 43`	Se detiene, es decir, el programa ha terminado su tarea.

Fig. 7.31 Códigos de operación del lenguaje de máquina Simpletron (LMS).

Ejemplo 1 Posición	Número	Instrucción
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

Este programa LMS lee dos números del teclado y calcula e imprime su suma. La instrucción **+1007** lee el primer número del teclado y lo coloca en la posición **07** (que ha sido inicializada a cero). Entonces **+1008** lee el siguiente número a la posición **08**. La instrucción *load*, **+2007**, coloca el primer número en el acumulador, y la instrucción *add* **+3008**, suma el segundo número al número en el acumulador. *Todas las instrucciones aritméticas LMS dejan sus resultados en el acumulador.* La instrucción *store* **+2109**, coloca el resultado de regreso en la posición de memoria **09** a partir de la cual la instrucción *write* **+1109**, toma el número y lo imprime (en forma de un número decimal de cuatro dígitos signado). La instrucción *halt* **+4300**, da por terminada la ejecución.

Ejemplo 2 Posición	Número	Instrucción
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Branch negative to 07)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)
08	+4300	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

Este programa LMS lee dos números del teclado, y determina e imprime el valor más grande. Note el uso de la instrucción **+4107** como una transferencia condicional de control, de la misma forma que un enunciado **if** de C. Ahora escriba programas LMS que lleven a cabo cada una de las tareas siguientes.

a) Utilice un ciclo controlado por centinela, para leer 10 números positivos y calcular e imprimir su suma.
b) Use un ciclo controlado por contador para leer 7 números, algunos positivos y algunos negativos, y calcular e imprimir su promedio.
c) Lea una serie de números y determine e imprima el número más grande. El primer número leído indicará cuántos números deberán de procesarse.

7.19 (*Un simulador de computadora*). Al principio pudiera resultar extravagante, pero en este problema construirá su propia computadora. No, no tendrá que soldar componentes. Más bien, utilizará la técnica poderosa de la *simulación basada en software* para crear un *modelo en software* de Simpletron. No quedará decepcionado. Su simuladora Simpletron convertirá la computadora que está utilizando en una Simpletron, y de hecho será capaz de ejecutar, probar y depurar los programas LMS que escribió en el ejercicio 7.18.

Cuando haga funcionar su simuladora Simpletron, deberá empezar imprimiendo:

```
*** Welcome to Simpletron! ***

*** Please enter your program one instruction ***
*** (or data word) at a time. I will type the ***
*** location number and a question mark (?).  ***
*** You then type the word for that location. ***
*** Type the sentinel -99999 to stop entering ***
*** your program. ***
```

Simule la memoria de Simpletron con un arreglo **memory** de un subíndice, que tenga 100 elementos. Ahora suponga que el simulador está funcionando, y examinemos el diálogo, conforme introducimos el programa del ejemplo 2 del ejercicio 7.18:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999

*** Program loading completed ***
*** Program execution begins  ***
```

Ahora el programa LMS ha sido colocado (es decir cargado) en el arreglo **memory**. A continuación Simpletron ejecutará su programa LMS. La ejecución empieza con la instrucción en la posición **00**, y, al igual que C, continuará en forma secuencial, a menos de que sea dirigido a otra parte del programa, mediante una transferencia de control.

Utilice la variable **accumulator**, para representar el registro de acumulador. Utilice la variable **instructionCounter**, para llevar control de la posición en memoria que contiene la instrucción bajo ejecución. Utilice la variable **operationCode**, para indicar la operación en este momento en ejecución, es decir los dos dígitos a la izquierda de la palabra de instrucción. Utilice la variable **operand**, para indicar la posición en memoria en la cual opera la instrucción actual. Entonces, **operand** son los dígitos más a la derecha de la instrucción en ejecución en ese momento. No ejecute las instrucciones desde la memoria directa. Más bien, transfiera la siguiente instrucción a ejecutarse de la memoria a una variable, llamada

instructionRegister. A continuación "tome" los dos dígitos a la izquierda y colóquelos en **operationCode**, y luego "tome" los dos dígitos a la derecha y colóquelos en **operand**.

Cuando Simpletron empieza a operar, los registros especiales son inicializados como sigue:

```
accumulator          +0000
instructionCounter     00
instructionRegister  +0000
operationCode          00
operand                00
```

Ahora "recorramos" la ejecución de la primera instrucción LMS, **+1009** en la posición de memoria **00**. Esto se conoce como un *ciclo de ejecución de instrucción*.

El **instructionCounter** nos indica la posición de la siguiente instrucción a ejecutarse. Tomamos el contenido de esa posición **memory**, utilizando el enunciado en C

```
istructionRegister = memory[instructionCounter];
```

El código de operación y el operando se extraen del registro de instrucción mediante los enunciados

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Ahora Simpletron debe determinar que el código de operación es de hecho un *read* (a diferencia de un *write*, un *load*, etcétera). Con un **switch** se diferencia entre las doce operaciones de LMS.

En la estructura **switch**, el comportamiento de las varias instrucciones LMS se simula como sigue (dejamos las demás al lector):

read: `scanf("%d", &memory[operand]);`
load: `accumulator = memory[operand];`
add: `accumulator += memory[operand];`
Varias instrucciones de ramificación: analizaremos éstas pronto.
halt:: Esta instrucción imprime el mensaje
`*** Simpletron execution terminated ***`

y a continuación el nombre y el contenido de cada registro, así como el contenido completo de la memoria. Una impresión como ésta, con frecuencia se conoce como un *vaciado de computadora* (y, no, un vaciado de computadora no es un lugar donde va la computadora vieja). Para auxiliarle a programar su función de vaciado, en la figura 7.32 se muestra un formato de muestra de vaciado. Note que después de ejecutar un programa Simpletron un vaciado mostraría los valores reales de las instrucciones y los valores de datos, en el momento en que la ejecución terminó.

Sigamos con la ejecución de la primera instrucción de su programa, es decir, la **+1009** en la posición **00**. Como hemos indicado, el enunciado **switch** simula lo anterior, ejecutando en C el enunciado

```
scanf("%d", &memory[operand]);
```

Antes de que se ejecute **scanf**, deberá aparecer un signo de interrogación (**?**) en pantalla, para solicitarle entradas al usuario. Simpletron espera a que el usuario escriba un valor y a continuación presione la *tecla Return*. El valor a continuación se lee a la posición **09**.

Llegado a ese punto, la simulación de la primera instrucción ha sido terminada. Todo lo que se requiere es preparar Simpletron para que ejecute la siguiente instrucción. Dado que la instrucción que se acaba de terminar no fue una transferencia de control, simplemente necesitamos incrementar el registro de contador de instrucciones, como sigue:

```
++instructionCounter;
```

```
REGISTERS:
accumulator              +0000
instructionCounter         00
instructionRegister      +0000
operationCode              00
operand                    00

MEMORY:
      0     1     2     3     4     5     6     7     8     9
 0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Fig. 7.32 Una muestra de vaciado.

Esto completa la ejecución simulada de la primera instrucción. Empieza otra vez todo el proceso, (es decir, el ciclo de ejecución de instrucción), con la obtención de la siguiente instrucción a ejecutarse.

Ahora veamos como se simulan las instrucciones de ramificación —las transferencias de control. Todo lo que necesitamos hacer es ajustar en forma apropiada el valor del contador de instrucciones. Por lo tanto, la instrucción de ramificación incondicional (**40**) se simula dentro de **switch** como

```
instructionCounter = operand;
```

La instrucción condicional "ramifique si el acumulador es cero" se simula como

```
if (accumulator == 0)
   instructionCounter = operand;
```

Llegado a este punto deberá poner en marcha su simulador Simpletron y ejecutar cada uno de los programas LMS escritos en el ejercicio 7.18. Puede embellecer a LMS con características adicionales e incluirlas en su simulador.

Su simulador deberá verificar varios tipos de errores. Durante la fase de carga de programa, por ejemplo, cada número que escriba el usuario en **memory** del Simpletron debe estar en el rango **-9999** hasta **+9999**. Su simulador deberá utilizar un ciclo **while** y probar que cada uno de los números escritos estén en este rango y, de lo contrario, insistir en solicitarle al usuario que vuelva a escribir el número, hasta que éste lo escriba correcto.

Durante la fase de ejecución, su simulador deberá de vigilar varios errores serios, como intentos de división entre cero, intentos de ejecución inválida de códigos de operaciones, desbordamiento del acumulador (es decir, operaciones aritméticas que resulten en valores mayores que **+9999** o menores **-9999**) y otras similares. Estos errores serios son conocidos como *errores fatales*. Cuando se detecta un error fatal, su simulador deberá imprimir un mensaje de error como:

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

y deberá imprimir un vaciado completo de computadora, en el formato ya analizado. Esto ayudará al usuario a localizar el error dentro del programa.

7.20 Modifique el programa de barajar y distribuir naipes de la figura 7.24, de tal forma que las operaciones de barajar y de reparto sean ejecutados por la misma función (**shuffleAndDeal**). La función deberá contener una estructura de ciclo anidado, similar a la función **shuffle**, de la figura 7.24.

7.21 ¿Qué es lo que efectúa este programa?

```
#include <stdio.h>
void mystery1 (char *, const char *);
main()
{
   char string1[80], string2[80];
   printf("Enter two strings: ");
   scanf("%s%s", string1, string2);
   mystery1(string1, string2);
   printf("%s\n", string1);
   return 0;
}

void mystery1(char *s1, const char *s2)
{
   while (*s1 != '0')
      ++s1;
   for ( ; *s1 = *s2; s1++, s2++)
      ; /* empty statement */
}
```

7.22 ¿Qué es lo que ejecuta este programa?

```
#include <stdio.h>

int mystery2(const char *);

main()
{
   char string[80];

   printf("Enter a string: ");
   scanf("%s", string);
   printf("%d\n", mystery2(string));
   return 0;
}

int mystery2(const char *s)
{
   int x = 0;

for ( ; *s != '\0' s++)
   ++x;

return x;
}
```

7.23 Encuentre el error en cada uno de los segmentos de programa siguientes. Si el error puede ser corregido, explique cómo

a)
```
int *number:
printf("%d\n",*number);
```

b)
```
float *realPtr;
long *integerPtr;
integerPtr = realPtr;
```

c)
```
int * x, y;
x = y;
```
d)
```
char s[] = "this is a character array";
int count;
for ( ; *s != '\0'; s++)
   printf("%c ", *s);
```
e)
```
short *numPtr, result;
void *genericPtr = numPtr;
result = *genericPtr + 7;
```
f)
```
float x 0 19.34;
float xPtr = &x;
printf("%f\n", xPtr);
```
g)
```
char *s;
printf("%s\n", s);
```

7.24 (*Quicksort*). En los ejemplos y ejercicios del capítulo 6, se analizaron técnicas de ordenamiento de tipo burbuja, de tipo cubeta y de tipo selección. Introducimos ahora la técnica de ordenación recursiva conocida como Quicksort. El algoritmo básico, para un conjunto de un subíndice de valores, es como sigue:

1) *Paso de particionamiento:* tome el primer elemento del arreglo no ordenado y determine su posición final en el arreglo ordenado. Esto ocurre cuando todos los valores a la izquierda del elemento del arreglo son menores que el elemento, y todos los valores a la derecha del elemento en el arreglo son mayores que el elemento. Tenemos ahora un elemento en su posición correcta y dos subarreglos no ordenados.

2) *Paso recursivo:* Ejecute el paso 1 para cada uno de los subarreglos no ordenados.

Cada vez que en un subarreglo se ejecuta el paso 1, se coloca otro elemento del arreglo ordenado en su posición final, y se crean dos subarreglos no ordenados. Cuando un subarreglo está formado por un solo elemento, deberá ser ordenado, por lo que dicho elemento aparece en su posición final.

El algoritmo básico parece lo suficiente simple, pero ¿cómo determinaremos la posición final del primer elemento de cada subarreglo? Como un ejemplo, considere el siguiente conjunto de valores (el elemento que aparece en negritas es el elemento particionante —será colocado en su posición final del arreglo ordenado):

37 2 6 4 89 8 10 12 68 45

1) Empezando a partir del elemento más a la derecha del arreglo, compare cada uno de los elementos con **37** hasta que se encuentre un elemento menor que **37** y a continuación intercambie **37** con dicho elemento. El primer elemento menor que **37** es 12, por lo que **37** y 12 son intercambiados. El siguiente arreglo queda así:

12 2 6 4 89 8 10 **37** 68 45

El elemento 12 aparece en itálicas, a fin de indicar que ha sido intercambiado con **37**.

2) Empezando a la izquierda del arreglo, pero comenzando con el elemento siguiente a 12, compare cada uno de los elementos con **37** hasta que se encuentra un elemento mayor que **37**, y a continuación intercambie **37** con dicho elemento. El primer elemento mayor que **37** es 89, por lo que se intercambian **37** y 89. El nuevo arreglo resulta:

12 2 6 4 **37** 8 10 *89* 68 45

3) Empezando a partir de la derecha, pero iniciando con el elemento de inmediato anterior a 89, compare cada uno de los elemento con **37**, hasta que se encuentre un elemento menor que **37**, y a continuación intercambie **37** con dicho elemento. El primer elemento menor de **37** es 10, por lo que **37** y 10 resultarán intercambiados. El nuevo arreglo es:

12 2 6 4 *10* 8 **37** 89 68 45

4) Empezando por la izquierda, pero iniciando con el elemento después de 10, compare cada elemento con **37** hasta que se encuentra un elemento mayor de **37** y a continuación intercambie **37** con dicho elemento. Ya no existen más elementos mayores que **37**, por lo que al comparar **37** consigo mismo sabemos que **37** ha sido colocado en su posición final en el arreglo ordenado.

Una vez que la partición ha sido aplicada en el arreglo anterior, tenemos dos subarreglos sin ordenar. El subarreglo con valores menores de 37 contiene 12, 2, 6, 4, 10 y 8. El subarreglo con valores mayores que 37 contiene 89, 68 y 45. La ordenación continúa con ambos subarreglos, particionándose de la misma forma que el arreglo original.

Basados en el análisis anterior, escriba la función recursiva `quicksort` para ordenar un arreglo entero de un subíndice. La función deberá recibir como argumentos un arreglo entero, un subíndice inicial y un subíndice final. La función `partition` deberá ser llamada por `quicksort` para ejecutar el paso de partición.

7.25 (*Atravesar laberintos*). La siguiente cuadrícula de unos y de ceros es un arreglo de doble subíndice, que representa un laberinto.

```
1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 1 0 0 0 0 0 0 1
0 0 1 0 1 0 1 1 1 1 0 1
1 1 1 0 1 0 0 0 0 1 0 1
1 0 0 0 0 1 1 1 0 1 0 0
1 1 1 1 0 1 0 1 0 1 0 1
1 0 0 1 0 1 0 1 0 1 0 1
1 1 0 1 0 1 0 1 0 1 0 1
1 0 0 0 0 0 0 0 0 1 0 1
1 1 1 1 1 1 0 1 1 1 0 1
1 0 0 0 0 0 0 1 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1
```

Los unos representan los muros del laberinto, y los ceros representan cuadros en las trayectorias posibles a través del mismo.

Existe un algoritmo simple para caminar a través de un laberinto que garantiza encontrar la salida (suponiendo que una exista). Si no existe salida, usted llegará de regreso a la posición inicial. Coloque su mano derecha sobre el muro a su derecha y empiece a caminar hacia adelante. No retire nunca su mano de la pared. Si el laberinto gira a la derecha, usted sigue el muro a la derecha. Siempre que no retire su mano de la pared, de forma eventual llegará a la salida del laberinto. Pudiera existir una trayectoria más corta de la que haya tomado, pero está garantizado de que saldrá del laberinto.

Escriba la función recursiva `mazeTraverse` para caminar a través del laberinto. La función deberá recibir como argumentos un arreglo de 12 por 12 caracteres, que representa el laberinto, y la posición inicial del laberinto. Conforme `mazeTraverse` intenta localizar la salida del laberinto, deberá colocar el carácter `x` en cada uno de los cuadros de la trayectoria. Después de cada movimiento la función deberá mostrar el laberinto, para que el usuario pueda observar conforme resuelve el laberinto.

7.26 (*Cómo generar laberintos en forma aleatoria*). Escriba una función `mazeGenerator`, que toma como argumento un arreglo de 12 por 12 de doble subíndice, y que produce en forma aleatoria un laberinto. La función también deberá proporcionar las posiciones iniciales y finales del mismo. Pruebe su función `mazeTreverse`, del ejercicio 7.25, utilizando varios laberintos generados al azar.

7.27 (*Laberintos de cualquier tamaño*). Generalice las funciones `mazeTreverse` y `mazeGenerator`, de los ejercicios 7.25 y 7.26, para poder procesar laberintos de cualquier ancho y altura.

7.28 (*Arreglos de apuntadores a funciones*). Vuelva a escribir el programa de la figura 6.22 para utilizar una interfaz manejada por menú. El programa deberá darle 4 opciones al usuario, como sigue:

```
Enter a choice:
  0  Print the array of grades
  1  Find the minimum grade
  2  Find the maximum grade
  3  Print the average on all tests for each student
  4  End program
```

Una restricción para el uso de arreglos de apuntadores a funciones es que todos los apuntadores tienen que ser del mismo tipo. Los apuntadores deben ser a funciones con el mismo tipo de regreso y que reciban argumentos del mismo tipo. Por esta razón, deberán ser modificadas las funciones en la figura 6.22, de tal forma que cada una de ellas pueda regresar el mismo tipo y tomar los mismos parámetros. Modifique las funciones **minimum y maximum** para imprimir el valor mínimo o máximo y para que regresen nada. En el caso de la opción 3, modifique la función **average** de la figura 6.22 para que se extraiga el promedio de cada alumno (y no de un alumno en particular). La función **average** debe regresar nada y tomar los mismos parámetros que **printArray**, **minimum** y **maximum**. Almacene los apuntadores a las cuatro funciones en el arreglo **processGrades** y utilice la elección que efectúe el usuario como subíndice del arreglo para llamar a cada una de las funciones.

7.29 (*Modificaciones al simulador Simpletron*). En el ejercicio 7.19, usted escribió una simulación en software de una computadora que ejecuta programas escritos en lenguaje de máquina Simpletron (LMS). En este ejercicio, proponemos varias modificaciones y mejoras al simulador Simpletron. En los ejercicios 12.26 y 12.27, proponemos la construcción de un compilador, que convierta programas escritos en un lenguaje de programación de alto nivel (una variación de BASIC) al lenguaje de máquina Simpletron. Algunas de las modificaciones y mejoras siguientes pudieran ser requeridas para ejecutar los programas producidos por el compilador.

a) Extienda la memoria del simulador Simpletron para que contenga 1000 posiciones de memoria y permitir que Simpletron maneje programas más grandes.
b) Permita que el simulador ejecute cálculos de módulos. Esto requiere de una instrucción adicional en lenguaje de máquina Simpletron.
c) Permita que el simulador ejecute cálculos de exponenciación. Esto requiere de una instrucción adicional en lenguaje de máquina Simpletron.
d) Modifique el simulador para que pueda utilizar valores hexadecimales en vez de valores enteros, para representar instrucciones en lenguaje de máquina Simpletron.
e) Modifique el simulador para que permita la salida de nueva línea. Esto requiere de una instrucción adicional en lenguaje de máquina Simpletron.
f) Modifique el simulador para que pueda procesar valores en punto flotante en adición a valores enteros.
g) Modifique el simulador para manejar entradas de cadenas. *Sugerencia*: cada palabra Simpletron puede ser dividida en dos grupos, cada uno de ellos conteniendo un entero de dos dígitos. Cada entero de dos dígitos puede representar el equivalente decimal ASCII de un carácter. Añada una instrucción en lenguaje máquina que introduzca una cadena y almacene el principio de la cadena en una posición específica de la memoria Simpletron. La primera mitad de la palabra en dicha posición será una cuenta del número de caracteres dentro de la cadena (es decir, la longitud de la cadena). Cada mitad de palabra siguiente contiene un carácter ASCII, expresado en forma de dos dígitos decimales. La instrucción en lenguaje de máquina convierte cada carácter en su equivalente ASCII y lo asigna a dicha media palabra.
h) Modifique el simulador para manejar la salida de cadenas almacenadas en el formato diseñado en la parte (g). *Sugerencia*: añada una instrucción en lenguaje de máquina que imprima una cadena, empezando en cierta posición de memoria Simpletron. La primera parte de la palabra en dicha posición es una cuenta del número de caracteres dentro de la cadena (es decir la longitud de la cadena). Cada media palabra siguiente contiene el carácter ASCII, expresado como dos

dígitos decimales. La instrucción en lenguaje de máquina verifica la longitud e imprime la cadena, traduciendo cada dos números en su carácter equivalente.

7.30 ¿Qué hace este programa?

```
#include <stdio.h>

int mystery3(const char *, const char *);

main()
{
   char string1[80], string2[80];

   printf("Enter two strings: ");
   scanf("%s%s", string1, string2);
   printf("The result is %d\n", mystery3(string1, string2));

   return 0;
}

int mystery3(const char *s1, const char *s2)
{
   for ( ; *s1 != '\0' && *s2 != '\0'; s1++, s2++)

      if (*s1 != *s2)
         return 0;

   return 1;
}
```

8

Caracteres y cadenas

Objetivos

- Ser capaz de utilizar las funciones de la biblioteca de manejo de caracteres (`ctype`).
- Ser capaz de utilizar las funciones de entrada/salida de cadenas y de caracteres de la biblioteca estándar de entrada/salidas (`stdio`).
- Ser capaz de utilizar las funciones de conversión de cadenas de la biblioteca general de utilerías (`stdlib`).
- Ser capaz de utilizar las funciones de procesamiento de cadenas de la biblioteca de manejo de cadenas (`string`).
- Realizar el poder que tienen las bibliotecas de funciones como medio de conseguir reutilización del software.

El defecto principal de Henry King
Era masticar pequeños pedazos de hilo.
Hilaire Belloc

Adecúe la acción a la palabra, y la palabra a la acción.
William Shakespeare

Un escrito vigoroso es conciso. Una oración no debería contener palabras innecesarias, y un párrafo ninguna oración innecesaria.
William Strunk, Jr.

En una concatenación de acuerdo.
Oliver Goldsmith.

Sinopsis

Resumen • Terminología • Errores comunes de programación • Prácticas sanas de programación • Sugerencias de portabilidad • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios • Sección especial: compendio de ejercicios de manipulación de cadenas más avanzados.

8.1 Introducción

En este capítulo, presentamos las funciones estándar de biblioteca de C, que facilitan el procesamiento de cadenas y caracteres. Las funciones le permiten a los programas procesar caracteres, cadenas, líneas de texto y bloques de memoria.

El capítulo analiza las técnicas utilizadas para desarrollar editores, procesadores de palabras, software de disposición de páginas, sistemas de tipografía computarizada, y otros tipos de software de procesamiento de texto. Las manipulaciones de texto con formato ejecutadas por funciones de entrada/salida como **printf** y **scanf** pueden ser puestas en operación, utilizando las funciones analizadas en este capítulo.

8.2 Fundamentos de cadenas y caracteres

Los caracteres son los bloques constructivos fundamentales de los programas fuente. Todo programa está compuesto de una secuencia de caracteres que cuando se agrupa en forma significativa es interpretado por la computadora como una serie de instrucciones utilizadas para llevar a cabo una tarea. Un programa puede contener *constantes de caracteres*. Una *constante de carácter* es un valor **int** representado como un carácter entre comillas sencillas. El valor de una constante de carácter es el valor entero del carácter en el conjunto de caracteres de la máquina. Por ejemplo, **'z'** representa el valor entero de **z**, y **'\n'** representa el valor entero de nueva línea.

Una cadena es una serie de caracteres tratados como una sola unidad. Una cadena puede incluir letras, dígitos y varios *caracteres especiales*, como son **+**, **-**, *****, **/**, **$**, y otros. En C, las *literales de cadena* o *constantes de cadena* se escriben entre dobles comillas, como sigue:

"John Q. Doe"	(un nombre)
"99999 Main Street"	(una dirección de calle)
"Waltham, Massachusetts"	(una ciudad y un estado)
"(201) 555-1212"	(un número telefónico)

En C una cadena es un arreglo de caracteres que termina con el *carácter nulo* (**'\0'**). Se tiene acceso a una cadena mediante un apuntador al primer carácter de la cadena. El valor de una cadena es la dirección de su primer carácter. Por lo tanto, en C, es apropiado decir que una *cadena es un apuntador* —de hecho, un apuntador al primer carácter de una cadena. En este sentido, las cadenas son como los arreglos, porque un arreglo también es un apuntador a su primer elemento.

Una cadena puede ser asignada en una declaración, ya sea a un arreglo de caracteres o a una variable del tipo **char ***. Las declaraciones

```
char color[] = "blue";
char colorPtr = "blue";
```

cada una de ellas inicializa una variable a la cadena **"blue"**. La primera declaración crea un arreglo **color**, de 5 elementos, que contiene los caracteres **'b'**, **'l'**, **'u'**, **'e'** y **'\0'**. La segunda declaración crea la variable de apuntador **colorPtr**, que señala a la cadena **"blue"** en alguna parte de la memoria.

Sugerencia de portabilidad 8.1

Cuando una variable del tipo **char *** *es inicializada con una literal de cadena, algunos compiladores pudieran colocar la cadena en una posición en memoria donde ésta no pueda ser modificada. Si pudiera necesitar modificar una literal de cadena, deberá ser almacenada en un arreglo de caracteres, para asegurarse la modificabilidad en todos los sistemas.*

La declaración de arreglo anterior también podría haberse escrito

```
char color[] = {'b', 'l', 'u', 'e', '\0'};
```

Al declarar un arreglo de caracteres que contenga una cadena, el arreglo debe ser lo suficiente grande para almacenar la cadena y su carácter de terminación **NULL**. La declaración anterior determina el tamaño de la cadena en forma automática, basado en el número de inicializadores en la lista de inicialización.

Error común de programación 8.1

No asignar suficiente espacio en un arreglo de caracteres para almacenar el carácter **NULL** *que da por terminado una cadena.*

Error común de programación 8.2

Imprimir una "cadena" que no contenga un carácter de terminación **NULL**.

Práctica sana de programación 8.1

Al almacenar una cadena de caracteres en un arreglo de caracteres, asegúrese que el arreglo es lo suficientemente grande para contener la cadena más extensa que pueda ser almacenada. C permite que se almacenen cadenas de cualquier longitud. Si una cadena resulta más larga que el arreglo de caracteres en la cual debe almacenarse, los caracteres que excedan del final del arreglo sobreescribirán datos en memoria a continuación del mismo.

Una cadena puede ser asignada a un arreglo utilizando **scanf**. Por ejemplo, el enunciado siguiente asigna una cadena a un arreglo de caracteres **word[20]**.

```
scanf("%s", word);
```

La cadena escrita por el usuario se almacena en **word** (note que **word** es un arreglo que es, naturalmente, un apuntador de tal forma que con el argumento **word** el **&** no se requiere). La función **scanf** leerá caracteres hasta que encuentre un espacio, una nueva línea o un indicador de fin de archivo. Note que la cadena deberá tener una longitud no mayor de 19 caracteres, y así dejar espacio para el carácter de terminación **NULL**. Para que un arreglo de caracteres pueda ser impreso como una cadena, el arreglo debe contener un carácter de terminación **NULL**.

Error común de programación 8.3

Procesar un carácter como si fuera una cadena. Una cadena es un apuntador probablemente un entero respetable grande. Sin embargo, un carácter es un entero pequeño (valores ASCII del rango 0-255). En muchos sistemas, esto causará un error, porque las direcciones bajas de memoria se reservan para fines especiales, como manejadores de interruptores del sistema operativo y, por lo tanto, se incurrirá en "violaciones de acceso".

Error común de programación 8.4

Pasar un carácter como argumento a una función, cuando se espera una cadena.

Error común de programación 8.5

Pasar una cadena como un argumento a una función, cuando se espera un carácter.

8.3 Biblioteca de manejo de caracteres

La biblioteca de manejo de caracteres incluye varias funciones que ejecutan pruebas útiles y manipulaciones de datos de caracteres. Cada función recibe un carácter representado como un **int** o un **EOF** como argumento. Como se analizó en el capítulo 4, a menudo los caracteres son manipulados como enteros, porque en C un carácter es un entero de un byte. Recuerde que **EOF** por lo regular tiene el valor -1 y que algunas arquitecturas de hardware no permiten valores negativos almacenados en variables **char**. Por lo tanto, las funciones de manejo de caracteres manipulan caracteres como enteros. La figura 8.1 resume las funciones de la biblioteca de manejo de caracteres.

Práctica sana de programación 8.2

Al usar funciones de la biblioteca de manejo de caracteres, incluya el archivo de cabecera **<ctype.h>**.

El programa de la figura 8.2 demuestra las funciones ***isdigit***, ***isalpha***, ***isalnum***, *y* ***isxdigit***. La función **isdigit** determina si su argumento es un dígito (**0** al **9**). La función **isalpha** determina si su argumento es una letra en mayúscula (**A** a la **Z**)o una minúscula (**a** a la **z**). La función **isalnum** determina si su argumento es una letra mayúscula, una letra minúscula o un dígito. La función **isxdigit** determina si su argumento es un dígito hexadecimal (**A** a **la F**, **a** a la **f**, **0 al 9**).

El programa de la figura 8.2 utiliza el operador condicional (**?:**) con cada una de las funciones para determinar si en la salida deberá ser impresa la cadena **" is a "** o la cadena **" is not a "** para cada uno de los caracteres probados. Por ejemplo, la expresión

```
isdigit('8') ? "8 is a " : "8 is not a "
```


Prototipo	Descripción de la función
`int isdigit(int c)`	Regresa un valor verdadero si **c** es un dígito, y 0 (falso) de lo contrario.
`int isalpha(int c)`	Si **c** es una letra, regresa un valor verdadero y 0 de lo contrario.
`int isalnum(int c)`	Si **c** es un dígito o una letra, regresa un valor verdadero y 0 de lo contrario.
`int isxdigit(int c)`	Si **c** es un carácter digital hexadecimal, regresa un valor verdadero y 0 de lo contrario. (Vea el Apéndice E, "Sistemas numéricos" para una explicación detallada de los números binarios, octales, decimales y hexadecimales).
`int islower(int c)`	Si **c** es una letra minúscula, regresa un valor verdadero y 0 de lo contrario.
`int isupper(int c)`	Si **c** es una letra mayúscula, regresa un valor verdadero y 0 de lo contrario.
`int tolower(int c)`	Si **c** es una letra mayúscula, **tolower** regresa **c** como una letra minúscula. De lo contrario, **tolower** regresa el argumento sin modificación.
`int toupper(int c)`	Si **c** es una letra minúscula, **toupper** regresa **c** como una letra mayúscula. De lo contrario, **toupper** regresa el argumento sin modificación.
`int isspace(int c)`	Regresa un valor verdadero, si **c** es un carácter de espacio en blanco —nueva línea (**'\n'**), espacio (**' '**), alimentación de forma (**'\f'**), regreso de carro (**'\r'**), tabulador horizontal(**'\t'**), o tabulador vertical (**'\v'**) —y 0 de lo contrario.
`int iscntrl(int c)`	Regresa un valor verdadero si **c** es un carácter de control y 0 de lo contrario.
`int ispunct(int c)`	Regresa un valor verdadero si **c** es un carácter de impresión distinto a un espacio, un dígito, una letra, y 0 de lo contrario.
`int isprint(int c)`	Regresa un valor verdadero si **c** es un carácter de impresión incluyendo el espacio (**' '**), y 0 de lo contrario.
`int isgraph(int c)`	Regresa un valor verdadero si **c** es un carácter de impresión distinto del espacio (**' '**), y 0 de lo contrario.

Fig. 8.1 Resumen de las funciones de biblioteca de manejo de caracteres.

indica que si **'8'** es un dígito, es decir, **isdigit** regresa un valor verdadero (no cero), la cadena **"8 is a "** se imprime, y si **'8'** no es un dígito, es decir **isdigit** regresa **0**, a cadena **"8 is not a "** será impresa.

El programa de la figura 8.3 demuestra las funciones ***islower***, ***isupper***, ***tolower*** y ***toupper***. La función **islower** determina si su argumento es una letra minúscula (**a-z**). La función **isupper** determina si su argumento es una letra mayúscula (**A-Z**). La función **tolower** convierte una letra mayúscula en minúscula, y regresa la letra minúscula. Si el argumento no es una letra mayúscula, **tolower** regresa el argumento sin modificación. La función **toupper** convierte una letra minúscula en mayúscula, y regresa la letra mayúscula. Si el argumento no es una letra minúscula, **toupper** regresa el argumento sin modificación.

```
/* Using functions isdigit, isalpha, isalnum, and isxdigit */
#include <stdio.h>
#include <ctype.h>

main()
{
   printf("%s\n%s%s\n%s%s\n\n", "According to isdigit: ",
       isdigit('8') ? "8 is a " : "8 is not a ", "digit",
       isdigit('#') ? "# is a " : "# is not a ", "digit");
   printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n", "According to isalpha:",
       isalpha('A') ? "A is a " : "A is not a ", "letter",
       isalpha('b') ? "b is a " : "b is not a ", "letter",
       isalpha('&') ? "& is a " : "& is not a ", "letter",
       isalpha('4') ? "4 is a " : "4 is not a ", "letter");
   printf("%s\n%s%s\n%s%s\n%s%s\n\n", "According to isalnum:",
       isalnum('A') ? "A is a " : "A is not a ", "digit or a letter",
       isalnum('8') ? "8 is a " : "8 is not a ", "digit or a letter",
       isalnum('#') ? "# is a " : "# is not a ", "digit or a letter");
   printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n",
       "According to isxdigit:",
       isxdigit('F') ? "F is a " : "F is not a ", "hexadecimal digit",
       isxdigit('J') ? "J is a " : "J is not a ", "hexadecimal digit",
       isxdigit('7') ? "7 is a " : "7 is not a ", "hexadecimal digit",
       isxdigit('$') ? "$ is a " : "$ is not a ", "hexadecimal digit",
       isxdigit('f') ? "f is a " : "f is not a ", "hexadecimal
digit");
   return 0;
}
```

```
According to isdigit:
8 is a digit
# is not digit

According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter

According to isalnum:
A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter

According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is not a hexadecimal digit
```

Fig. 8.2 Cómo utilizar **isdigit**, **isalpha**, **isalnum**, e **isxdigit**.


```
/* Using functions islower, isupper, tolower, toupper */
#include <stdio.h>
#include <ctype.h>

main()
{
   printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
          "According to islower:",
          islower('p') ? "p is a " : "p is not a ",
          "lowercase letter",
          islower('P') ? "P is a " : "P is not a ",
          "lowercase letter",
          islower('5') ? "5 is a " : "5 is not a ",
          "lowercase letter",
          islower('!') ? "! is a " : "! is not a ",
          "lowercase letter");
   printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
          "According to isupper:",
          isupper('D') ? "D is an " : "D is not an ",
          "uppercase letter",
          isupper('d') ? "d is an " : "d is not an ",
          "uppercase letter",
          isupper('8') ? "8 is an " : "8 is not an ",
          "uppercase letter",
          isupper('$') ? "$ is an " : "$ is not an ",
          "uppercase letter");
   printf("%s%c\n%s%c\n%s%c\n%s%c\n",
          "u converted to uppercase is ", toupper('u'),
          "7 converted to uppercase is ", toupper('7'),
          "$ converted to uppercase is ", toupper('$'),
          "L converted to lowercase is ", tolower('L'));
   return 0;
}
```

```
According to islower:
p is a lowercase letter
P is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter

According to a isupper:
D is an a uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter

u converted to uppercase is U
7 converted to uppercase is 7
$ converted to uppercase is $
L converted to lowercase is l
```

Fig. 8.3 Cómo utilizar **islower**, **isupper**, **tolower** y **toupper**.

La figura 8.4 demuestra las funciones ***isspace***, ***iscntrl***, ***ispunct***, ***isprint*** e ***isgraph***. La función **isspace** determina si su argumento es alguno de los siguientes caracteres de espacio en blanco: espacio (' '), alimentación de forma ('\f'), nueva línea ('\n'), retorno de carro ('\r'), tabulador horizontal ('\t'), o tabulador vertical ('\v'). La función **iscntrl** determina si su argumento es alguno de los caracteres de control siguientes: tabulador horizontal ('\t'), tabulador vertical ('\v'), alimentación de forma ('\f'), alerta ('\a'), retroceso ('\b'), retorno de carro ('\r'), o nueva línea ('\n'). La función **ispunct** determina si su argumento es un carácter de impresión distinto de un espacio, de un dígito o de una letra como **$, #, (,), [,], {, }, ;, :,** %, etcétera. La función **isprint** determina si su argumento es un carácter, que puede ser desplegado en pantalla (incluyendo un carácter de espacio). La función **isgraph** prueba los mismos caracteres que **isprint**, sin embargo, el carácter de

```
/* Using functions isspace, iscntrl, ispunct, isprint, isgraph */
#include <stdio.h>
#include <ctype.h>

main()
{
   printf("%s\n%s%s%s\n%s%s%s\n%s%s\n\n", "According to isspace:",
      "Newline", isspace('\n') ? " is a " : " is not a ",
      "whitespace character", "Horizontal tab",
      isspace('\t') ? " is a " : " is not a ",
      "whitespace character",
      isspace('%') ? "% is a " : "% is not a ",
      "whitespace character");
   printf("%s\n%s%s%s\n%s%s\n\n", "According to iscntrl:",
      "Newline", iscntrl('\n') ? " is a " : " is not a ",
      "control character", iscntrl('$') ? "$ is a " : "$ is not a ",
      "control character");
   printf("%s\n%s%s\n%s%s\n%s%s\n\n", "According to ispunct:",
      ispunct(';') ? "; is a " : "; is not a ",
      "punctuation character",
      ispunct('Y') ? "Y is a " : "Y is not a ",
      "punctuation character",
      ispunct('#') ? "# is a " : "# is not a ",
      "punctuation character");
   printf("%s\n%s%s\n%s%s%s\n\n", "According to isprint:",
      isprint('$') ? "$ is a " : "$ is not a ", "printing character",
      "Alert", isprint('\a') ? " is a " : " is not a ",
      "printing character");
   printf("%s\n%s%s\n%s%s%s\n", "According to isgraph:",
      isgraph('Q') ? "Q is a " : "Q is not a ",
      "printing character other than a space",
      "Space", isgraph(' ') ? " is a " : " is not a ",
      "printing character other than a space");
   return 0;
}
```

Fig. 8.4 Cómo utilizar **isspace**, **iscntrl**, **inspunct**, **isprint** e **isgraph** (Parte 1 de 2).


```
According to isspace:
Newline is a whitespace character
Horizontal tab is a whitespace character
% is not a whitespace character

According to iscntrl:
Newline is a control character
$ is not a control character

According to ispunct:
; is a punctuation character
Y is not a punctuation character
# is a punctuation character

According to isprint:
$ is a printing character
Alert is not a printing character

According to isgraph:
Q is a printing character other than a space
Space is not a printing character other than a space
```

Fig. 8.4 Cómo utilizar **isspace**, **iscntrl**, **ispunct**, **isprint** e **isgraph** (Parte 2 de 2).

8.4 Funciones de conversión de cadenas

Esta sección presenta las *funciones de conversión de cadenas* correspondientes a la *biblioteca general de utilerías* (***stdlib***). Estas funciones convierten cadenas de dígitos a enteros y valores de punto flotante. La figura 8.5 resume las funciones de conversión de cadenas. Note la utilización de **const** para declarar la variable **nPtr** en los encabezados de función (léase de derecha a izquierda como "**nPtr** es un apuntador a la constante de carácter"); **const** declara que el valor del argumento no será modificado.

Prototipo de función	Descripción de la función
`double atof(const char *nPtr)`	Convierte la cadena `nPtr` a `double`.
`int atoi(const char *nPtr)`	Convierte la cadena `nPtr` a `int`.
`long atol(const char *nPtr,)`	Convierte la cadena `nPtr` a `int` a largo.
`double strtod(const char *nPtr, char **endPtr)`	Convierte la cadena `nPtr` a double.
`long strtol(const char *nPtr, char **endPtr, int base)`	Convierte la cadena `nPtr` a `long`.
`unsigned long strtoul(const char *nPtr, char **endPtr, int base)`	Convierte la cadena `nPtr` a `unsigned long`.

Fig. 8.5 Resumen de las funciones de conversión de cadenas de la biblioteca general de utilerías.

Práctica sana de programación 8.3

Al utilizar funciones de la biblioteca general de utilerías, incluya el archivo de cabecera `<stdlib.h>`

La función **`atof`** (figura 8.6) convierte su argumento —una cadena que represente un número en punto flotante a un valor **`double`**. La función regresa un valor **`double`**. Si el valor convertido no puede ser representado —por ejemplo, si el primer carácter de la cadena no es un dígito— el comportamiento de la función **`atof`** quedará indefinida.

La función **`atoi`** (figura. 8.7) convierte su argumento —una cadena de dígitos que representa a un entero— a un valor **`int`**. La función regresa un valor **`int`**. Si el valor convertido no puede ser representado, el comportamiento de la función **`atoi`** quedará indefinido.

La función **`atol`** (figura 8.8) convierte su argumento —una cadena de dígitos representando un entero largo— a un valor **`long`**. La función regresa un valor **`long`**. Si el valor convertido no puede ser representado, el comportamiento de la función **`atol`** quedará indefinido. Si **`int`** y **`long`** ambos están almacenados en 4 bytes, la función **`atoi`** y la función **`atol`** funcionan en forma idéntica.

La función **`strtod`** (figura 8.9) convierte una secuencia de caracteres representando un valor en punto flotante a **`double`**. La función recibe dos argumentos —una cadena (**`char*`**) y un apuntador a una cadena. La cadena contiene la secuencia de caracteres a ser convertida a **`double`**. El apuntador se asigna la posición del primer carácter después de la porción convertida de la cadena. El enunciado

```
d = strtod(string, &stringPtr);
```

del programa de la figura 8.9 indica que **`d`** es asignado el valor **`double`** convertido de **`string`**, y **`&stringPtr`** es asignado la posición del primer carácter después del valor convertido (**`51.2`**) en **`string`**.

```
/* Using atof */
#include <stdio.h>
#include <stdlib.h>

main()
{
   double d;

   d = atof("99.0");
   printf("%s%.3f\n%s%.3f\n",
          "The string \"99.0\" converted to double is ", d,
          "The converted value divided by 2 is ", d / 2.0);
   return 0;
}
```

```
The string "99.0" converted to double is 99.000
The converted value divided by 2 is 49.500
```

Fig. 8.6 Cómo utilizar **`atof`**.


```
/* Using atoi */
#include <stdio.h>
#include <stdlib.h>

main()
{
   int i;

   i = atoi("2593");
   printf("%s%d\n%s%d\n",
          "The string \"2593\" converted to int is ", i,
          "The converted value minus 593 is ", i - 593);
   return 0;
}
```

```
The string "2593" converted to int is 2593
The converted value minus 593 is 2000
```

Fig. 8.7 Cómo utilizar **`atoi`**.

```
/* Using atol */
#include <stdio.h>
#include <stdlib.h>

main()
{
   long l;

   l = atol("1000000");
   printf("%s%ld\n%s%ld\n",
       "The string \"1000000\" converted to long int is ", l,
       "The converted value divided by 2 is ", l / 2);
   return 0;
}
```

```
The string "1000000" converted to long int is 1000000
The converted value divided by 2 is 500000
```

Fig. 8.8 Cómo utilizar **`atol`**.

La función **`strtol`** (figura 8.10) convierte a **`long`** una secuencia de caracteres representando un entero. La función recibe tres argumentos —una cadena (**`char *`**), un apuntador a una cadena y un entero. La cadena contiene la secuencia de caracteres a convertirse. El apuntador se asigna la posición del primer carácter después de la porción convertida de la cadena. El entero especifica la *base* del valor bajo conversión. El enunciado

```
x = strtol(string, &remainderPtr, 0);
```

```
/* Using strtod */
#include <stdio.h>
#include <stdlib.h>

main()
{
   double d;
   char *string = "51.2% are admitted";
   char *stringPtr;

   d = strtod(string, &stringPtr);
   printf("The string \"%s\" is converted to the\n",
          string);
   printf("double value %.2f and the string \"%s\"\n",
          d, stringPtr);

   return 0;
}
```

```
The string "51.2% are admitted" is converted to the
double value 51.20 and the string "% are admitted"
```

Fig. 8.9 Cómo utilizar **strtod**.

en el programa de la figura 8.10 se indica que **x** es asignado el valor **long** convertido del **string**. El segundo argumento, **&remainderPtr**, se asigna el resto de **string** después de la conversión. Si se usa **NULL** como segundo argumento se hará que se ignore el resto de la cadena. El tercer argumento, **0**, indica que el valor a convertirse puede ser en base octal (base 8), decimal (base 10), o hexadecimal (base 16). La base puede ser especificada como **0** o cualquier valor entre 2 y 36. Vea el Apéndice E, "Sistemas numéricos", para una explicación detallada de los sistemas octal, decimal y hexadecimal. Las representaciones numéricas de enteros de la base 11 hasta la base 36 utilizan los caracteres A a la Z para representar los valores del 10 al 35. Por ejemplo, los valores hexadecimales pueden consistir de los dígitos 0 al 9 y de los caracteres A a la F. Un entero de base 11 puede consistir de los dígitos 0 al 9 y el carácter A. Un entero de base 24 puede consistir de los dígitos 0-9 y de los caracteres A-N. Un entero en base 36 puede consistir de los dígitos 0 al 9 y de los caracteres A a la Z.

La función ***strtoul*** (figura 8.11) convierte una secuencia de caracteres a **unsigned long** representando un entero **unsigned long**. La función opera en forma idéntica a la función **strtol**. El enunciado

```
x = strtoul(string, &remainderPtr, 0);
```

en el programa de la figura 8.11 indica que **x** es asignado el valor **unsigned long** convertido de **string**. El segundo argumento, **&remainderPtr**, es asignado al resto de **string**, después de la conversión. El tercer argumento, 0, indica que el valor a ser convertido puede estar en formato octal, decimal o hexadecimal.


```
/* Using strtol */
#include <stdio.h>
#include <stdlib.h>

main()
{
   long x;
   char *string = "-1234567abc", *remainderPtr;

   x = strtol(string, &remainderPtr, 0);
   printf("%s\"%s\"\n%s%ld\n%s\"%s\"\n%s%ld\n",
       "The original string is ", string,
       "The converted value is ", x,
       "The remainder of the original string is ",
       remainderPtr,
       "The converted value plus 567 is ", x + 567);
   return 0;
}
```

```
The original string is "-1234567abc"
The converted value is -1234567
The remainder of the original string is "abc"
The converted value plus 567 is -1234000
```

Fig. 8.10 Cómo utilizar **strtol**.

```
/* Using strtoul */
#include <stdio.h>
#include <stdlib.h>

main()
{
   unsigned long x;
   char *string = "1234567abc", *remainderPtr;

   x = strtoul(string, &remainderPtr, 0);
   printf("%s\"%s\"\n%s%lu\n%s\"%s\"\n%s%lu\n",
          "The original string is ", string,
          "The converted value is ", x,
          "The remainder of the original string is ",
          remainderPtr,
          "The converted value minus 567 is ", x - 567);
   return 0;
}
```

```
The original string is "1234567abc"
The converted value is 1234567
The remainder of the original string is "abc"
The converted value minus 567 is 1234000
```

Fig. 8.11 Cómo utilizar **strtoul**.

8.5 Funciones de la biblioteca estándar de entrada/salida

Esta sección presenta varias funciones de la biblioteca estándar de entrada/salida (**stdio**), específicas para manipular datos de caracteres y de cadenas. La figura 8.12 resume las funciones de entrada y salida de caracteres y de cadenas de la biblioteca estándar de entrada/salidas.

Práctica sana de programación 8.4

Al utilizar funciones de la biblioteca estándar de entrada/salida, incluya el archivo de cabecera **<stdio.h>**.

El programa de la figura 8.13 utiliza la función ***gets*** y ***putchar*** para leer una línea de texto de la entrada estándar (teclado), y sacar en forma recursiva los caracteres de la línea en orden inverso. La función **gets** lee caracteres de la entrada estándar hacia su argumento—un arreglo del tipo **char**— hasta que se encuentra con un carácter de nueva línea o un indicador de fin de archivo. Cuando se termina la lectura, se agrega al arreglo un carácter **NULL** (**'\0'**). La función **putchar** imprime su argumento de carácter. El programa llama la función recursiva **reverse**, para imprimir al revés la línea de texto. Si el primer carácter del arreglo recibido por **reverse** es el carácter nulo (**NULL**) **'\0'** **reverse** regresa. De lo contrario, **reverse** vuelve a ser llamada con la dirección del subarreglo empezando con el elemento **s[1]**, y el carácter **s[0]** es sacado mediante **putchar**, cuando la llamada recursiva ha terminado. El orden de los dos enunciados en la porción **else** de la estructura **if** hace que **reverse** avance hacia el carácter de terminación **NULL** de la cadena, antes de que se imprima un carácter. Conforme se completan las llamadas recursivas, los caracteres son sacados en orden inverso.

El programa de la figura 8.14 utiliza las funciones ***getchar*** y ***puts*** para leer caracteres de la entrada estándar al arreglo de caracteres **sentence**, e imprime el arreglo de caracteres como una cadena. La función **getchar** lee un carácter de la entrada estándar y regresa el carácter como un entero. La función **puts** toma una cadena (**char ***) como argumento, e imprime la cadena seguida por un carácter de nueva línea.

Función prototipo	Descripción de la función
`int getchar(void)`	Introduce el siguiente carácter de la entrada estándar y lo regresa como un entero.
`char *gets(char *s)`	Introduce caracteres de la entrada estándar en el arreglo s hasta que encuentra un carácter de nueva línea o de terminación de archivo. Se agrega al arreglo un carácter de terminación NULL.
`int putchar(int c)`	Imprime el carácter almacenado en c.
`int puts(const char *s)`	Imprime la cadena s seguida por un carácter de nueva línea.
`int sprintf(char *s, const char *format, ...)`	Equivalente a **printf**, excepto que la salida se almacena en el arreglo s, en vez de imprimir a la pantalla.
`int sscanf(char *s, const char *format, ...)`	Equivalente a **scanf**, excepto que la entrada se lee del arreglo s, en vez de leerlo desde el teclado.

Fig. 8.12 Funciones de caracteres y cadenas de la biblioteca estándar de entrada/salida.


```
/* Using gets and putchar */
#include <stdio.h>

main()
{
   char sentence[80];
   void reverse(char *);

   printf("Enter a line of text:\n");
   gets(sentence);

   printf("\nThe line printed backwards is:\n");
   reverse(sentence);

   return 0;
}

void reverse(char *s)
{
   if (s[0] == '\0')
      return;
   else {
      reverse(&s[1]);
      putchar(s[0]);
   }
}
```

```
Enter a line of text:
Characters and Strings

The line printed backwards is:
sgnirtS dna sretcarahC
```

```
Enter a line of text:
able was I ere I saw elba

The line printed backwards is:
able was I ere I saw elba
```

Fig. 8.13 Cómo usar **gets** y **putchar**.

El programa deja de introducir caracteres, cuando **getchar** lee el carácter de nueva línea, escrito por el usuario para terminar la línea de texto. Se agrega una carácter **NULL** al arreglo **sentence**, de tal forma que el arreglo pueda ser tratado como una cadena. La función **puts** imprime la cadena contenida en **sentence**.

UNIVERSIDAD DE LA REPUBLICA
FACULTAD DE INGENIERIA
DEPARTAMENTO DE
DOCUMENTACION Y BIBLIOTECA
MONTEVIDEO - URUGUAY

```
/* Using getchar and puts */
#include <stdio.h>

main()
{
   char c, sentence[80];
   int i = 0;

   puts("Enter a line of text:");
   while ( ( c = getchar() ) != '\n')
      sentence[i++] = c;

   sentence[i] = '\0';  /* insert NULL at end of string */
   puts("\nThe line entered was:");
   puts(sentence);
   return 0;
}
```

```
Enter a line of text:
This is a test.

The line entered was:
This is a test.
```

Fig. 8.14 Cómo utilizar `getchar` y `puts`.

El programa de la figura 8.15 utiliza la función ***sprintf*** para imprimir datos con formato al arreglo **s** —un arreglo de caracteres. La función usa las mismas especificaciones de conversión que utiliza **printf** (vea el capítulo 9 para un análisis detallado de todas las características de formato de impresión). El programa introduce un valor **int** y un valor **float** al arreglo s para dársele formato y para ser impreso. El arreglo **s** es el primer argumento de **sprintf**.

```
/* Using sprintf */
#include <stdio.h>

main()
{
   char s[80];
   int x;
   float y;

   printf("Enter an integer and a float:\n");
   scanf("%d%f", &x, &y);
   sprintf(s, "Integer:%6d\nFloat:%8.2f", x, y);
   printf("%s\n%s\n",
          "The formatted output stored in array s is:", s);
   return 0;
}
```

Fig. 8.15 Cómo utilizar **sprintf** (parte 1 de 2).


```
Enter an integer and a float:
298  87.375
The formatted output stored in array s is:
Integer:     298
Float:     87.38
```

Fig. 8.15 Cómo utilizar **sprintf** (parte 2 de 2).

El programa de la figura 8.16 utiliza la función ***scanf*** para leer datos con formato del arreglo de caracteres **s**. La función utiliza las mismas especificaciones de conversión que **scanf**. El programa lee un **int** y un **float** del arreglo **s**, y almacena los valores en **x** y en **y**, respectivamente. Los valores de **x** y de **y** son impresos. El arreglo **s** es el primer argumento de **sscanf**.

8.6 Funciones de manipulación de cadenas de la biblioteca de manejo de cadenas

La biblioteca de manejo de cadenas proporciona muchas funciones útiles para manipular datos de cadenas, comparar cadenas, buscar en cadenas caracteres y otras cadenas, dividir cadenas (separar cadenas en partes lógicas) y determinar la longitud de las mismas. Esta sección presenta las funciones de manipulación de cadenas de la biblioteca de manejo de cadenas. Las funciones se resumen en la figura 8.17.

Práctica sana de programación 8.5

Al utilizar funciones de la biblioteca de manejo de cadenas, incluya el archivo de cabecera ***<string.h>*** *.*

```
/* Using sscanf */
#include <stdio.h>

main()
{
   char s[] = "31298 87.375";
   int x;
   float y;

   sscanf(s, "%d%f", &x, &y);
   printf("%s\n%s%6d\n%s%8.3f\n",
          "The values stored in character array s are:",
          "Integer:", x, "Float:", y);
   return 0;
}
```

```
The values stored in character array s are:
Integer: 31298
Float:  87.375
```

Fig. 8.16 Cómo utilizar **sscanf**.

Prototipo de función	Descripción de la función
`char *strcpy(char *s1, const char *s2)`	Copia la cadena `s2` al arreglo `s1`. Es regresado el valor de `s1`.
`char *strncpy(char *s1, const char *s2, size_t n)`	Copia en la mayor parte de `n` caracteres de la cadena `s2` en el arreglo `s1`. Es regresado el valor de `s1`.
`char *strcat(char *s1, const char *s2)`	Agrega la cadena `s2` al arreglo `s1`. El primer carácter de `s2` sobreescribe el carácter de terminación `NULL` de `s1`. Es regresado el valor de `s1`.
`char *strncat(char *s1, const char *s2, size_t n)`	Agrega como máximo `n` caracteres de la cadena `s2` al arreglo `s1`. El primer carácter de `s2` sobreescribe el carácter de terminación `NULL` de `s1`. Es regresado el valor de `s1`.

Fig. 8.17 Funciones de manipulación de cadenas de la biblioteca de manejo de cadenas.

La función ***strcpy*** copia su segundo argumento —una cadena en su primer argumento, un arreglo de caracteres que debe ser lo suficiente grande para almacenar la cadena y su carácter de terminación **NULL**, que también debe ser copiado. La función ***strncpy*** es equivalente a **strcpy**, excepto que **strncpy** especifica el número de caracteres a copiarse de la cadena al arreglo. Note que la función **strncpy** no necesariamente copia el carácter de terminación **NULL** de su segundo argumento. Un carácter de terminación **NULL** será escrito, sólo si el número de caracteres a copiarse es por lo menos uno más que la longitud de la cadena. Por ejemplo, si el segundo argumento es **"test"**, se escribirá un carácter de terminación **NULL** si el tercer argumento a **strncpy** es por lo menos **5** (los 4 caracteres en **"test"** más 1 carácter de terminación **NULL**). Si el tercer argumento es mayor que **5**, al arreglo se agregarán caracteres **NULL**, hasta que quede escrito el número total de caracteres especificado por el tercer argumento.

Error común de programación 8.6

No agregar un carácter de terminación **NULL** *al primer argumento de un* **strncpy**, *cuando el tercer argumento es menor que o igual a la longitud de la cadena del segundo argumento.*

El programa de la figura 8.18 utiliza a **strcpy** para copiar toda la cadena del arreglo **x** al arreglo **y**, y después utiliza **strncpy** para copiar los primeros **14** caracteres del arreglo **x** al arreglo **z**. Se agrega al arreglo **z** un carácter **NULL** (**'\0'**), porque la llamada a **strncpy** en el programa no escribe un carácter de terminación **NULL** (el tercer argumento tiene una longitud menor que la longitud de la cadena del segundo argumento).

La función ***strcat*** agrega su segundo argumento —un cadena— a su primer argumento —un arreglo de caracteres que contiene una cadena. El primer carácter del segundo argumento remplaza el **NULL** (**'\0'**) que termina la cadena del primer argumento. El programador deberá asegurarse que el arreglo utilizado para almacenar la primera cadena es lo suficiente extensa para almacenar dicha primera cadena, la segunda cadena y el carácter de terminación **NULL** (copiado


```
/* Using strcpy and strncpy */
#include <stdio.h>
#include <string.h>

main()
{
   char x[] = "Happy Birthday to You";
   char y[25], z[15];

   printf("%s%s\n%s%s\n",
          "The string in array x is: ", x,
          "The string in array y is: ", strcpy(y, x));

   strncpy(z, x, 14);
   z[14] = '\0';
   printf("The string in array z is: %s\n", z);
   return 0;
}
```

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

Fig. 8.18 Cómo utilizar **strcpy** y **strncpy**.

de la segunda cadena). La función ***strncat*** agrega un número especificado de caracteres de la segunda cadena a la primera cadena. Al resultado de forma automática se le agrega un carácter de terminación **NULL**. El programa de la figura 8.19 demuestra la función **strcat** y la función **strncat**.

```
/* Using strcat and strncat */
#include <stdio.h>
#include <string.h>

main()
{
   char s1[20] = "Happy ";
   char s2[] = "New Year ";
   char s3[40] = "";

   printf("s1 = %s\ns2 = %s\n", s1, s2);
   printf("strcat(s1, s2) = %s\n", strcat(s1, s2));
   printf("strncat(s3, s1, 6) = %s\n", strncat(s3, s1, 6));
   printf("strcat(s3, s1) = %s\n", strcat(s3, s1));
   return 0;
}
```

Fig. 8.19 Cómo utilizar **strcat** y **strncat** (parte 1 de 2).

```
s1 = Happy
s2 = New Year
strcat (s1, s2) = Happy New Year
strncat (s3, s1, 6) = Happy
strcat (s3, s1) = Happy Happy New Year
```

Fig. 8.19 Cómo utilizar `strcat` y `strncat` (parte 2 de 2).

8.7 Funciones de comparación de la biblioteca de manejo de cadenas

En esta sección se estudian las funciones de comparación de cadenas, **strcmp** y **strncmp**, de la biblioteca de manejo de cadenas. En la figura 8.20 aparecen los encabezados de función y una breve descripción de cada una de las funciones.

El programa de la figura 8.21 compara tres cadenas utilizando las funciones ***strcmp*** y ***strncmp***. La función **strcmp** compara su primer argumento de cadena con su segundo argumento de cadena, carácter por carácter. Si las cadenas son iguales la función regresa 0, un valor negativo si la primera cadena es menor que la segunda cadena, y un valor positivo si la primera cadena es mayor que la segunda cadena. La función **strncmp** es equivalente a **strcmp**, excepto que **strncmp** compara sólo hasta un número especificado de caracteres. La función **strncmp** no compara en una cadena caracteres después de un carácter **NULL**. El programa imprime el valor entero regresado por cada llamada de función.

Error común de programación 8.7

Suponiendo que `strcmp` *y* `strncmp` *regresan 1 cuando sus argumentos son iguales. Ambas funciones regresan 0 (valor falso en C) en caso de igualdad. Por lo tanto, cuando se prueben dos cadenas buscando igualdad, para determinar si las cadenas son iguales, el resultado de la función* `strcmp` *o la* `strncmp` *deberá ser comparado con 0.*

Prototipo de función	Descripción de la función
`int strcmp(const char *s1, const char *s2)`	Compara la cadena `s1` con la cadena `s2`. La función regresa 0, menos que 0, o mayor que 0, si `s1` es igual a, menor que o mayor que `s2`, respectivamente.
`int strncmp(const char *s1, const char *s2, size_t n)`	Compara hasta `n` caracteres de la cadena `s1` con la cadena `s2`. La función regresa 0, menos que 0, o mayor que 0, si `s1` es igual a, menor que o mayor que `s2`, respectivamente.

Fig. 8.20 Funciones de comparación de cadenas de la biblioteca de manejo de cadenas.


```
/* Using strcmp and strncmp */
#include <stdio.h>
#include <string.h>

main()
{
   char *s1 = "Happy New Year";
   char *s2 = "Happy New Year";
   char *s3 = "Happy Holidays";

   printf("%s%s\n%s%s\n%s%s\n\n%s%2d\n%s%2d\n%s%2d\n\n",
          "s1 = ", s1, "s2 = ", s2, "s3 = ", s3,
          "strcmp(s1, s2) = ", strcmp(s1, s2),
          "strcmp(s1, s3) = ", strcmp(s1, s3),
          "strcmp(s3, s1) = ", strcmp(s3, s1));

   printf("%s%2d\n%s%2d\n%s%2d\n",
          "strncmp(s1, s3, 6) = ", strncmp(s1, s3, 6),
          "strncmp(s1, s3, 7) = ", strncmp(s1, s3, 7),
          "strncmp(s3, s1, 7) = ", strncmp(s3, s1, 7));
   return 0;
}
```

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Hollidays

strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  1
strncmp(s3, s1, 7) = -1
```

Fig. 8.21 Cómo utilizar `strcmp` y `strncmp`.

Para comprender con exactitud el significado de que una cadena sea "mayor que" o "menor que" otra cadena, considere el proceso de alfabetizar una serie de apellidos. El lector colocaría, sin duda "Jones" antes de "Smith" porque la primera letra de "Jones" viene en el alfabeto antes de la primera letra de "Smith". Pero el alfabeto es más que una lista de 26 letras —es una lista ordenada de caracteres. Cada letra está en posición específica dentro de la lista. "Z" es más que una letra del alfabeto; "Z" específicamente es la veintiseisava letra del alfabeto.

¿Cómo sabe la computadora que una letra en particular viene antes de otra? Todos los caracteres se representan dentro de la computadora como códigos numéricos; cuando la computadora compara dos cadenas, de hecho está comparando los códigos numéricos de los caracteres en las cadenas.

Sugerencia de portabilidad 8.2

Los códigos numéricos internos utilizados para representar caracteres pudieran ser distintos en diferentes computadoras.

En un esfuerzo para estandarizar representaciones de caracteres, la mayor parte de los fabricantes de computadoras han diseñado sus máquinas para utilizar uno de dos esquemas populares de codificación *ASCII* o *EBCDIC*. ASCII significa "American Standard Code for Information Interchange", y EBCDIC significa "Extended Binary Coded Decimal Interchange Code". Existen otros esquemas de codificación, pero éstos son los dos más populares.

ASCII y EBCDIC se conocen como *códigos de caracteres* o *conjuntos de caracteres*. Las manipulaciones de cadenas y de caracteres, de hecho, involucran la manipulación de los códigos numéricos apropiados y no de los caracteres mismos. Esto explica la intercambiabilidad en C entre caracteres y pequeños enteros. Dado que tiene sentido decir que un código numérico es mayor que, menor que o igual a otro código numérico, es posible relacionar varios caracteres o cadenas, una con la otra, refiriéndose a los códigos de caracteres. El Apéndice D contiene una lista de los códigos de caracteres ASCII.

8.8 Funciones de búsqueda de la biblioteca de manejo de cadenas

Esta sección presenta las funciones de biblioteca de manejo de cadenas, que se utilizan para buscar caracteres y otras cadenas, dentro de cadenas. Las funciones se resumen en la figura 8.22. Note que las funciones **strcspn** y **strspn** especifican un regreso del tipo **size_t**. El tipo **size_t** es un tipo definido por la norma como el tipo integral del valor regresado por el operador **sizeof**.

Sugerencia de portabilidad 8.3

El tipo **size_t** *es un sinónimo, dependiente del sistema, para el tipo* **unsigned long** *o para el tipo* **unsigned int**.

Prototipo de función	Descripción de la función
`char *strchr(const char *s, int c)`	Localiza la primera instancia del carácter **c** en la cadena **s**. Si encuentra **c**, regresa un apuntador a **c**. De lo contrario, regresa un apuntador **NULL**.
`size_t strcspn(const char *s1, const char *s2)`	Determina y regresa la longitud del segmento inicial de la cadena **s1**, consistiendo de caracteres no contenidos en la cadena **s2**.
`size_t strspn(const char *s1, const char *s2)`	Determina y regresa la longitud del segmento inicial de la cadena **s1**, que consiste sólo de los caracteres contenidos en la cadena **s2**.

Fig. 8.22 Funciones de manipulación de cadenas de la biblioteca de manejo de cadenas (parte 1 de 2)

Prototipo de función	Descripción de la función
`char *strpbrk(const char *s1, const char *s2)`	Localiza la primera ocurrencia en la cadena **s1** de cualquier carácter de la cadena **s2**. Si encuentra un carácter de la cadena **s2**, regresa un apuntador al carácter en la cadena **s1**. De lo contrario, regresa un apuntador **NULL**.
`char *strrchr(const char *s, int c)`	Localiza la última instancia de **c** en la cadena **s**. Si encuentra **c**, regresa un apuntador a **c** en la cadena **s**. De lo contrario, regresa un apuntador **NULL**.
`char *strstr(const char *s1, const char *s2)`	Localiza la primera ocurrencia en la cadena **s1** de la cadena **s2**. Si la cadena es hallada, regresa un apuntador a la cadena en **s1**. De lo contrario, regresa un apuntador **NULL**.
`char *strtok(char *s1, const char *s2)`	Una secuencia de llamadas a **strtok** divide la cadena **s1** en "tokens" —partes lógicas, como palabras, en una línea de texto— separados por caracteres, contenidos en la cadena **s2**. La primera llamada contiene **s1** como primer argumento, y las llamadas subsecuentes, para continuar esta división de la misma cadena, contienen **NULL** como primer argumento. Para cada llamada se regresa un apuntador al token o ficha actual. Si cuando la función es llamada ya no hay más tokens o fichas, se regresará **NULL**.

Fig. 8.22 Funciones de manipulación de cadenas de la biblioteca de manejo de cadenas (parte 2 de 2)

La función ***strchr*** busca la primera ocurrencia de un carácter en una cadena. Si encuentra dicho carácter, **strchr** regresa un apuntador al carácter en la cadena, de lo contrario **strchr** regresa **NULL**. El programa de la figura 8.23 utiliza **strchr** para buscar las primeras ocurrencias de **'a'** y **'z'** en la cadena **"This is a test"**.

La función ***strcspn*** (figura 8.24) determina la longitud de la parte inicial de la cadena en su primer argumento, que no contenga ningún carácter de la cadena existente en su segundo argumento. La función regresa la longitud del segmento.

La función ***strpbrk*** busca la primera ocurrencia, en su primer argumento de cadena, de cualquier carácter existente en su segundo argumento de cadena. Si encuentra un carácter del segundo argumento, **strpbrk** regresa un apuntador al carácter en el primer argumento, de lo contrario **strpbrk** regresa **NULL**. El programa de la figura 8.25 localiza la primera ocurrencia en **string1**, de cualquier carácter existente en **string2**.

La función ***strrchr*** busca la última ocurrencia de un carácter especificado en una cadena. Si encuentra dicho carácter, **strrchr** regresa un apuntador al carácter en la cadena, de lo contrario **strrchr** regresa **NULL**. El programa de la figura 8.26 busca la última ocurrencia del carácter **'z'** en la cadena **"A zoo has many animals including zebras"**.

```
/* Using strchr */
#include <stdio.h>
#include <string.h>

main()
{
   char *string = "This is a test";
   char character1 = 'a', character2 = 'z';

   if (strchr(string, character1) != NULL)
      printf("\'%c\' was found in \"%s\".\n",
             character1, string);
   else
      printf("\'%c\' was not found in \"%s\".\n",
             character1, string);

   if (strchr(string, character2) != NULL)
      printf("\'%c\' was found in \"%s\".\n",
             character2, string);
   else
      printf("\'%c\' was not found in \"%s\".\n",
             character2, string);
   return 0;
}
```

```
'a' was found in "This is a test".
'z' was not found in "This is a test".
```

Fig. 8.23 Cómo utilizar **strchr**.

```
/* Using strcspn */
#include <stdio.h>
#include <string.h>

main()
{
   char *string1 = "The value is 3.14159";
   char *string2 = "1234567890";

   printf("%s%s\n%s%s\n\n%s\n%s%u",
          "string1 = ", string1, "string2 = ", string2,
          "The length of the initial segment of string1",
          "containing no characters from string2 = ",
          strcspn(string1, string2));
   return 0;
}
```

```
string1 = The value is 3.14159
string2 = 1234567890

The length of the initial segment of string1
containing no characters from string2 = 13
```

Fig. 8.24 Cómo utilizar **strcspn**.


```
/* Using strpbrk */
#include <stdio.h>
#include <string.h>

main()
{
   char *string1 = "This is a test";
   char *string2 = "beware";

   printf("%s\"%s\"\n'%c'%s\n\"%s\"\n",
          "Of the characters in ", string2,
          *strpbrk(string1, string2),
          " is the first character to appear in ", string1);
   return 0;
}
```

```
Of the characters in "beware"
'a' is the first character to appear in
"This is a test"
```

Fig. 8.25 Cómo utilizar **strpbrk**.

```
/* Using strrchr */
#include <stdio.h>
#include <string.h>

main()
{
   char *string1 = "A zoo has many animals including zebras";
   int c = 'z';

   printf("%s\n%s'%c'%s\"%s\"\n",
          "The remainder of string1 beginning with the",
          "last occurrence of character ", c,
          " is: ", strrchr(string1, c));
   return 0;
}
```

```
The remainder of string1 beginning with the
last occurrence of character 'z' is: "zebras"
```

Fig. 8.26 Cómo utilizar **strrchr**.

La función ***strspn*** (figura 8.27) determina la longitud de la parte inicial de la cadena en su primer argumento, que sólo contiene caracteres de la cadena existente en su segundo argumento. La función regresa la longitud del segmento.

```
/* Using strspn */
#include <stdio.h>
#include <string.h>

main()
{
   char *string1 = "The value is 3.14159";
   char *string2 = "aehilsTuv ";

   printf("%s%s\n%s%s\n\n%s\n%s%u\n",
          "string1 = ", string1, "string2 = ", string2,
          "The length of the initial segment of string1",
          "containing only characters from string2 = ",
          strspn(string1, string2));
   return 0;
}
```

```
string1 = The value is 3.14159
string2 = aehilsTuv

The length of the initial segment of string1
containing only characters from string2 = 13
```

Fig. 8.27 Cómo utilizar **strspn**.

La función ***strstr*** busca la primera ocurrencia de su segundo argumento de cadena en su primer argumento de cadena. Si su segunda cadena se encuentra en la primera cadena, se regresa un apuntador a la localización de la cadena en el primer argumento. El programa de la figura 8.28 utiliza **strstr** para encontrar la cadena **"def"** en la cadena **"abcdefabcdef"**.

La función ***strtok*** se utiliza para dividir una cadena en una serie de *tokens*. Un token es una serie de caracteres separados por *caracteres delimitantes* (usualmente espacios o marcas de puntuación). Por ejemplo, en una línea de texto, cada palabra puede ser considerada como un token, y los espacios que separan las palabras pueden ser considerados delimitantes.

Se requieren de múltiples llamadas a **strtok** para dividir una cadena en tokens (suponiendo que la cadena contenga más de uno de ellos). La primera llamada a **strtok** contiene dos argumentos, una cadena que va a ser dividida, y una cadena que contiene los caracteres que separan a los tokens. En el programa de la figura 8.29, el enunciado

```
tokenPtr = strtok(string, " ");
```

asigna **tokenPtr** un apuntador al primer token en **string**. El segundo argumento de **strtok**, **" "**, indica que los tokens en **string** están separados por espacios. La función **strtok** busca el primer carácter en **string** que no sea un carácter delimitante (espacio). Esto iniciará el primer token. La función, a continuación, encuentra el siguiente carácter delimitante dentro de la cadena y lo remplaza con un carácter nulo (**'\0'**). Con esto se da por terminado el token actual. La función **strtok** guarda un apuntador al carácter que sigue al token en **string**, y regresa un apuntador al token o ficha actual.


```
/* Using strstr */
#include <stdio.h>
#include <string.h>

main()
{
   char *string1 = "abcdefabcdef";
   char *string2 = "def";

   printf("%s%s\n%s%s\n\n%s\n%s%s%s\n",
          "string1 = ", string1, "string2 = ", string2,
          "The remainder of string1 beginning with the",
          "first occurrence of string2 is: ",
          strstr(string1, string2));

   return 0;
}
```

```
string1 = abcdefabcdef

string2 = def

The remainder of string1 beginning with the
first occurrence of string2 is: defabcdef
```

Fig. 8.28 Cómo utilizar **strstr**.

```
/* Using strtok */
#include <stdio.h>
#include <string.h>

main()
{
   char string[] = "This is a sentence with 7 tokens";
   char *tokenPtr;

   printf("%s\n%s\n\n%s\n",
          "The string to be tokenized is:", string,
          "The tokens are:");

   tokenPtr = strtok(string, " ");

   while (tokenPtr != NULL) {
      printf("%s\n", tokenPtr);
      tokenPtr = strtok(NULL, " ");
   }

   return 0;
}
```

Fig. 8.29 Cómo utilizar **strtok** (parte 1 de 2).

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:
This
is
a
sentence
with
7
tokens
```

Fig. 8.29 Cómo utilizar `strtok` (parte 2 de 2).

Las siguientes llamadas a **strtok**, para continuar la división de **string**, contienen a **NULL** como primer argumento. El argumento **NULL** indica que la llamada a **strtok** deberá continuar la división, a partir de la localización en **string**, guardada por la última llamada a **strtok**. Si después de llamar a **strtok** ya no quedan tokens, **strtok** regresará **NULL**. El programa de la figura 8.29 utiliza **strtok** para dividir la cadena **"This is a sentence with 7 tokens"**. Cada token se imprime por separado. Note que **strtok** modifica la cadena de entrada y, por lo tanto, deberá hacerse una copia de la cadena, si después de las llamadas a **strtok** dicha cadena debe ser utilizada otra vez en el programa.

8.9 Funciones de memoria de la biblioteca de manejo de cadenas

Las funciones de la biblioteca de manejo de cadenas presentadas en esta sección, facilitan manipular, comparar y buscar en bloques de memoria. Las funciones tratan los bloques de memoria como arreglos de caracteres. Estas funciones pueden manipular cualquier bloque de datos. En la figura 8.30 se resumen las funciones de memoria de la biblioteca de manejo de cadenas. En los análisis de función, "objeto" se refiere a un bloque de datos.

Los parámetros de apuntador a estas funciones se declaran **void ***. En el capítulo 7, vimos que un apuntador a cualquier tipo de datos puede ser asignado de forma directa a un apuntador del tipo **void ***, y un apuntador del tipo **void *** puede ser asignado directamente a un apuntador de cualquier tipo de datos. Por esta razón, estas funciones pueden recibir apuntadores de cualquier tipo de datos. Debido a que un apuntador **void *** no puede ser desreferenciado, cada función recibe un argumento de tamaño que defina el número de caracteres (bytes) que la función procesará. Por razones de simplicidad, los ejemplos en esta sección manipulan arreglos de caracteres (bloques de caracteres).

La función ***memcpy*** copia un número especificado de caracteres del objeto al cual señala su segundo argumento, al objeto al cual señala su primer argumento. La función puede recibir un apuntador a cualquier tipo de objeto. Si los dos objetos se sobreponen en memoria, es decir, si ambos son parte del mismo objeto, el resultado de esta función queda indefinido. El programa de la figura 8.31 utiliza **memcpy** para copiar la cadena en el arreglo **s2** al arreglo **s1**.

La función ***memmove***, como la función **memcpy**, copia un número especifico de bytes del objeto señalado por su segundo argumento, al objeto señalado por su primer argumento. La copia

Prototipo de función	Descripción de la función
`void *memcpy(void *s1, const void *s2, size_t n)`	Copia **n** caracteres del objeto señalado por **s2** al objeto señalado por **s1**. Regresa un apuntador al objeto resultante.
`void *memmove(void *s1, const void *s2, size_t n)`	Copia **n** caracteres del objeto señalado por **s2** al objeto señalado por **s1**. La copia se ejecuta como si los caracteres primero fueran copiados del objeto al cual apunta **s2** a un arreglo temporal, y a continuación del arreglo temporal al objeto apuntador por **s1**. Regresa un apuntador al objeto resultante.
`int *memcmp(const void *s1, const void *s2, size_t n)`	Compara los primeros **n** caracteres de los objetos señalados por **s1** y **s2**. La función regresa **0**, menos que **0**, o más que **0**, si **s1** es igual a, menor que o mayor que **s2**.
`void *memchr(const void *s, int c, size_t n)`	Localiza la primera instancia de **c** (convertida a **unsigned char**) en los primeros **n** caracteres del objeto señalado por **s**. Si **c** es encontrado, regresa un apuntador a **c** en el objeto. De lo contrario, regresa **NULL**.
`void *memset(void *s, int c, size_t n)`	Copia **c** (convertido a **unsigned char**) en los primeros **n** caracteres del objeto señalado por **s**. Regresa un apuntador hacia el resultado.

Fig. 8.30 Funciones de memoria de la biblioteca de manejo de cadenas.

```
/* Using memcpy */
#include <stdio.h>
#include <string.h>

main()
{
   char s1[17], s2[]  = "Copy this string";

   memcpy(s1, s2, 17);
   printf("%s\n%s\"%s\"\n",
          "After s2 is copied into s1 with memcpy,",
          "s1 contains ", s1);
   return 0;
}
```

```
After s2 is copied into s1 with memcpy,
s1 contains "Copy this string"
```

Fig. 8.31 Cómo utilizar `memcpy`.

se lleva a cabo como si primero los bytes fueran copiados del segundo argumento a un arreglo temporal de caracteres, y a continuación copiados de dicho arreglo temporal al primer argumento. Esto permite que los caracteres de una parte de una cadena sean copiados a otra parte de la misma cadena.

Error común de programación 8.8

Funciones de manipulación de cadenas distintas a **memmove** *que copian caracteres dan resultados indefinidos, cuando la copia se efectúa entre partes de una misma cadena.*

El programa de la figura 8.32 utiliza **memmove** para copiar los últimos **10** bytes del arreglo **x**, en los primeros **10** bytes del arreglo **x**.

La función ***memcmp*** (figura 8.33) compara el número especificado de caracteres de su primer argumento, con los caracteres correspondientes de su segundo argumento. La función regresa un valor mayor que 0, si el primer argumento es mayor que el segundo argumento, regresa 0 si los argumentos son iguales, y regresa un valor menor que 0 si el primer argumento es menor que el segundo argumento.

La función ***memchr*** busca la primera incidencia de un byte, representado como un **unsigned char**, en el número especificado de bytes de un objeto. Si encuentra el byte, regresa un apuntador al byte en el objeto, de lo contrario regresa un apuntador **NULL**. El programa de la figura 8.34 busca el carácter (byte) **'r'** en la cadena **"This is a string"**.

La función **memset** copia el valor del byte en su segundo argumento en un número específico de bytes del objeto al cual señala su primer argumento. El programa en la figura 8.35 utiliza **memset** para copiar **'b'** en los primeros 7 bytes de **string1**.

```
/* Using memmove */
#include <stdio.h>
#include <string.h>

main()
{
   char x[] = "Home Sweet Home";

   printf("%s%s\n",
          "The string in array x before memmove is: ", x);
   printf("%s%s\n",
          "The string in array x after memmove is:  ",
          memmove(x, &x[5], 10));

   return 0;
}
```

```
The string in array x before memmove is: Home Sweet Home
The string in array x after memmove is:  Sweet Home Home
```

Fig. 8.32 Cómo utilizar **memmove**.


```
/* Using memcmp */
#include <stdio.h>
#include <string.h>

main()
{
   char s1[] = "ABCDEFG", s2[] = "ABCDXYZ";

   printf("%s%s\n%s%s\n\n%s%2d\n%s%2d\n%s%2d\n",
          "s1 = ", s1, "s2 = ", s2,
          "memcmp(s1, s2, 4) = ", memcmp(s1, s2, 4),
          "memcmp(s1, s2, 7) = ", memcmp(s1, s2, 7),
          "memcmp(s2, s1, 7) = ", memcmp(s2, s1, 7));
   return 0;
}
```

```
s1 = ABCDEFG
s2 = ABCDXYZ

memcmp(s1, s2, 4) =  0
memcmp(s1, s2, 7) = -1
memcmp(s2, s1, 7) =  1
```

Fig. 8.33 Cómo utilizar **memcmp**.

```
/* Using memchr */
#include <stdio.h>
#include <string.h>

main()
{
   char *s = "This is a string";

   printf("%s\'%c\'%s\"%s\"\n",
          "The remainder of s after character ", 'r',
          " is found is ", memchr(s, 'r', 16));
   return 0;
}
```

```
The remainder of s after character 'r' is found is "ring"
```

Fig. 8.34 Cómo utilizar **memchr**.

8.10 Otras funciones de la biblioteca de manejo de cadenas

Las dos funciones restantes de la biblioteca de manejo de cadenas son **strerror** y **strlen**. La figura 8.36 resume las funciones **strerror** y **strlen**.

```
/* Using memset */
#include <stdio.h>
#include <string.h>

main()
{
   char string1[15] = "BBBBBBBBBBBBBB";

   printf("string1 = %s\n", string1);
   printf("string1 after memset = %s\n",
          memset(string1, 'b', 7));
   return 0;
}
```

```
string1 = BBBBBBBBBBBBBB
string1 after memset = bbbbbbbBBBBBBB
```

Fig. 8.35 Cómo utilizar **memset**.

La función ***strerror*** toma un número de error y crea un cadena de mensaje de error. Regresa un apuntador a la cadena. El programa de la figura 8.37 demuestra **strerror**.

Sugerencia de portabilidad 8.4

El mensaje generado por **strerror** *es dependiente del sistema.*

La función ***strlen*** toma una cadena como argumento, y regresa el número de caracteres en la cadena —el carácter de terminación no queda incluido en la longitud. El programa de la figura 8.38 demuestra la función **strlen**.

Resumen

- La función **islower** determina si su argumento es una letra minúscula (**a - z**).
- La función **isupper** determina si su argumento es una letra mayúscula (**A - Z**).

Prototipo de función	Descripción de la función
char *strerror(int errornum)	Proyecta **errornum** en una cadena completa de texto, de forma dependiente del sistema. Regresa un apuntador a la cadena.
size_t strlen(const char *s)	Determina la longitud de la cadena **s**. Regresa el número de caracteres que anteceden al carácter de terminación **NULL**.

Fig. 8.36 Funciones de manipulación de cadenas de la biblioteca de manejo de cadenas.


```
/* Using strerror */
#include <stdio.h>
#include <string.h>

main()
{
   printf("%s\n", strerror(2));
   return 0;
}
```

```
Error 2
```

Fig. 8.37 Cómo utilizar **strerror**.

```
/* Using strlen */
#include <stdio.h>
#include <string.h>

main()
{
   char *string1 = "abcdefghijklmnopqrstuvwxyz";
   char *string2 = "four";
   char *string3 = "Boston";

   printf("%s\"%s\"%s%lu\n%s\"%s\"%s%lu\n%s\"%s\"%s%lu\n",
          "The length of ", string1, " is ", strlen(string1),
          "The length of ", string2, " is ", strlen(string2),
          "The length of ", string3, " is ", strlen(string3));
   return 0;
}
```

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

Fig. 8.38 Cómo utilizar **strlen**.

- La función **isdigit** determina si su argumento es un dígito (**0 - 9**).
- La función **isalpha** determina si su argumento es una letra mayúscula (**A - Z**), o una minúscula (**a - z**).
- La función **isalnum** determina si su argumento es una letra mayúscula (**A - Z**), una minúscula (**a - z**), o un dígito (**0 - 9**).
- La función **isxdigit** determina si su argumento es un dígito hexadecimal (**A - F**, **a - f**, **0 - 9**).

- La función **toupper** convierte una letra minúscula en mayúscula, y regresa la letra mayúscula.
- La función **tolower** convierte una letra mayúscula en minúscula, y regresa la letra minúscula.
- La función **isspace** determina si su argumento es alguno de los caracteres de espacio en blanco siguientes: `' '` (espacio), `'\f'`, `'\n'`, `'\r'`, `'\t'` o bien `'\v'`.
- La función **iscntrl** determina si su argumento es uno de los caracteres de control siguientes: `'\t'`, `'\v'`, `'\f'`, `'\a'`, `'\b'`, `'\r'`, o `'\n'`.
- La función **ispunct** determina si su argumento es un carácter de impresión distinto de un espacio, un dígito o una letra.
- La función **isprint** determina si su argumento es cualquier carácter de impresión, incluye el carácter de espacio.
- La función **isgraph** determina si su argumento es un carácter de impresión distinto al carácter de espacio.
- La función **atof** convierte su argumento —una cadena que empieza con una serie de dígitos que representan un número de punto flotante— a un valor **double**.
- La función **atoi** convierte su argumento —una cadena empezando con una serie de dígitos que representan un entero— a un valor **int**.
- La función **atol** convierte su argumento —una cadena empezando con una serie de dígitos que representan un entero **long**— a un valor **long**.
- La función **strtod** convierte una secuencia de caracteres que representan un valor de punto flotante a **double**. La función recibe dos argumentos —una cadena (**char ***) y un apuntador a **char ***. La cadena contiene la secuencia de caracteres a convertirse, y el apuntador a **char *** se asigna al resto de la cadena, después de la conversión.
- La función **strtol** convierte una secuencia de caracteres representando un entero **long**. La función recibe tres argumentos—una cadena (**char ***), un apuntador a **char ***, y un entero. La cadena contiene la secuencia de caracteres a convertirse, el apuntador a **char *** es asignado el resto de la cadena después de la conversión, y el entero define la base del valor bajo conversión.
- La función **strtoul** convierte una secuencia de caracteres representando un entero a **unsigned long**. La función recibe tres argumentos —una cadena (**char ***), un apuntador a **char ***, y un entero. La cadena contiene la secuencia de caracteres a convertirse, el apuntador a **char *** es asignado el resto de la cadena después de la conversión, y el entero define la base del valor que está siendo convertido.
- La función **gets** lee caracteres de la entrada estándar (teclado) hasta que encuentra un carácter de nueva línea o un indicador de fin de archivo. El argumento a **gets** es un arreglo del tipo **char**. Después de que se termina la lectura, se agrega un carácter **NULL** (`'\0'`) al arreglo.
- La función **putchar** imprime su argumento de carácter.
- La función **getchar** lee un carácter de la entrada estándar y regresa el carácter como un entero. Si encuentra el indicador de fin de archivo, **getchar** regresa **EOF**.
- La función **puts** toma una cadena (**char ***) como argumento e imprime la cadena, seguida por un carácter de nueva línea.

- La función **sprintf** utiliza las mismas especificaciones de conversión que **printf** para imprimir datos con formato, en un arreglo del tipo **char**.
- La función **sscanf** utiliza las mismas especificaciones de conversión que la función **scanf** para leer datos con formato de una cadena.
- La función **strcpy** copia su segundo argumento —una cadena —en su primer argumento— un carácter. El programador deberá asegurarse que el arreglo es lo suficiente grande para almacenar la cadena junto con su carácter de terminación **NULL**.
- La función **strncpy** es equivalente a **strcpy**, excepto que una llamada a **strncpy** especifica el número de caracteres a copiarse de la cadena al arreglo. El carácter de terminación **NULL** sólo será copiado si el número de caracteres a copiarse es uno más que la longitud de la cadena.
- La función **strcat** agrega su segundo argumento de cadena —incluyendo el carácter de terminación **NULL**— a su primer argumento de cadena. El primer carácter de la segunda cadena remplaza el carácter **NULL** (`'\0'`) de la primera cadena. El programador deberá asegurarse que el arreglo utilizado para almacenar la primera cadena es lo suficiente extenso para almacenar tanto la primera como la segunda cadena.
- La función **strncat** agrega un número específico de caracteres provenientes de la segunda a la primera cadena. Al resultado se agrega un carácter de terminación **NULL**.
- La función **strcmp** compara su primer argumento de cadena con su segundo argumento de cadena, carácter por carácter. La función regresa 0 si las cadenas son iguales, regresa un valor negativo si la primera cadena es menor que la segunda cadena, y regresa un valor positivo si la primera cadena es mayor que la segunda cadena.
- La función **strncmp** es equivalente a **strcmp**, excepto que **strncmp** compara un número específico de caracteres. Si el número de caracteres en una de las cadenas es menor que el número de caracteres especificado, **strncmp** comparará caracteres, hasta que se encuentre con el carácter **NULL** en la cadena más corta.
- La función **strchr** busca la primera ocurrencia de un carácter en una cadena. Si encuentra el carácter, **strchr** regresa un apuntador del carácter a la cadena, de lo contrario, **strchr** regresa **NULL**.
- La función **strcspn** determina la longitud de la parte inicial de la cadena de su primer argumento, que no contenga ningún carácter de la cadena incluida en su segundo argumento. La función regresa la longitud del segmento.
- La función **strpbrk** busca en su primer argumento la primera ocurrencia de cualquier carácter existente en su segundo argumento. Si encuentra un carácter del segundo argumento, **strpbrk** regresa un apuntador al carácter, de lo contrario **strpbrk** regresa **NULL**.
- La función **strrchr** busca la última ocurrencia de un carácter en una cadena. Si dicho carácter es encontrado, **strrchr** regresa un apuntador al carácter en la cadena, de lo contrario **strrchr** regresa **NULL**.
- La función **strspn** determina la longitud de la parte inicial de la cadena de su primer argumento, que contenga sólo caracteres de la cadena de su segundo argumento. La función regresa la longitud del segmento.
- La función **strstr** busca la primera ocurrencia de su segundo argumento de cadena, en su primer argumento de cadena. Si la segunda cadena se encuentra en la primera cadena, regresa un apuntador a la posición de la cadena en el primer argumento.

- Una secuencia de llamadas a **strtok** divide la cadena **s1** en tokens, que están separados por caracteres contenidos en la cadena **s2**. La primera llamada contiene a **s1** como primer argumento, y las llamadas subsecuentes para continuar la división de la misma cadena, contienen **NULL** como primer argumento. Para cada llamada se devuelve un apuntador al token actual. Si ya no existen más tokens cuando la función es llamada, regresa un apuntador **NULL**.
- La función **memcpy** copia un número especificado de caracteres del objeto a donde señala su segundo argumento, al objeto a donde señala su primer argumento. La función puede recibir un apuntador a cualquier tipo de objeto. Los apuntadores son recibidos por **memcpy** como apuntadores **void**, y para uso en la función convertidos a apuntadores **char**. La función **memcpy** manipula los bytes del objeto como si fueran caracteres.
- La función **memmove** copia un número especifico de bytes del objeto señalado por su segundo argumento, al objeto señalado por su primer argumento. La copia se lleva a cabo como si los bytes fuesen copiados del segundo argumento a un arreglo de caracteres temporal, y a continuación copiados del arreglo temporal al primer argumento.
- La función **memcmp** compara el número especificado de caracteres en su primero y segundo argumentos.
- La función **memchr** busca la primera señal de un byte, representado como un **unsigned char**, en el número especificado de bytes de un objeto. Si encuentra el byte, regresa un apuntador hacia el byte, de lo contrario, regresa un apuntador **NULL**.
- La función **memset** copia su segundo argumento, tratado como un **unsigned char**, a un número especifico de bytes del objeto al cual señala su primer argumento.
- La función **strerror** define un número entero de error en una cadena completa de texto, de forma dependiente del sistema. Regresa un apuntador hacia la cadena.
- La función **strlen** toma como argumento una cadena, y regresa el número de caracteres en la cadena —en la longitud de la cadena no se incluye el carácter de terminación **NULL**.

Terminología

agregar cadenas a otras cadenas
ASCII
atof
atoi
atol
código de carácter
constante de carácter
conjunto de caracteres
comparar cadenas
carácter de control
copiar cadenas
ctype.h
delimitador
ABCDI
biblioteca general de utilería
getchar
gets
dígitos hexadecimales
isalnum
isalpha
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper

isxdigit
longitud de una cadena
literal
memchr
memcmp
memcpy
memmove
memset
representación del código numérico de un carácter
impresión de carácter
putchar
puts
cadena de búsqueda
sprintf
sscanf
stdio.h
stdlib.h
strcat
strchr
strcmp
strcpy
strcspn
strerror
cadena
funciones de comparación de cadena
concatenación de cadenas
constante de cadena
funciones de conversión de cadenas
string.h
literal de cadena
procesamiento de cadenas
strlen
strncat
strncmp
strncpy
strpbrk
strrchr
strspn
strstr
strtod
strtok
strtol
strtoul
tolower
token
división de cadenas
toupper
caracteres de espacio en blanco
procesamiento de palabras

Errores comunes de programación

8.1 No asignar suficiente espacio en un arreglo de caracteres para almacenar el carácter **NULL** que da por terminado una cadena.

8.2 Imprimir una "cadena" que no contenga un carácter de terminación **NULL**.

8.3 Procesar un carácter como si fuera una cadena. Una cadena es un apuntador —probablemente un entero grande. Sin embargo, un carácter es un entero pequeño (valores ASCII del rango 0-225). En muchos sistemas, esto causará un error, porque las direcciones bajas de memoria se reservan para fines especiales, como manejadores de interruptores del sistema operativo y, por lo tanto, se incurrirá en "violaciones de acceso".

8.4 Pasar un carácter como argumento a una función, cuando se espera una cadena.

8.5 Pasar una cadena como argumento a una función, cuando se espera un carácter.

8.6 No agregar un carácter de terminación **NULL** al primer argumento de un **strncpy**, cuando el tercer argumento es menor que o igual a la longitud de la cadena del segundo argumento.

8.7 Suponiendo que **strcmp** y **strncmp** regresan 1 cuando sus argumentos son iguales. Ambas funciones regresan 0 (valor falso en C) en caso de igualdad. Por lo tanto, cuando se prueben dos cadenas buscando igualdad, para determinar si las cadenas son iguales, el resultado de la función **strcmp** o la **strncmp** deberá ser comparado con 0.

8.8 Funciones de manipulación de cadenas distintas a **memmove** que copian caracteres dan resultados indefinidos, cuando la copia se efectúa entre partes de una misma cadena.

Prácticas sanas de programación

8.1 Al almacenar una cadena de caracteres en un arreglo de caracteres, asegúrese que el arreglo sea lo suficiente extenso para contener la cadena más grande que pueda ser almacenada. C permite que se almacenen cadenas de cualquier longitud. Si una cadena resulta más larga que el arreglo de caracteres en la cual debe almacenarse, los caracteres que excedan del final del arreglo sobreescribirán datos en memoria a continuación del mismo.

8.2 Al usar funciones de la biblioteca de manejo de caracteres, incluya el archivo de cabecera `<ctype.h>`

8.3 Al utilizar funciones de la biblioteca general de utilerías, incluya el archivo de cabecera `<stdlib.h>`

8.4 Al utilizar funciones de la biblioteca estándar de entrada/salidas, incluya el archivo de cabecera `<stdio.h>`.

8.5 Al utilizar funciones de la biblioteca de manejo de cadenas, incluya el archivo de cabecera `<string.h>`.

Sugerencias de portabilidad

8.1 Cuando una variable del tipo `char *` es inicializada con una literal de cadena, algunos compiladores pudieran colocar la cadena en una posición en memoria donde ésta no pueda ser modificada. Si pudiera necesitar modificar una literal de cadena, deberá ser almacenada en un arreglo de caracteres, para asegurarse la modificabilidad en todos los sistemas.

8.2 Los códigos numéricos internos utilizados para representar caracteres pudieran ser distintos en diferentes computadoras.

8.3 El tipo `size_t` es un sinónimo, dependiente del sistema, para el tipo `unsigned long` o para el tipo `unsigned int`.

8.4 El mensaje generado por `strerror` es dependiente del sistema.

Ejercicios de autoevaluación

8.1 Escriba un enunciado para ejecutar cada uno de los siguientes. Suponga que las variables `c` (que almacenan un carácter), `x`, `y` y `z` son del tipo `int`, y las variables `d`, `e`, y `f` son del tipo `float`, que la variable `ptr` es del tipo `char *`, y los arreglos `s1[100]` y `s2[100]` son del tipo `char`.

a) Convierta en una letra mayúscula, el carácter almacenado en la variable `c`. Asigne el resultado a la variable `c`.
b) Determine si el valor de la variable `c` es un dígito. Cuando se despliegue el resultado útilice el operador condicional, para imprimir `"is a "` o `"is not a "`, tal y como se mostró en las figuras 8.2, 8.3 y 8.4.
c) Convierta la cadena `"1234567"` a `long` e imprima el valor.
d) Determine si el valor de la variable `c` es un carácter de control. Cuando se despliegue el resultado utilice el operador condicional para imprimir `"is a "`, o bien `"is not a "`.
e) Lea desde el teclado una línea de texto al arreglo `s1`. No utilice `scanf`.
f) Imprima la línea de texto almacenado en el arreglo `s1`. No utilice `printf`.
g) Asigne `ptr` la posición de la última instancia de `c` en `s1`.
h) Imprima el valor de la variable `c`. No utilice `printf`.
i) Convierta la cadena `"8.63582"` a `double` e imprima el valor.
j) Determine si el valor de `c` es una letra. Cuando se muestre el resultado utilice el operador condicional para imprimir `"is a "`, o bien `"is not a "`.
k) Lea un carácter del teclado, y almacene dicho carácter en la variable `c`.
l) Asigne `ptr` la posición de la primera instancia de `s2` en `s1`.
m) Determine si el valor de la variable `c` es un carácter de impresión. Cuando se muestre el resultado utilice el operador condicional para imprimir `"is a "`, o bien `"is not a "`.
n) Lea tres valores `float` en las variables `d`, `e` o `f` a partir de la cadena `"1.27 10.3 9.432"`.

o) Copie la cadena almacenada en el arreglo `s2` al arreglo `s1`.
p) Asigne `ptr` a la posición en la primera ocurrencia de `s1`, de cualquier carácter de `s2`.
q) Compare la cadena en `s1` con la cadena en `s2`. Imprima el resultado.
r) Asigne `ptr` a la posición en la primera ocurrencia de `c` en `s1`.
s) Utilice `sprintf` para imprimir los valores de las variables enteras `x`, `y` y `z` en el arreglo `s1`. Cada valor deberá ser impreso con un ancho de campo de 7.
t) Agregue 10 caracteres de la cadena en `s2` a la cadena en `s1`.
u) Determine la longitud de la cadena en `s1`. Imprima el resultado.
v) Convierta la cadena `"-21"` a `int` e imprima el valor.
w) Asigne `ptr` a la posición del primer token en `s1`. Los tokens en `s2` están separados por comas (,).

8.2 Muestre dos métodos diferentes para inicializar el arreglo de caracteres `vowel` con la cadena de vocales, `"AEIOU"`

8.3 ¿Qué es lo que, si es que existe, se imprime cuando se ejecutan cada uno de los siguientes enunciados de C? Si el enunciado contiene un error, descríbalo e indique cómo corregirlo. Suponga las siguientes declaraciones de variables:

```
char s1[50] = "jack", s2[50] = "jill", s3[50], *sptr;
```

a) `printf("%c%s", toupper(s1[0]), &s1[1]);`
b) `printf("%s", strcpy(s3, s2));`
c) `printf("%s", strcat(strcat(strcpy(s3, s1),"and"), s2));`
d) `printf("%u", strlen(s1) + strlen(s2));`
e) `printf("%u", strlen(s3));`

8.4 Encuentre el error en cada uno de los segmentos de programas siguientes, y explique cómo corregirlo:

a)
```
char s[10];
strncpy(s, "hello", 5);
printf("%s\n", s);
```
b) `printf("%s", 'a');`
c)
```
char s[12];
strcpy(s, "Welcome Home");
```
d)
```
if (strcmp(string1, string2))
printf("The strings are equal\n");
```

Respuestas a los ejercicios de autoevaluación

8.1
a) `c = toupper(c);`
b)
```
printf("'%c'%sdigit\n",
   c, isdigit(c) ? " is a " : " is not a ");
```
c) `printf("%ld\n", atol("1234567"));`
d)
```
printf("'%c'%scontrol character\n",
   c, iscntrl(c) ? " is a " : " is not a ");
```
e) `gets(s1);`
f) `puts(s1);`
g) `ptr = strrchr(s1, c);`
h) `putchar(c);`
i) `printf("%f\n", atof("8.63582"));`
j)
```
printf("'%c'%sletter\n",
   c, isalpha(c) ? " is a " : " is not a ");
```
k) `c = getchar();`

```
l) ptr = strstr(s1, s2);
m) printf("'%c'%sprinting character\n;
   c, isprint(c) ? " is a " : " is not a ");
n) sscanf("1.27 10.3 0.432", "%f%f%f", &d, &e, &f);
o) strcpy(s1, s2);
p) ptr = strpbrk(s1, s2);
q) printf("strcmp(s1, s2) = %d\n", strcmp(s1, s2));
r) ptr = strchr(s1, c);
s) sprintf(s1, "%7d%7d%7d", x, y, z);
t) strncat(s1, s2, 10);
u) printf("strlen(s1) = %u\n, strlen(s1));
v) printf("%d\n", atoi("-21"));
w) ptr = strtok(s2, ",");
```

8.2
```
char vowel[] = "AEIOU";
CHAR vowel[] = {'A', 'E', 'I', 'O', 'U', '\0'};
```

8.3
a) Jack
b) jill
c) jack and jill
d) 8
e) 13

8.4
a) Error: la función **strncpy** no escribe un carácter de terminación **NULL** al arreglo **s**, porque su tercer argumento es igual a la longitud de la cadena **"hello"**.
   Corrección: haga 6 el tercer argumento de **strncpy**, o asigne **'\0'** a **s[5]**.
b) Error: intentar imprimir una constante de caracteres como una cadena.
   Corrección: utilice **%c** para extraer el carácter, o bien remplace **'a'** con **"a"**.
c) Error: el arreglo de caracteres **s** no tiene suficiente tamaño para almacenar el carácter de terminación **NULL**.
   Corrección: Declare el arreglo con más elementos.
d) Error: la función **strcmp** regresará 0 si las cadenas son iguales, por lo tanto, será falsa la condición en la estructura **if**, y **printf** no será ejecutado.
   Corrección: compare en la condición el resultado de **strcmp** con **0**.

Ejercicios

8.5 Escriba un programa que introduzca un carácter del teclado, y pruebe el carácter con cada una de las funciones en la biblioteca de manejo de caracteres. El programa deberá imprimir el valor regresado por cada una de las funciones.

8.6 Escriba un programa que introduzca una línea de texto con la función **gets**, en el arreglo de caracteres **s[100]**. Extraiga la línea en letras mayúsculas y minúsculas.

8.7 Escriba un programa que introduzca 4 cadenas que representen enteros, convierta las cadenas a enteros, sume los valores e imprima el total de los 4 valores.

8.8 Escriba un programa que introduzca 4 cadenas, que representen valores en punto flotante, convierta las cadenas a valores **double**, sume los valores e imprima el total de los 4 valores.

8.9 Escriba un programa que utilice la función **strcmp**, para comparar dos cadenas introducidas por el usuario. El programa deberá indicar si la primera cadena es menor que, igual que o mayor que la segunda cadena.

8.10 Escriba un programa que utilice la función **strncmp**, para comparar dos cadenas introducidas por el usuario. El programa deberá introducir el número de caracteres a compararse. El programa deberá indicar si la primera cadena es menor que, igual que o mayor que la segunda cadena.

8.11 Escriba un programa que utilice generación de números aleatorios para crear oraciones. El programa deberá utilizar cuatro arreglos de apuntadores a **char** llamados **article, noun, verb,** y **preposition.** El programa deberá crear una oración, seleccionando una palabra al azar de cada uno de los arreglos, en el orden siguiente: **article, noun, verb, preposition, article** y **noun.** Conforme se seleccione cada palabra, deberá ser concatenada con las palabras anteriores en un arreglo lo suficiente extensa para contener a toda la oración. Las palabras deberán estar separadas por espacios. Cuando se extraiga la oración final, deberá iniciar con una letra mayúscula y terminar con un punto. El programa deberá generar 20 oraciones de este tipo.

Los arreglos deberán ser llenados como sigue: el arreglo **article** deberá contener los artículos **"the", "a", "one" "some", y "any"**; el arreglo **noun** deberá contener los nombres **"boy", "girl", "dog", "town",** y **"car"**; el arreglo **verb** deberá contener los verbos **"drove", "jumped", "ran", "walked",** y **"skipped"**; el arreglo **preposition** deberá contener las preposiciones: **"to", "from", "over", "under",** y **"on"**.

Una vez escrito el programa anterior y funcionando, modifíquelo para producir una corta historia que esté formada con varias de estas oraciones. (¡Qué tal le parecería la posibilidad de convertirse en un escritor de tesis al azar!)

8.12 (*Quintillas jocosas*). Una quintilla jocosa es un verso cómico de cinco líneas, en las cuales la primera y segunda línea riman con la quinta, y la tercera rima con la cuarta. Utilizando técnicas similares a las desarrolladas en el ejercicio 8.10, escriba un programa en C que produzca quintillas al azar. Pulir este programa para que pueda producir buenas quintillas es un verdadero reto, ¡pero el resultado puede merecer el esfuerzo!.

8.13 Escriba un programa que cifre frases de la lengua inglesa y las convierta en latín infantil. El latín infantil es una forma de lenguaje codificado utilizado a menudo para diversión. Existen muchas variantes en los métodos utilizados para formar frases en latín infantil. Para simplificar, utilice el algoritmo siguiente:

Para formar una frase en latín infantil a partir de una frase en lengua inglesa, divida la frase en palabras, utilizando la función **strtok**. Para traducir cada palabra inglesa en una palabra en latín infantil, coloque la primera letra de la palabra inglesa al final de la palabra y añada las letras **"ay"**. De ahí la palabra **"jump"** se convierte en **"umpjay"**, la palabra **"the"** se convierte en **"hetay"**, y la palabra **computer** se convierte en **"omputercay"**. Los espacios en blanco entre palabras se conservan como están. Suponga lo siguiente: la frase en inglés está formada de palabras separadas por espacios en blanco, no hay signos de puntuación y todas las palabras tienen dos o más letras. La función **printLatinWord** deberá desplegar cada palabra. *Sugerencia*: cada vez que se encuentre un token en una llamada a **strtok**, pase el apuntador del token a la función **printLatinWord**, e imprima la palabra en latín infantil.

8.14 Escriba un programa que introduzca un número telefónico como cadena, en la forma **(555) 555-5555**. El programa deberá utilizar la función **strtok** para extraer el código de área como token, los tres primeros dígitos del número telefónico como token y los últimos cuatro dígitos del número telefónico como token. Los siete dígitos del número telefónico deberán ser concatenados en una cadena. El programa deberá convertir la cadena del código de área a **int**, y convertir la cadena del número telefónico a **long**. Tanto el código de área como el número telefónico deberán ser impresos.

8.15 Escriba un programa que introduzca una línea de texto, divida la línea con la función **strtok** y extraiga los tokens en orden inverso.

8.16 Escriba un programa que introduzca desde el teclado una línea de texto y una cadena de búsqueda. Utilizando la función **strstr** localice en la línea de texto la primera ocurrencia de la cadena de búsqueda, y asigne la posición a la variable **searchPtr** del tipo **char *.** Si encuentra la cadena de búsqueda, imprima el resto de la línea de texto, empezando con la cadena de texto. A continuación utilice otra vez **strstr**, para localizar en la línea de texto la siguiente ocurrencia de la cadena de búsqueda. Si encuentra una segunda ocurrencia, imprima el resto de la línea de texto, empezando con la segunda ocurrencia. *Sugerencia*: la segunda llamada a **strstr** deberá contener **searchPtr + 1** como primer argumento.

8.17 Escriba un programa basado en el programa del ejercicio 8.16, que introduzca varias líneas de texto y una cadena de búsqueda, y utilice la función `strstr` para determinar todas las ocurrencias de la cadena, en las líneas de texto. Imprima el resultado.

8.18 Escriba un programa que introduzca varias líneas de texto y un carácter de búsqueda, y utilice la función `strchr` para determinar todas las ocurrencias del carácter, en las líneas de texto

8.19 Escriba un programa basado en el programa del ejercicio 8.18 que introduzca varias líneas de texto y utilice la función `strchr` para determinar todas las ocurrencias de cada letra del alfabeto en las líneas de texto. Las letras mayúsculas y minúsculas deberán ser contadas juntas. Almacene en un arreglo los totales de cada letra, y una vez que se hayan determinado los totales, imprímalos en formato tabular.

8.20 Escriba un programa que introduzca varias líneas de texto y utilice `strtok`, para contar el número total de palabras. Suponga que las palabras están separadas, ya sea por espacios o por caracteres de nueva línea.

8.21 Utilice las funciones de comparación de cadenas, analizadas en la Sección 8.6, y las técnicas para ordenamiento de arreglos, desarrolladas en el capítulo 6, para escribir un programa que alfabetice una lista de cadenas. Utilice como datos para su programa los nombres de 10 ó 15 ciudades en su área.

8.22 La gráfica en el apéndice D muestra las representaciones de código numérico, para los caracteres del conjunto de caracteres ASCII. Estudie esta gráfica y a continuación, indique si cada uno de los siguientes es verdadero o falso.

a) La letra `"A"` viene antes de la letra `"B"`.
b) El dígito `"9"` viene antes del dígito `"0"`.
c) Los símbolos comúnmente utilizados para la suma, resta, multiplicación y división, todos ellos vienen antes de cualquiera de los dígitos.
d) Los dígitos vienen antes de las letras.
e) Si un programa de ordenación ordena cadenas en una secuencia ascendente, entonces el programa colocará el símbolo para un paréntesis derecho, antes del símbolo para uno izquierdo.

8.23 Escriba un programa que lea una serie de cadenas, e imprima sólo estas cadenas, empezando con la letra `"b"`.

8.24 Escriba un programa que lea una serie de cadenas, e imprima sólo aquellas cadenas que terminan con las letras `"ED"`.

8.25 Escriba un programa que introduzca un código ASCII e imprima el carácter correspondiente. Modifique este programa de tal forma que genere todos los códigos posibles de tres dígitos en el rango de 000 a 255 e intente imprimir los caracteres correspondientes. ¿Qué es lo que ocurre cuando se ejecuta este programa?

8.26 Utilizando la gráfica de caracteres ASCII del apéndice D como guía, escriba sus propias versiones de las funciones de manejo de caracteres de la figura 8.1.

8.27 Escriba sus propias versiones de las funciones en la figura 8.5 para conversión de cadenas a números.

8.28 Escriba dos versiones de cada una de las funciones de copia y concatenación de cadenas de la figura 8.17. La primera versión deberá utilizar subíndices de arreglos, y la segunda versión deberá utilizar apuntadores y aritmética de apuntadores.

8.29 Escriba sus propias versiones de las funciones `getchar`, `gets`, `putchar`, y `puts`, descritas en la figura 8.12.

8.30 Escriba dos versiones de cada una de las funciones de comparación de cadenas de la figura. 8.20. La primera versión deberá usar subíndices de arreglos, y la segunda apuntadores y aritmética de apuntadores.

8.31 Escriba sus propias versiones de las funciones en la figura 8.22, para la búsqueda de cadenas.

8.32 Escriba su propias versiones de las funciones en la figura 8.30, para manipular bloques de memoria.

8.33 Escriba dos versiones de la función `strlen` de la figura 8.36. La primera versión deberá utilizar subíndices de arreglos, y la segunda deberá utilizar apuntadores y aritmética de apuntadores.

Sección especial: ejercicios avanzados de manipulación de cadenas

Los ejercicios anteriores están relacionados con el texto y diseñados para probar la comprensión del lector sobre conceptos fundamentales de manipulación de cadenas. En esta sección se incluye un conjunto de problemas intermedios y avanzados. El lector encontrará estos problemas excitantes y divertidos. Los problemas varían de forma considerable al grado de dificultad. Algunos requieren de una hora o dos de escritura y puesta en marcha de los programas. Otros son útiles para tareas de laboratorio, que pudieran requerir dos a tres semanas para su estudio y puesta en marcha. Algunos son proyectos de tesis que plantean retos interesantes.

8.34 (*Análisis de texto*). La disponibilidad de computadoras con capacidades de manipulación de cadenas ha resultado en varios métodos interesantes para analizar lo escrito por grandes autores. Se ha puesto gran atención al hecho de saber si William Shakespeare alguna vez existió. Algunos estudiosos creen que existe evidencia sustancial indicando que Christopher Marlowe, de hecho fue el que escribió las obras maestras atribuidas a Shakespeare. Los investigadores han utilizado computadoras para localizar similitudes en los textos de estos dos autores. Este ejercicio examina tres métodos para analizar texto, utilizando una computadora.

a) Escriba un programa que lea varias líneas de texto e imprima una tabla indicando el número de instancias de cada letra del alfabeto en dicho texto. Por ejemplo, la frase

```
To be, or not to be: that is the question:
```

contiene una "a", dos "b", ninguna "c", etcétera.

b) Escriba un programa que lea varias líneas de texto e imprima una tabla que indique el número de palabras de una letra, de dos letras, de tres letras que aparecen en el texto. Por ejemplo, la frase

```
Whether 'tis nobler in the mind to suffer
```

contiene

Longitud de palabra	Ocurrencias
1	0
2	2
3	2
4	2 (including 'tis)
5	0
6	2
7	1

c) Escriba un programa que lea varias líneas de texto e imprima una tabla indicando el número de ocurrencias de cada palabra distinta en el texto. La primera versión de su programa deberá incluir en la tabla las palabras en el mismo orden en que aparecen dentro de texto. Una impresión más interesante (y más útil) deberá intentarse, en la cual las palabras se ordenarán en forma alfabética. Por ejemplo, las líneas

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
```

contiene las palabras "to" tres veces, la palabra "be" dos veces, la palabra "or" una vez, etcétera.

8.35 (*Procesamiento de palabras*). El tratamiento detallado de la manipulación de cadenas en este texto debe atribuirse principalmente al excitante crecimiento, en años recientes, del procesamiento de texto. Una función importante de los sistemas de procesamiento de palabras es el *tipo justificado* —la alineación de las palabras, tanto en los márgenes izquierdo como derecho de una página. Esto genera un documento de apariencia profesional, que parece haber sido formado en tipografía, en vez de preparado en una máquina de escribir. El tipo justificado puede ser llevado a cabo en sistemas de computación, insertando uno o más caracteres en blanco entre cada una de las palabras de una línea, de tal forma que la palabra más a la derecha se alinee con el margen derecho.

Escriba un programa que lea varias líneas de texto e imprima este texto en formato de tipo justificado. Suponga que el texto debe ser impreso en papel de un ancho de 8 1/2 pulgadas, y que se deben dejar márgenes de una pulgada, tanto en el lado derecho como izquierdo de la página impresa. Suponga que la computadora imprime 10 caracteres por cada pulgada horizontal. Por lo tanto, su programa deberá imprimir 6 1/2 pulgadas de texto, es decir, 65 caracteres por línea.

8.36 (*Impresión de fechas en varios formatos*). En la correspondencia de negocios, las fechas se imprimen comúnmente en varios formatos diferentes. Dos de los formatos más comunes son:

```
07/21/55 and July 21, 1955
```

Escriba un programa que lea una fecha en el primer formato y la imprima en el segundo.

8.37 (*Protección de cheques*). Con frecuencia las computadoras se utilizan en sistemas de escritura de cheques, como son aplicaciones de nóminas y cuentas por pagar. Circulan muchas historias raras, en relación con la impresión de nóminas semanales impresas (por error) por cantidades en exceso de 1 millón. Debido a fallas humanas o de máquina, cantidades rarísimas han sido impresas por sistemas computarizados de escritura de cheques. Los diseñadores de estos sistemas, naturalmente, hacen todo tipo de esfuerzo para incorporar controles en sus sistemas, a fin de evitar que se emitan cheques equivocados.

Otro problema serio es la alteración intencional de la cantidad en el cheque, por alguien que intenta cobrar un cheque de forma fraudulenta. Para evitar que sea alterada una cantidad en dólares, la mayor parte de los sistemas computarizados de escritura de cheques emplean una técnica conocida como *protección de cheques*.

Los cheques diseñados para impresión por computadora contienen un número fijo de espacios, en el cual la computadora puede imprimir una cantidad. Suponga que un cheque de nómina contiene ocho espacios en blanco, en el cual la computadora se supone imprimirá la cantidad de la nómina semanal. Si la cantidad es grande, entonces todos los ocho espacios quedarán llenos, por ejemplo:

```
1,230.60   (check amount)
--------
12345678   (position numbers)
```

Por otra parte, si la cantidad es menor de $1000, entonces varios de los espacios quedarían en blanco. Por ejemplo,

```
   99.87
--------
12345678
```

contiene tres espacios en blanco. Si un cheque se imprime con espacios en blanco, es más fácil que alguien pueda alterar la cantidad del cheque. A fin de evitar que se modifique un cheque, muchos sistemas de escritura de cheques *presentan asteriscos*, para proteger la cantidad, como sigue:

```
***99.87
--------
12345678
```


Escriba un programa que introduzca una cantidad en dólares a imprimirse en un cheque, y a continuación imprima en formato protegido de cheque con asteriscos anteriores, si ello fuera necesario. Suponga que para la impresión de una cantidad están disponibles nueve espacios.

8.38 (*Escribir el equivalente en palabras de una cantidad de cheques*). Continuando con el análisis del ejemplo anterior, insistimos en la importancia de diseñar sistemas de escritura de cheques, que impidan la modificación de las cantidades de los cheques. Un método común de seguridad requiere que la cantidad de cheque se escriba tanto en números como en palabras. Aun si alguien es capaz de modificar la cantidad numérica del cheque, resulta en extremo, difícil modificar la cantidad en palabras.

Muchos sistemas computarizados de escritura de cheques no incluyen la cantidad del cheque en palabras. Quizás la razón principal de esta omisión es el hecho de que la mayoría de los lenguajes de alto nivel utilizados en aplicaciones comerciales, no contienen características adecuadas de manipulación de cadenas. Otra razón es que la lógica para la escritura de equivalentes en palabras de las cantidades de los cheques es algo compleja.

Escriba un programa en C que introduzca una cantidad de cheque numérico y escriba el equivalente en palabras de dicha cantidad. Por ejemplo, la cantidad 112.43 deberá quedar escrita como

```
ONE HUNDRED TWELVE and 43/100
```

8.39 (*Código Morse*). Quizás el más famoso de todos los sistemas de codificación es el código Morse, desarrollado por Samuel Morse en 1832, para uso en el sistema telegráfico. El código Morse asigna una serie de puntos y rayas a cada letra del alfabeto, a cada dígito y a unos cuantos caracteres especiales (como el punto, coma, punto y coma; y dos puntos). En los sistemas orientados a sonido, el punto representa un sonido corto y la raya representa un sonido largo. Se utilizan otras representaciones de los puntos y rayas, con sistemas orientados a luz y sistemas de señalización por banderas.

La separación entre palabras se indica por un espacio, o por la ausencia de un punto o de una raya. En un sistema orientado a sonidos, un espacio queda indicado por un corto periodo de tiempo, en el cual ningún sonido se transmite. La versión internacional del código Morse aparece en la figura 8.39.

Escriba un programa que lea una frase en lengua inglesa y que cifre la frase en código Morse. También escriba un programa que lea una frase en código Morse y la convierta en el equivalente en lengua inglesa. Utilice un espacio en blanco entre cada letra codificada Morse y tres espacios en blanco entre cada palabra codificada en Morse.

8.40 (*Programa de conversión métrica*). Escriba un programa que ayude al usuario con las conversiones métricas. Su programa deberá permitir al usuario especificar como cadenas los nombres de las unidades (es decir, centímetros, litros, gramos, etcétera para el sistema métrico y pulgadas, cuartos, libras, etcétera para el sistema inglés) y deberá responder a preguntas sencillas como

```
"¿How many inches are in 2 meters?"
"¿How many liters are in 10 quarts?"
```

Su programa deberá poder reconocer conversiones inválidas. Por ejemplo, la pregunta

```
"¿How many feet in 5 kilograms?"
```

no tiene sentido, porque `"feet"` son unidades de longitud, en tanto que `"kilogram"` es una unidad de peso.

8.41 (*Cartas de cobranza*). Muchos negocios gastan mucho tiempo y dinero cobrando deudas vencidas. La cobranza de cuentas vencidas es el proceso de efectuar demandas repetidas o insistentes por carta a un deudor, en un intento de cobrar una deuda.

A menudo se utilizan computadoras para generar automáticamente cartas de cobranza, en grado creciente de severidad, conforme la cuenta se hace más vieja. La teoría es que mientras más vieja sea la cuenta, más difícil será su cobranza, por lo tanto, las cartas correspondientes deberán ser más y más amenazadoras.

Carácter	Código	Carácter	Código
A	.-	T	-
B	-...	U	..-
C	-.-.	V	...-
D	-..	W	.--
E	.	X	-..-
F	..-.	Y	-.--
G	--.	Z	--..
H	....		
I	..	**Digits**	
J	.---	1	.----
K	-.-	2	..---
L	.-..	3	...--
M	--	4	....-
N	-.	5	.....
O	---	6	-....
P	.--.	7	--...
Q	--.-	8	---..
R	.-.	9	----.
S	...	0	-----

Fig. 8.39 Las letras del alfabeto tal y como se expresan en el código internacional Morse.

Escriba un programa en C que contenga los textos de cinco cartas de cobranza, de severidad creciente. Su programa deberá aceptar como entrada:

1. El nombre del deudor.
2. La dirección del deudor.
3. La cuenta del deudor.
4. La cantidad adeudada.
5. El atraso en la cantidad adeudada (es decir, un mes de vencida, dos meses de vencida, etcétera).

Utilice el atraso en el pago para seleccionar uno de los cinco textos de mensaje, y a continuación imprima la carta de cobranza, insertando donde resulte apropiada la otra información proporcionada por el usuario.

Un proyecto de manipulación de cadenas que resulta un reto

8.42 (*Un generador de palabras cruzadas*). La mayor parte de las personas alguna vez han resuelto un crucigrama, pero pocos han intentado generar uno. Generar un crucigrama es un problema difícil. Se sugiere aquí como un proyecto de manipulación de cadenas, que requiere complejidad y esfuerzo sustancial. Existen muchos ángulos que el programador deberá resolver, para conseguir que funcione, inclusive el más simple

de los programas de generación de crucigramas. Por ejemplo, ¿cómo representa la cuadrícula de un crucigrama en la computadora? ¿Deberá utilizarse una serie de cadenas, o arreglos de doble subíndice? El programador necesitará una fuente de palabras (es decir, un diccionario computarizado) que pueda ser consultado de forma directa por el programa. ¿En qué forma deberán ser almacenadas estas palabras, para facilitar las manipulaciones complejas requeridas por el programa? El lector de verdad ambicioso deseará generar la porción de "indicios o pistas" en la cual se imprimen las breves sugerencias para cada palabra "horizontal" y cada palabra "vertical" para quien está resolviendo el crucigrama. Simplemente imprimir una versión del crucigrama en blanco por sí mismo no resulta un problema sencillo.

9

Entradas/salidas con formato

Objetivos

- Comprender flujos de entrada y de salida.
- Ser capaz de utilizar todas las capacidades de formato de impresión.
- Ser capaz de utilizar todas las capacidades de formato de entrada.

Todas las noticias posibles de imprimir.
Adolph S. Ochs

¿Qué loca quimera? ¿Qué lucha por escapar?
John Keats

No quiten las mojoneras en los límites de los campos.
Amenemope

El fin debe de justificar los medios.
Matthew Prior

Sinopsis

Resumen • Terminología • Errores comunes de programación • Prácticas sanas de programación • Sugerencias de portabilidad • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios.

9.1 Introducción

Una parte importante de la solución de cualquier problema es la presentación de los resultados. En este capítulo analizamos a fondo las características de formato de **printf** y de **scanf**. Estas funciones introducen *flujos de datos estándar de entrada* y extraen *flujos de datos estándar de salida*, respectivamente. Las otras cuatro funciones que utilizan entradas y salidas estándar **gets**, **puts**, **getchar**, y **putchar** fueron analizadas en el capítulo 8. Incluya el archivo de cabecera **<stdio.h>** en aquellos programas que llamen a estas funciones.

Muchas características de **printf** y de **scanf** han sido ya analizadas en el texto. Este capítulo resume estas características y presenta muchas otras. En el capítulo 11 se analizarán varias otras funciones incluidas en la biblioteca estándar de entrada/salidas (**stdio**).

9.2 Flujos

Toda entrada y salida se ejecuta mediante *flujos* —secuencias de caracteres organizadas en líneas. Cada línea está formada de cero o más caracteres y está terminada por el carácter de nueva línea. El estándar indica que las aplicaciones de ANSI C deben soportar líneas de por lo menos 254 caracteres, incluyendo un carácter de nueva línea de terminación.

Cuando empieza la ejecución del programa, de forma automática se conectan tres flujos al programa. Por lo regular, el flujo estándar de entrada se conecta al teclado y el flujo estándar de salida se conecta a la pantalla. Con frecuencia los sistemas operativos permiten que estos flujos sean *redirigidos* a otros dispositivos. Un tercer flujo, el *error estándar*, se conecta a la pantalla.

Los mensajes de error son extraídos o desplegados al flujo estándar de error. Los flujos se analizan en detalle en el capítulo 11, "Procesamiento de archivos".

9.3 Salida con formato utilizando printf

Utilizando **printf** se consigue un formato de salida preciso. Cada llamada **printf** contiene una *cadena de control de formato*, que describe el formato de la salida. La cadena de control de formato consiste de *especificadores de conversión, banderas, anchos de campo, precisiones y caracteres literales.* Junto con el signo de por ciento (%), éstos forman *especificaciones de conversión*. La función **printf** puede llevar a cabo las siguientes capacidades de formato, cada una de las cuales se analiza en este capítulo.

1. *Redondear* valores de punto flotante, a un número indicado de valores decimales.
2. *Alinear* una columna de números, con puntos decimales apareciendo uno por encima del otro.
3. *Salidas justificadas* a la *derecha* o a la *izquierda*.
4. *Insertar caracteres literales* en posiciones precisas en una línea de salida.
5. Representación en formato exponencial de números de punto flotante.
6. Representación en formato octal y hexadecimal de enteros no signados. Vea el apéndice E, "Sistemas numéricos" para más información sobre valores octales y hexadecimales.
7. Despliegue de todo tipo de datos con anchos de campo de tamaño fijo y precisiones.

La función **printf** tiene la forma:

printf (*format-control-string*, *other-arguments*) ;

Format-control-string describe el formato de salida, y *other-arguments* (estos son opcionales) corresponden a cada especificación de conversión existente en *format-control-string*. Cada especificación de conversión se inicia con un signo de por ciento y termina con un especificador de conversión. En una cadena de control de formato pueden existir muchas especificaciones de conversión.

Error común de programación 9.1

Olvidar encerrar entre comillas una cadena de control de formato.

Práctica sana de programación 9.1

Edite las salidas de manera nítida para su presentación. Esto hace más legible las salidas del programa y reduce errores de usuario.

9.4 Cómo imprimir enteros

Un entero es un número completo, como 776 o -52, que no contiene punto decimal. Los valores enteros se despliegan en uno de varios formatos. En la figura 9.1 se describe cada uno de los especificadores de conversión de enteros.

El programa de la figura 9.2 imprime un entero, utilizando cada uno de los especificadores de conversión de enteros. Note que sólo se imprime el signo menos; los signos más son suprimidos. Más adelante en este capítulo, veremos cómo obligar a que los signos más se impriman. También note que en una computadora con enteros de 2 bytes, el valor **-455** es leído por **%u** y se convierte al valor no signado **65081**.

Especificador de conversión	Descripción
`d`	Despliega un entero decimal signado.
`i`	Muestra un entero decimal signado. (Nota: los especificadores `i` y `d` son diferentes cuando se utilizan con `scanf`).
`o`	Muestra un entero octal no signado.
`u`	Muestra un entero decimal no signado.
`x o bien X`	Muestra un entero hexadecimal no signado. `X` mayúscula hace que sean mostrados los dígitos `0-9` y las letras `A-F`, y `x` minúsculas hacen que se muestren los dígitos `0-9` y `a-f`.
`h o bien l (letra l)`	Se coloca antes de cualquier especificador de conversión de enteros para indicar que se muestra un entero `short`, o bien `long` respectivamente.

Fig. 9.1 Especificadores de conversión de enteros.

```
/* Using the integer conversion specifiers */
#include <stdio.h>

main()
{
   printf("%d\n", 455);
   printf("%i\n", 455);  /* i same as d in printf */
   printf("%d\n", +455);
   printf("%d\n", -455);
   printf("%hd\n", 32000);
   printf("%ld\n", 2000000000);
   printf("%o\n", 455);
   printf("%u\n", 455);
   printf("%u\n", -455);
   printf("%x\n", 455);
   printf("%X\n", 455);
   return 0;
}
```

```
455
455
455
-455
32000
2000000000
707
455
65081
1c7
1C7
```

Fig. 9.2 Cómo utilizar especificadores de conversión de enteros.

Error común de programación 9.2

Imprimir un valor negativo utilizando un especificador de conversión que espera un valor no signado.

9.5 Cómo imprimir números de punto flotante

Un valor de punto flotante contiene un punto decimal, como en `33.5` o `657.983`. Los valores de punto flotante son desplegados en uno de varios formatos. En la figura 9.3 se describen los especificadores de conversión de punto flotante.

Los especificadores de conversión `e` y `E` despliegan valores de punto flotante en *notación exponencial*. La notación exponencial es el equivalente de computadora de la *notación científica* utilizada en matemáticas. Por ejemplo, el valor 150.4582 se representa en notación científica como

$$1.504582 \times 10^2$$

y se representa en notación exponencial como

```
1.504582E+02
```

por la computadora. Esta notación indica que `1.504582` debe ser multiplicado por `10` elevado a la segunda potencia (`E+02`). La `E` significa "exponente".

En forma preestablecida, los valores impresos con los especificadores de conversión `e`, `E`, y `f` son extraídos con 6 dígitos de precisión a la derecha del punto decimal; otras precisiones pueden ser especificadas de forma explícita. El especificador de conversión `f` siempre imprime por lo menos un dígito a la izquierda del punto decimal. Los especificadores de conversión `e`, y `E` imprimen respectivamente la `e` en minúsculas y la `E` en mayúsculas antes del exponente y siempre se imprimen exactamente un dígito a la izquierda del punto decimal.

El especificado de conversión `g` (`G`) imprime en formato `e` (`E`), o en formato `f`, sin ceros después del punto decimal (es decir, `1.234000` se imprime como `1.234`). Los valores se imprimen con `e` (`E`), si después de convertir el valor a notación exponencial, el exponente del valor es menor de `-4`, o es mayor o igual a la precisión especificada (por omisión, en el caso de `g` y de `G`, 6 dígitos significativos). De lo contrario, para imprimir el valor se utilizará el especificador de conversión `f`. Utilizando `g` o `G`, no se imprimen los ceros a la derecha del punto decimal, en la parte fraccionaria de la salida del valor. Para que se extraiga el punto decimal se requiere por lo menos de un dígito decimal.

Especificador de conversión	Descripción
`e o bien E`	Muestra un valor en punto flotante en notación exponencial.
`f`	Muestra valores en punto flotante.
`g o bien G`	Despliega un valor en punto flotante, ya sea en la forma de punto flotante `f`, o en la forma exponencial `e` (o `E`).
`L`	Se coloca antes de cualquier especificador de conversión de punto flotante, para indicar que está mostrado un valor de punto flotante `long double`.

Fig. 9.3 Especificadores de conversión de punto flotante.

Si se utiliza la especificación de conversión %g, los valores 0.0000875, 8750000.0, 8.75, 87.50 y 875 se imprimen como 8.75e-05, 8.75e+06, 8.75, 87.5 y 875. El valor 0.0000875 utiliza notación e porque, al convertirse a notación exponencial, su exponente es menor que -4. El valor 8750000.0 utiliza notación e debido a que su exponente es igual a la precisión por omisión.

La precisión de los especificadores de conversión g y G indican el número máximo de dígitos significativos impresos, incluyendo el dígito a la izquierda del punto decimal. Utilizando la especificación de conversión %g, el valor 1234567.0 se imprime como 1.23457e+06 (recuerde que todos los especificadores de conversión de punto flotante tienen una precisión por omisión de 6). Note que en el resultado aparecen 6 dígitos significativos. La diferencia entre g y G es idéntica a la diferencia entre e y E, cuando el valor se imprime en notación exponencial la g minúscula hace que la salida sea una e, y la G hace que la salida sea una E.

Práctica sana de programación 9.2

Al extraer datos, asegúrese que el usuario está consciente de situaciones en las cuales, debido al formato, los datos pudieran resultar imprecisos (por ejemplo, errores de redondeo debidos a precisiones especificadas).

El programa de la figura 9.4 demuestra cada una de las tres especificaciones de conversión de punto flotante. Advierta que las especificaciones de conversión %E y %g hacen que en la salida el valor se redondee.

```
/* Printing floating-point numbers with
   floating-point conversion specifiers */

#include <stdio.h>

main()
{
   printf("%e\n", 1234567.89);
   printf("%e\n", +1234567.89);
   printf("%e\n", -1234567.89);
   printf("%E\n", 1234567.89);
   printf("%f\n", 1234567.89);
   printf("%g\n", 1234567.89);
   printf("%G\n", 1234567.89);

   return 0;
}
```

```
1.234568e+06
1.234568e+06
-1.234568e+06
1.234568E+06
1234567.890000
1.23457e+06
1.23457E+06
```

Fig. 9.4 Cómo utilizar especificadores de conversión de punto flotante.

9.6 Cómo imprimir cadenas y caracteres

Los especificadores de conversión c y s se utilizan para imprimir respectivamente caracteres individuales y cadenas. El especificador de conversión c requiere de un argumento char. El especificador de conversión s requiere de un apuntador a char como argumento. El especificador de conversión s hace que se impriman caracteres hasta que encuentre un carácter de terminación NULL ('\0'). El programa que se muestra en la figura 9.5 despliega caracteres y cadenas con los especificadores de conversión c y s.

Error común de programación 9.3

*Utilizar %c para imprimir el primer carácter de una cadena. La especificación de conversión %c espera un argumento char. Una cadena es un apuntador a char, es decir, a char *.*

Error común de programación 9.4

Utilizar %s para imprimir un argumento char. La especificación de conversión %s espera un argumento de tipo apuntador a char. En algunos sistemas, esto causará un error fatal en tiempo de ejecución, conocido como violación de acceso.

Error común de programación 9.5

Es un error de sintaxis usar comillas sencillas alrededor de cadenas de caracteres. Las cadenas de caracteres deben estar encerradas entre comillas dobles.

Error común de programación 9.6

Usar comillas dobles alrededor de una constante de carácter. Esto, de hecho, crea una cadena formado por dos caracteres, el segundo de los cuales es el carácter de terminación NULL. Una constante de carácter es un carácter, encerrado entre comillas sencillas.

```
/* Printing strings and characters */
#include <stdio.h>

main()
{
   char character = 'A';
   char string[] = "This is a string";
   char *stringPtr = "This is also a string";

   printf("%c\n", character);
   printf("%s\n", "This is a string");
   printf("%s\n", string);
   printf("%s\n", stringPtr);
   return 0;
}
```

```
A
This is a string
This is a string
This is also a string
```

Fig. 9.5 Cómo utilizar los especificadores de conversión de caracteres y de cadenas.

9.7 Otros especificadores de conversión

Los tres especificadores de conversión restantes son **p**, **n** y % (figura 9.6).

Sugerencia de portabilidad 9.1

El especificador de conversión **p** *despliega una dirección de apuntador en forma de puesta en marcha definida (en muchos sistemas se utiliza notación hexadecimal en preferencia a notación decimal).*

El especificador de conversión **n** almacena el número de caracteres ya extraídos en el enunciado actual **printf**— el argumento correspondiente es un apuntador a una variable entera en la cual se almacena el valor. Mediante una especificación de conversión %**n** nada se imprime. El especificador de conversión % hace que se extraiga un signo de %.

En el programa de la figura 9.7, %**p** imprime el valor de **ptr** y la dirección de **x**; estos valores resultan idénticos porque a **ptr** es asignada a la dirección de **x**. A continuación, %**n** almacena el número de caracteres extraídos por el tercer enunciado **printf** en la variable entera **y**, y es impreso el valor de **y**. El último enunciado **printf** utiliza %% para imprimir el carácter % en una cadena de carácter. Note que cada una de las llamadas **printf** regresa un valor ya sea el número de caracteres extraídos o si ha ocurrido un error de salida, un valor negativo.

Error común de programación 9.7

Intentar imprimir un carácter literal de por ciento, utilizando en la cadena de control de formato % en vez de %%. Cuando en la cadena de control de formato aparece el signo de %, debe estar seguido por un especificador de conversión.

9.8 Cómo imprimir con anchos de campo y precisiones

El tamaño exacto de un campo en el cual se imprimen datos se especifica por el *ancho del campo*. Si el ancho del campo es mayor que los datos que se están imprimiendo, a menudo los datos dentro de dicho campo quedarán justificados a la derecha. Un entero que representa el ancho del campo es insertado en la especificación de conversión, entre el signo de por ciento (%) y el especificador de conversión. El programa en la figura 9.8 imprime dos grupos cada uno de cinco números, justificando a la derecha aquellos números que contienen menos dígitos que el ancho del campo. Note que el ancho de campo es de manera automática aumentado para imprimir valores más anchos que el campo, y que el signo de menos de un valor negativo ocupa en el ancho de campo

Especificador de conversión	Descripción
p	Muestra un valor de apuntador en forma de puesta en marcha definida.
n	Almacena el número de caracteres ya extraídos en el enunciado **printf** actual. Se proporciona un apuntador a un entero como argumento correspondiente. No se muestra nada.
%	Muestra el carácter de por ciento.

Fig. 9.6 Otros especificadores de conversión.


```
/* Using the p, n, and % conversion specifiers */
#include <stdio.h>

main()
{
   int *ptr;
   int x = 12345, y;

   ptr = &x;
   printf("The value of ptr is %p\n", ptr);
   printf("The address of x is %p\n\n", &x);

   printf("Total characters printed on this line is:%n", &y);
   printf(" %d\n\n", y);

   y = printf("This line has 28 characters\n");
   printf("%d characters were printed\n\n", y);

   printf("Printing a %% in a format control string\n");
   return 0;
}
```

```
The value of ptr is 001F2BB4
The address of x is 001F2BB4

Total characters printed on this line is: 41

This line has 28 characters
28 characters were printed

Printing a % in a format control string
```

Fig. 9.7 Cómo utilizar los especificadores de conversión **p**, **n** y %.

una posición de carácter. Los anchos de campo pueden ser utilizados con todos los especificadores de conversión.

Error común de programación 9.8

No proporcionar un ancho de campo lo suficiente extenso para manejar un valor a imprimirse. Esto puede desplazar otros datos imprimiéndose y puede producir salidas confusas. ¡Familiarícese con sus datos!

La función **printf** también da la capacidad de definir la *precisión* con la cual los datos se imprimirán. La precisión tiene significados distintos para diferentes tipos de datos. Cuando se utiliza con especificadores de conversión de enteros, la precisión indica el número mínimo de dígitos a imprimirse. Si el valor impreso contiene menos dígitos que la precisión especificada, al valor impreso se le antepondrán ceros, hasta que el número total de dígitos sea equivalente a la precisión. La precisión por omisión para enteros es 1. Cuando se utiliza con especificadores de conversión de punto flotante **e**, **E** y **f**, la precisión es el número de dígitos que aparecerá después del punto decimal. Cuando se utilice con los especificadores de conversión **g** y **G**, la precisión es

```
/* Printing integers right-justified */
#include <stdio.h>

main()
{
   printf("%4d\n", 1);
   printf("%4d\n", 12);
   printf("%4d\n", 123);
   printf("%4d\n", 1234);
   printf("%4d\n\n", 12345);

   printf("%4d\n", -1);
   printf("%4d\n", -12);
   printf("%4d\n", -123);
   printf("%4d\n", -1234);
   printf("%4d\n", -12345);

   return 0;
}
```

```
   1
  12
 123
1234
12345

  -1
 -12
-123
-1234
-12345
```

Fig. 9.8 Justificación de enteros a la derecha en un campo.

el número máximo de dígitos significativos a imprimirse. Cuando se utiliza con el especificador de conversión **s**, la precisión es el número máximo de caracteres a escribirse a partir de una cadena. Para utilizar la precisión, coloque un punto decimal (.) seguido por un entero que representa la precisión, entre el signo de por ciento y el especificador de conversión. El programa de la figura 9.9 demuestra el uso de la precisión en cadenas de control de formato. Note que cuando un valor en punto flotante se imprime con una precisión menor en el valor que el número original de decimales, el valor quedará redondeado.

Se pueden combinar el ancho de campo y la precisión, colocando el ancho de campo seguido por el punto decimal, seguido por la precisión, entre el signo de por ciento y el especificador de conversión, como en el enunciado

```
printf("%9.3f", 123.456789);
```

que despliega **123.4567** con 3 dígitos a la derecha del punto decimal, y queda justificado a la derecha, en un campo de 9 dígitos.


```
/* Using precision while printing integers,
   floating-point numbers, and strings */
#include <stdio.h>

main()
{
   int i = 873;
   float f = 123.94536;
   char s[] = "Happy Birthday";

   printf("Using precision for integers\n");
   printf("\t%.4d\n\t%.9d\n\n", i, i);
   printf("Using precision for floating-point numbers\n");
   printf("\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f);
   printf("Using precision for strings\n");
   printf("\t%.11s\n", s);
   return 0;
}
```

```
Using precision for integers
        0873
        000000873

Using precision for floating-point numbers
        123.945
        1.239e+02
        124

Using precision for strings
        Happy Birth
```

Fig. 9.9 Cómo usar precisiones para mostrar información de varios tipos.

Utilizando expresiones enteras en la lista de argumentos, a continuación de la cadena de control de formato, es posible especificar el ancho de campo y la precisión. Para utilizar esta característica, inserte un * (asterisco) en el lugar del ancho de campo o de la precisión (o de ambos). El argumento coincidente en la lista de argumentos se evalúa y se utiliza en lugar del asterisco. El valor del argumento puede ser negativo para el ancho de campo, pero debe ser positivo para la precisión. Un valor negativo para el ancho de campo hace que la salida se justifique a la izquierda en el campo, tal y como se describe en la sección siguiente. El enunciado

```
printf("%*.*f", 7, 2, 98.736);
```

utiliza **7** para el ancho de campo, **2** para la precisión y extrae el valor **98.74** justificado a la derecha.

9.9 Uso de banderas en la cadena de control de formato de printf

La función **printf** tiene también *banderas* para complementar sus capacidades de formato de salida. Están disponibles para el usuario cinco banderas, para uso en cadenas de control de formato (figura 9.10).

Bandera	Descripción
- (signo de menos)	Justificación a la izquierda de la salida, dentro del campo especificado.
+ (signo más)	Despliega un signo más, antes de valores positivos y un signo menos, antes de valores negativos.
espacio	Imprime un espacio antes de un valor positivo que no se imprima con la bandera +.
#	Antecede un 0 al valor extraído, cuando se utiliza con el especificador de conversión octal o.
	Antecede 0x o 0X al valor de salida, cuando se utiliza con los especificadores de conversión hexadecimales x o X.
	Obliga a un punto decimal para un número de punto decimal impreso con e, E, f, g o G, que no contenga una parte fraccionaria. (Por lo regular sólo se imprime el punto decimal si le sigue algún dígito). En el caso de los especificadores g y G, no se eliminan los ceros a la derecha.
0 (cero)	Rellena un campo con ceros a la izquierda.

Fig. 9.10 Banderas de cadenas de control de formato.

Para utilizar una bandera en una cadena de control de formato, colóquela de inmediato a la derecha del signo de por ciento. En una especificación de conversión se pueden combinar varias banderas.

El programa de la figura 9.11 demuestra la justificación a la derecha y a la izquierda de una cadena, de un entero, de un carácter y de un número de punto flotante.

El programa de la figura 9.12 imprime un número positivo y uno negativo, cada uno de ellos con y sin la bandera +. Note que en ambos casos el signo de menos queda desplegado, pero el signo de más sólo es desplegado cuando se utiliza la bandera más.

El programa de la figura 9.13 antepone un espacio al número positivo con la bandera de espacio. Esto resulta útil para alinear números positivos y negativos con el mismo número de dígitos.

```
/* Right justifying and left justifying values */
#include <stdio.h>

main()
{
   printf("%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23);
   printf("%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23);
   return 0;
}
```

```
     hello         7         a  1.230000

hello     7         a         1.230000
```

Fig. 9.11 Cómo justificar a la izquierda cadenas en un campo.


```
/* Printing numbers with and without the + flag */
#include <stdio.h>

main()
{
   printf("%d\n%d\n", 786, -786);
   printf("%+d\n%+d\n", 786, -786);
   return 0;
}
```

```
786
-786
+786
-786
```

Fig. 9.12 Cómo imprimir números positivos y negativos con y sin la bandera +.

```
/* Printing a space before signed values
   not preceded by + or - */
#include <stdio.h>

main()
{
   printf("% d\n% d\n", 547, -547);
   return 0;
}
```

```
 547
-547
```

Fig. 9.13 Cómo utilizar la bandera espacio.

El programa de la figura 9.14 utiliza la bandera # para anteceder 0 al valor octal, 0x y 0X a valores hexadecimales, y para obligar a el punto decimal en un valor impreso utilizando a g.

El programa en la figura 9.15 combina la bandera + y la bandera 0 (cero) para imprimir 452 en un campo de 9 espacios, con un signo de + y ceros a la izquierda, y a continuación vuelve a imprimir 452 utilizando sólo la bandera 0 y el campo de 9 espacios.

9.10 Cómo imprimir literales y secuencias de escape

La mayor parte de los caracteres literales a imprimirse en un enunciado **printf** pueden ser incluidos en la cadena de control de formato. Sin embargo, existen varios caracteres "problema" tal y como los signos de comillas ("), que delimitan la cadena de control de formato misma. Varios caracteres de control, como son la nueva línea y el tabulador, deben ser representados por *secuencias de escape*. Una secuencia de escape es representada por una diagonal invertida (\) seguida por un carácter de escape particular. La tabla en la figura 9.16 enlista todas las secuencias de escape y las acciones que éstas causan.

UNIVERSIDAD DE LA REPUBLICA
FACULTAD DE INGENIERIA
DEPARTAMENTO

```
/* Using the # flag with conversion specifiers
   o, x, X, and any floating-point specifier */
#include <stdio.h>

main()
{
   int c = 1427;
   float p = 1427.0;

   printf("%#o\n", c);
   printf("%#x\n", c);
   printf("%#X\n", c);
   printf("\n%g\n", p);
   printf("%#g\n", p);
   return 0;
}
```

```
02623
0x593
0X593

1427
1427.00
```

Fig. 9.14 Cómo utilizar la bandera # .

```
/* Printing with the 0(zero) flag fills in leading zeros */
#include <stdio.h>

main()
{
   printf("%+09d", 452);
   printf("%09d", 452);
   return 0;
}
```

```
+00000452
000000452
```

Fig. 9.15 Cómo utilizar la bandera 0 (cero).

Error común de programación 9.9

Intentar imprimir como datos literales en un enunciado `printf` *una comilla, dobles comillas, signo de interrogación o un carácter de diagonal invertida, sin anteceder dichos caracteres por una diagonal invertida para formar una secuencia de escape correcta.*

Secuencia de escape	Descripción
\'	Salida del carácter de una sola comilla (').
\"	Salida del carácter de dobles comillas (").
\?	Salida del signo de interrogación (?).
\\	Salida del carácter de diagonal invertida (\).
\a	Genera una alerta visual o audible (campana).
\b	Mueve el cursor hacia atrás una posición en la línea actual.
\f	Mueve el cursor al inicio de la siguiente página lógica.
\n	Mueve el cursor al inicio de la línea siguiente.
\r	Mueve el cursor al principio de la línea actual.
\t	Mueve el cursor a la siguiente posición de tabulador horizontal.
\v	Mueve el cursor a la siguiente posición de tabulador vertical.

Fig. 9.16 Secuencias de escape.

9.11 Formato de entrada utilizando scanf

Se consigue entrada con formato preciso utilizando **`scanf`**. Cada enunciado **`scanf`** contiene una cadena de control de formato que describe el formato de los datos que se introducen. La cadena de control está formada de especificaciones de conversión y de caracteres literales. La función **`scanf`** tiene las siguientes capacidades de formato de entrada:

1. Entrada de todo tipo de datos.
2. Entrada de caracteres específicos desde un flujo de entrada.
3. Omitir caracteres específicos del flujo de entrada.

La función **`scanf`** se escribe en la forma siguiente:

`scanf` (*format-control-string, other-arguments*) ;

Format-control-string describe los formatos de la entrada, y *other-arguments* son apuntadores a variables, en los cuales se almacena la entrada.

Práctica sana de programación 9.3

Al introducir datos, solicite al usuario un elemento o pocos elementos de datos a la vez. Evite solicitar al usuario la introducción de muchos elementos de datos, en respuesta a una solicitud.

En la figura 9.17 se resumen los especificadores de conversión utilizados para introducir todos los tipos de datos. El resto de esta sección presenta programas que demuestran la lectura de datos utilizando los varios especificadores de conversión de **`scanf`**.

El programa en la figura 9.18 lee enteros con varios especificadores de conversión de enteros, y despliega los enteros como números decimales. Note que **`%i`** es capaz de introducir enteros decimales, octales y hexadecimales.

Especificador de conversión	Descripción
Enteros	
`d`	Lee un entero decimal, opcionalmente signado. El argumento correspondiente es un apuntador a un entero.
`i`	Lee un entero decimal, octal o hexadecimal, opcionalmente signado. El argumento correspondiente es un apuntador a un entero.
`o`	Lee un entero octal. El argumento correspondiente es un apuntador a un entero no signado.
`u`	Lee un entero decimal no signado. El argumento correspondiente es un apuntador a un entero no signado.
`x` o `X`	Lee un entero hexadecimal. El argumento correspondiente es un apuntador a un entero no signado.
`h` o `l`	Se coloca antes de cualquiera de los especificadores de conversión de enteros, para indicar que un entero **`short`** o **`long`** será introducido.
Números de punto flotante	
`e, E, f, g,` o `G`	Lee un valor en punto flotante. El argumento correspondiente es un apuntador a una variable de punto flotante.
`l` o `L`	Se coloca delante de cualquier especificador de conversión de punto flotante para indicar que un valor **`double`** o **`long double`** será introducido.
Caracteres y cadenas.	
`c`	Lee un carácter. El argumento correspondiente es un apuntador a **`char`** y **`NULL`** (`'\0'`) no se agrega.
`s`	Lee una cadena. El argumento correspondiente es un apuntador a un arreglo del tipo **`char`**, que es lo suficiente extenso para contener la cadena y un carácter de terminación **`NULL`** (`'\0'`).
Conjunto de rastreo *[scan characters]*	Rastrea una cadena buscando un conjunto de caracteres almacenados en un arreglo.
Misceláneos	
`p`	Lee una dirección de apuntador producido de la misma forma que cuando una dirección es extraída con `%p` en un enunciado **`printf`**.
`n`	Almacena el número de caracteres introducidos hasta el momento en este **`scanf`**. El argumento correspondiente es un apuntador a entero.
`%`	Saltarse un signo de por ciento (%) en la entrada.

Fig. 9.17 Especificadores de conversión para **`scanf`**.

Al introducir números de punto flotante, cualquiera de los especificadores de conversión de punto flotante **`e`**, **`E`**, **`f`**, **`g`**, o **`G`** pueden ser utilizados. El programa de la figura 9.19 demuestra la lectura de tres números de punto flotante, con cada uno de los tres tipos de los especificadores flotantes de conversión y muestra todos los tres números mediante el especificador de conversión **`f`**. Note que la salida del programa confirma el hecho que los valores del punto flotante no son precisos —este hecho queda resaltado en el segundo valor impreso.


```
/* Reading integers */
#include <stdio.h>

main()
{
   int a, b, c, d, e, f, g;

   printf("Enter seven integers: ");
   scanf("%d%i%i%i%o%u%x", &a, &b, &c, &d, &e, &f, &g);
   printf("The input displayed as decimal integers is:\n");
   printf("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
   return 0;
}
```

```
Enter seven integers: -70 -70 070 0x70 70 70 70
The input displayed as decimal integers is:
-70 -70 56 112 56 70 112
```

Fig. 9.18 Cómo leer entradas con especificadores de conversión a enteros.

```
/* Reading floating-point numbers */
#include <stdio.h>

main()
{
   float a, b, c;

   printf("Enter three floating-point numbers: \n");
   scanf("%e%f%g", &a, &b, &c);
   printf("Here are the numbers entered in plain\n");
   printf("floating-point notation:\n");
   printf("%f %f %f\n", a, b, c);
   return 0;
}
```

```
Enter three floating-point numbers:
1.27987 1.27987e+03 3.38476e-06
Here are the numbers entered in plain
floting-point notation:
1.279870
1279.869995
0.000003
```

Fig. 9.19 Cómo leer entradas con especificadores de conversión de punto flotante.

Los caracteres y cadenas se introducen utilizando los especificadores de conversión **`c`** y **`s`**, respectivamente. El programa de la figura 9.20 solicita al usuario que introduzca una cadena. El programa introduce el primer carácter de la cadena utilizando `%c` y lo almacena en la variable de

carácter **x**, y a continuación introduce el resto de la cadena, utilizando **%s** y lo almacena en el arreglo de caracteres **y**.

Se puede introducir una secuencia de caracteres utilizando un *conjunto de rastreo*. Un conjunto de rastreo es un conjunto de caracteres encerrados en corchetes [], y precedidos por un signo por ciento en la cadena de control de formato. Un conjunto de rastreo rastrea los caracteres en el flujo de entrada, buscando sólo aquellos caracteres que coincidan con los caracteres contenidos en el conjunto de rastreo. Cada vez que se encuentre un carácter, se almacena en el argumento correspondiente del conjunto de caracteres que es un apuntador a un arreglo de caracteres. El conjunto de rastreo deja de introducir caracteres cuando encuentra un carácter que no esté contenido en el conjunto de rastreo. Si el primer carácter en el flujo de entrada no coincide con un carácter en el conjunto de rastreo, sólo se almacenará en el arreglo el carácter nulo. El programa en la figura 9.21 utiliza un conjunto de rastreo **[aeiou]** para rastrear el flujo de entrada buscando vocales. Observe que se leerán las primeras siete letras de la entrada. La octava letra (**h**) no forma parte del conjunto de rastreo y, por lo tanto, el rastreo terminará.

El conjunto de rastreo también puede ser utilizado para rastrear buscando caracteres que no estén contenidos en el mismo, utilizando un *conjunto de rastreo invertido*. Para crear un conjunto de rastreo invertido, coloque un *acento circunflejo* (^) en los corchetes antes de los caracteres de rastreo. Esto hará que sean almacenados los caracteres que no aparecen en el conjunto de rastreo. Cuando se detecte un carácter contenido en el conjunto de rastreo invertido, se terminará la entrada. El programa en la figura 9.22 utiliza el conjunto de rastreo invertido **[^aeiou]** para buscar consonantes o más apropiadamente para buscar "no vocales".

En la especificación de conversión **scanf** se puede utilizar un ancho de campo para leer un número específico de caracteres a partir del flujo de entrada. El programa de la figura 9.23 introduce una serie de dígitos consecutivos como un entero de dos dígitos y un entero que consiste de los dígitos restantes en el flujo de entrada.

```
/* Reading characters and strings */
#include <stdio.h>

main()
{
   char x, y[9];

   printf("Enter a string: ");
   scanf("%c%s", &x, y);

   printf("The input was:\n");
   printf("the character \"%c\" ", x);
   printf("and the string \"%s\"\n", y);
   return 0;
}
```

```
Enter a string: Sunday
The input was:
the character "S" and the string "unday"
```

Fig. 9.20 Cómo introducir caracteres y cadenas.


```
/* Using a scan set */
#include <stdio.h>

main()
{
   char z[9];

   printf("Enter string: ");
   scanf("%[aeiou]", z);
   printf("The input was \"%s\"\n", z);
   return 0;
}
```

```
Enter String: ooeeooahah
The input was "ooeeooa"
```

Fig. 9.21 Cómo usar un conjunto de rastreo.

```
/* Using an inverted scan set */
#include <stdio.h>

main()
{
   char z[9];

   printf("Enter a string: ");
   scanf("%[^aeiou]", z);
   printf("The input was \"%s\"\n", z);
   return 0;
}
```

```
Enter a string: String
The input was "Str"
```

Fig. 9.22 Cómo utilizar un conjunto de rastreo invertido.

A menudo es necesario omitir ciertos caracteres del flujo de entrada. Por ejemplo, una fecha podía ser introducida como

```
7-9-91
```

Es necesario almacenar cada número de la fecha, pero los guiones que separan los números pueden ser descartados. A fin de eliminar caracteres innecesarios, inclúyalos en la cadena de control de formato de **scanf** (caracteres de espacio en blanco —como es el espacio en blanco, la nueva línea y el tabulador— pasar por alto todos los espacios en blanco a la izquierda). Por ejemplo, para pasar por alto los guiones en la entrada, utilice el enunciado

```
scanf("%d-%d-%d", &month, &day, &year);
```

```
/* inputting data with a field width */
#include <stdio.h>

main()
{
   int x, y;

   printf("Enter a six digit integer: ");
   scanf("%2d%d", &x, &y);
   printf("The integers input were %d and %d\n", x, y);
   return 0;
}
```

```
Enter a six digit integer: 123456
The integers input were 12 and 3456
```

Fig. 9.23 Cómo introducir datos con un ancho de campo.

Aunque este **scanf** sí elimina los guiones de la entrada anterior, es posible que la fecha hubiera sido escrita como

```
7/9/91
```

En este caso el **scanf** precedente no eliminaría caracteres innecesarios. Por esta razón, **scanf** proporciona un *carácter de supresión de asignación* *. El carácter de supresión de asignación, le permite a **scanf** leer cualquier tipo de datos a partir de la entrada y descartarlos sin asignarlos a una variable. El programa de la figura 9.24 utiliza el carácter de supresión de asignación en la especificación de conversión **%c**, para indicar que un carácter que aparece en el flujo de entrada deberá ser leído y descartado. Sólo se almacenará mes, día y año. Los valores de las variables se imprimen para demostrar que de hecho han sido introducidos en forma correcta. Note que ninguna variable en la lista de argumentos corresponde a las especificaciones de conversión que utiliza el carácter de supresión de asignación, porque con estas especificaciones de conversión no se efectúa ninguna asignación.

Resumen

- Toda entrada y salida se maneja a través de flujos —secuencias de caracteres organizadas en línea. Cada línea consiste de cero o más caracteres y terminan con el carácter de nueva línea.
- Por lo regular, el flujo estándar de entrada se conecta al teclado, y el flujo estándar de salida está conectado a la pantalla de la computadora.
- Los sistemas operativos a menudo permiten que los flujos de entrada y salida estándar, sean redirigidos a otros dispositivos.
- La cadena de control de formato **printf** describe el formato o formatos en los cuales aparecerán los valores de salida. La cadena de control de formato está formada de los especificadores de conversión, las banderas, los anchos de campo, las precisiones y los caracteres literales.


```
/* Reading and discarding characters from the input stream */
#include <stdio.h>

main()
{
   int month1, day1, year1, month2, day2, year2;

   printf("Enter a date in the form mm-dd-yy: ");
   scanf("%d%*c%d%*c%d", &month1, &day1, &year1);
   printf("month = %d  day = %d  year = %d\n\n",
      month1, day1, year1);
   printf("Enter a date in the form mm/dd/yy: ");
   scanf("%d%*c%d%*c%d", &month2, &day2, &year2);
   printf("month = %d  day = %d  year = %d\n",
      month2, day2, year2);
   return 0;
}
```

```
Enter a date in the form mm-dd-yy: 11-18-71
month = 11  day = 18  year = 71

Enter a date in the form mm/dd/yy: 11/18/71
month = 11  day = 18  year = 71
```

Fig. 9.24 Cómo leer y descartar caracteres del flujo de entrada.

- Los enteros se imprimen con los especificadores de conversión siguientes : **d** o **i** para enteros opcionalmente signados, o para enteros no signados en forma octal, **u** para enteros no signados en forma decimal, y **x** o bien **X** para enteros no signados en forma hexadecimal. Para indicar un entero **short** o **long** respectivamente a los especificadores de conversión ya descritos, se antecede el modificador **h** o **l**.
- Se imprimen los valores en punto flotante utilizando los especificadores de conversión siguientes: **e** o **E** en el caso de notación exponencial, **f** para la notación de punto flotante normal y **g** o **G** para ya sea **e** o (**E**) o para la notación **f**. Cuando se indica el especificador de conversión **g** o (**G**), si el exponente del valor es menor de **-4** o mayor o igual que la precisión con la cual se imprime el valor, se utiliza el especificador de conversión **e** o (**E**).
- La precisión para los especificadores de conversión **g** y **G** indica el número máximo de dígitos significativos que se imprimirán.
- El especificador de conversión **c** imprime un carácter.
- El especificador de conversión **s** imprime una cadena de caracteres terminadas por un carácter nulo.
- El especificador de conversión **p** despliega una dirección de apuntador en forma de puesta en marcha definida (en muchos sistemas se utiliza la notación hexadecimal).
- El especificador de conversión **n** almacena el número de caracteres ya salido en el enunciado actual **printf**. El argumento correspondiente es un apuntador a un entero.

- El especificador de conversión %% hace que salga un % literal.
- Si el ancho de campo es mayor que el objeto que se está imprimiendo, el objeto quedará justificado a la derecha dentro de dicho campo.
- Los anchos de campo pueden ser utilizados por todos los especificadores de conversión.
- La precisión utilizada con los especificadores de conversión enteros indican el número mínimo de dígitos a imprimirse. Si el valor contiene menos dígitos que la precisión especificada, se antecederán ceros al valor impreso, hasta que el número de dígitos sea equivalente a la precisión.
- La precisión utilizada con especificadores de conversión de punto flotante **e**, **E**, y **f** indica el número de dígitos que aparecerán después del punto decimal.
- La precisión utilizada con especificadores de conversión de punto flotante **g** y **G** indica el número de dígitos significativos que aparecerán.
- La precisión utilizada con el especificador de conversión s indica el número de caracteres a imprimirse.
- El ancho de campo y la precisión pueden ser combinados, colocando el ancho de campo seguido por un punto decimal y a su vez seguido por la precisión, entre el % y el especificador de conversión.
- Es posible especificar el ancho de campo y la precisión mediante expresiones enteras en la lista de argumentos, a continuación de la cadena de control de formato. Para utilizar esta característica, inserte un * (asterisco) en lugar del ancho de campo o de la precisión. El argumento coincidente en la lista de argumentos se evalúa y se utilizará en lugar del asterisco. El valor del argumento puede ser negativo para el ancho de campo, pero para la precisión deberá ser positivo.
- La bandera signo de - justifica a la izquierda su argumento dentro de un campo.
- La bandera + imprime un signo más para valores positivos y un signo menos para valores negativos.
- La bandera de espacio imprime un espacio antes de un valor positivo, que no esté mostrado con la bandera de +.
- La bandera de signo de # antecede un 0 a los valores octal, **0x** o **0X** a los valores hexadecimales, y obliga a que se imprima el punto decimal en el caso de valores de punto flotante impresos con **e**, **E**, **f**, **g**, o **G** (normalmente el punto decimal se desplegará sólo si el valor contiene una parte fraccionaria).
- La bandera 0 imprime ceros a la izquierda para un valor que no ocupa todo su ancho de campo.
- Se consigue un formato preciso de entrada utilizando la función de biblioteca **scanf**.
- Los enteros se introducen mediante los especificadores de conversión **d** e **i** para enteros opcionalmente signados, y **o**, **u**, **x**, o **X** para enteros no signados. Para introducir un entero **short** o bien **long** respectivamente, se colocan los modificadores **h** y **l** antes de un especificador de conversión entero.
- Los valores de punto flotante son introducidos con los especificadores de conversión **e**, **E**, **f**, **g**, o **G**. Se colocan los modificadores **l** y **L** antes de cualquiera de los especificadores de conversión de punto flotante, para indicar que el valor introducido es un valor **double**, o bien **long double**, respectivamente.
- Los caracteres se introducen mediante el especificador de conversión **c**.
- Las cadenas se introducen mediante el especificador de conversión **s**.

- Un conjunto de rastreo, rastrea los caracteres en la entrada buscando sólo aquellos caracteres que coincidan con los caracteres que contiene el conjunto de rastreo. Cuando se encuentra un carácter, se almacena un arreglo de caracteres. El conjunto de rastreo deja de introducir caracteres cuando es encontrado un carácter no contenido en el conjunto de rastreo.
- Para crear un conjunto de rastreo invertido, coloque un acento circunflejo (^) en los corchetes, antes de los caracteres de rastreo. Esto hace que los caracteres que no aparecen en el conjunto de rastreo sean almacenados hasta que se encuentre un carácter contenido en el conjunto de rastreo invertido.
- Los valores de dirección se introducen utilizando el especificador de conversión **p**.
- El especificador de conversión **n** almacena el número de caracteres ya introducido en el **scanf** actual. El argumento correspondiente es un apuntador a **int**.
- La especificación de conversión %% hace coincidir en la entrada un carácter único de %.
- El carácter de supresión de asignación se utiliza para leer datos del flujo de entrada y descartar dichos datos.
- Un ancho de campo se puede utilizar en un **scanf** para leer un número específico de caracteres a partir de flujo de entrada.

Terminología

bandera #
especificador de conversión %
* en ancho de campo
* en precisión
bandera + (signo de más)
bandera - (signo de menos)
bandera 0 (cero)
<stdio.h>
secuencia de escape \"
secuencia de escape \'
secuencia de escape \?
secuencia de escape \\
secuencia de escape \a
secuencia de escape \b
secuencia de escape \f
secuencia de escape \n
secuencia de escape \r
secuencia de escape \t
secuencia de escape \v
alineación
carácter de supresión de asignación (*)
inserción de espacio en blanco
especificador de conversión c
acento circunflejo (^)
especificación de conversión
especificadores de conversión
especificador de conversión d
especificador de conversión e o E
secuencia de escape
formato exponencial de punto flotante
especificador de conversión f
ancho de campo
bandera
punto flotante
cadena de control de formato
especificador de conversión g o G
especificador de conversión h
formato hexadecimal
especificador de conversión i
especificadores de conversión enteros
conjunto de rastreo invertido
especificador de conversión L
especificador de conversión l
justificación a la izquierda
caracteres literales
entero long
especificador de conversión n
especificador de conversión o
formato octal
especificador de conversión p
precisión
printf
inserción de carácter de impresión
redireccionamiento de un flujo
justificación a la derecha
redondeo
especificador de conversión s
conjunto de rastreo

scanf
notación científica
entero **short**
formato entero signado
bandera de espacio
flujo estándar de error
flujo de entrada estándar
flujo de salida estándar
flujo
especificador de conversión **u**
formato de entero no signado
espacio en blanco
especificador de conversión **x** (o **X**)

Errores comunes de programación

9.1 Olvidar encerrar entre comillas una cadena de control de formato.

9.2 Imprimir un valor negativo utilizando un especificador de conversión que espera un valor no signado.

9.3 Utilizar **%c** para imprimir el primer carácter de una cadena. La especificación de conversión **%c** espera un argumento **char**. Una cadena es un apuntador a **char**, es decir, a **char ***.

9.4 Utilizar **%s** para imprimir un argumento **char**. La especificación de conversión **%s** espera un argumento de tipo apuntador a **char**. En algunos sistemas, esto causará un error fatal en tiempo de ejecución, conocido como violación de acceso.

9.5 Es un error de sintaxis usar comillas sencillas alrededor de cadenas de caracteres. Las cadenas de caracteres deben estar encerradas entre comillas dobles.

9.6 Usar comillas dobles alrededor de una constante de carácter. Esto, de hecho, crea una cadena formada por dos caracteres, el segundo de los cuales es el carácter de terminación **NULL**. Una constante de carácter es un carácter, encerrado entre comillas sencillas.

9.7 Intentar imprimir un carácter literal de por ciento, utilizando en la cadena de control de formato % en vez de %%. Cuando en la cadena de control de formato aparece el signo de %, debe ser seguida por un especificador de conversión.

9.8 No proporcionar un ancho de campo lo suficiente extenso para manejar un valor a imprimirse. Esto puede desplazar otros datos imprimiéndose y puede producir salidas confusas. ¡Familiarícese con sus datos!

9.9 Intentar imprimir como datos literales en un enunciado **printf** una comilla, dobles comillas, signo de interrogación o un carácter de diagonal invertida, sin anteceder dichos caracteres por una diagonal invertida para formar una secuencia de escape correcta.

Prácticas sanas de programación

9.1 Edite las salidas de manera nítida para su presentación. Esto hace más legible las salidas del programa y reduce errores de usuario.

9.2 Al extraer datos, asegúrese que el usuario está consciente de situaciones en las cuales, debido al formato, los datos pudieran resultar imprecisos (por ejemplo, errores de redondeo debidos a precisiones especificadas).

9.3 Al introducir datos, solicite al usuario un elemento o pocos elementos de datos a la vez. Evite solicitar al usuario la introducción de muchos elementos de datos, en respuesta a una sola solicitud.

Sugerencias de portabilidad

9.1 El especificador de conversión **p** despliega una dirección de apuntador en forma de puesta en marcha definida (en muchos sistemas se utiliza notación hexadecimal en preferencia a notación decimal).

Ejercicios de autoevaluación

9.1 Llene los espacios de cada uno de los siguientes:

a) Todas las entradas y salidas se manejan en forma de _____.
b) El flujo de _____ se conecta por lo regular al teclado
c) El flujo de _____ se conecta por lo regular a la pantalla de la computadora.
d) Un formato preciso de salida se consigue con la función _____.
e) La cadena de control de formato puede contener _____, _____, _____, _____, y _____.
f) El especificador de conversión _____, o bien _____ puede ser utilizado para la salida de un entero decimal signado.
g) Los especificadores de conversión _____, _____, y _____ se utilizan para desplegar enteros no signados en forma octal, decimal y hexadecimal, respectivamente.
h) Se colocan los modificadores _____, y _____ antes de los especificadores de conversión enteros, a fin de indicar que se despliegan valores enteros **short**, o bien **long**.
i) El especificador de conversión _____ se utiliza para desplegar un valor en punto flotante en notación exponencial.
j) El modificador _____ se coloca antes de cualquier especificador de conversión de punto flotante para indicar que se va a delegar un valor **long double**.
k) Si no se ha especificado precisión, los especificadores de conversión **e**, **E** y **f** se despliegan con _____ dígitos de precisión a la derecha del punto decimal.
l) Para imprimir cadenas y caracteres respectivamente se utilizan los especificadores de conversión _____, y _____.
m) Todas las cadenas terminan con el carácter _____.
n) El ancho de campo y la precisión en una especificación de conversión **printf** pueden ser controladas con expresiones enteras, substituyendo un _____ en lugar del ancho de campo o de la precisión, y colocando la expresión entera en el argumento correspondiente de la lista de argumento.
o) La bandera _____ hace que en un campo la salida quede justificada a la izquierda.
p) La bandera _____ hace que se desplieguen valores, ya sea con un signo de más o con un signo de menos.
q) Se consigue un formato de entrada preciso con la función _____.
r) Se utiliza un _____ para rastrear una cadena buscando caracteres específicos y almacenar dichos caracteres en un arreglo.
s) El especificador de conversión _____ puede ser utilizado para introducir enteros octales, decimales y hexadecimales opcionalmente signados.
t) El especificador de conversión _____ puede ser utilizado para introducir un valor **double**.
u) El _____ se utiliza para leer datos del flujo de entrada y descartarlos sin asignarlos a una variable.
v) Se puede utilizar un _____ en una especificación de conversión **scanf** para indicar que un número específico de caracteres o de dígitos debe ser leído a partir del flujo de entrada.

9.2 Encuentre el error en cada uno de los siguientes y explique cómo puede ser corregido.

a) El siguiente enunciado deberá imprimir el carácter **'c'**

```
printf(" %s\n", 'c');
```

b) El siguiente enunciado debería imprimir **9.375%**

```
printf("%.3f%", 9.375);
```

c) El siguiente enunciado debería imprimir el primer carácter de la cadena **"Monday"**

```
printf("%c\n", "Monday");
```

d) `printf(""A string in quotes"");`
e) `printf(%d%e, 12, 20);`
f) `printf("%c", "x");`
g) `printf("%s\n", 'Richard');`

9.3 Escriba un enunciado para cada uno de los siguientes:

a) Imprima `1234`, justificados a la derecha, en un campo de `10` dígitos.
b) Imprima `123.456789`, en notación exponencial, con un signo de (+ o de -) y `3` dígitos de precisión.
c) Lea un valor `double` a la variable `number`.
d) Imprima `100` en forma octal, precedida por `0`.
e) Lea una cadena del arreglo de caracteres `string`.
f) Lea caracteres al arreglo `n`, hasta que se encuentre un carácter no dígito.
g) Utilice las variables enteras `x` e `y`, para especificar un ancho de campo y una precisión utilizada para desplegar el valor `double 87.4573`.
h) Lea un valor de la forma `3.5%`. Almacene el porcentaje en la variable `float percent`, y elimine el signo de % del flujo de entrada. No utilice el carácter de supresión de asignación.
i) Imprima `3.333333` como un valor `long double` con un signo ,(+ o -) en un campo de `20` caracteres con una precisión de `3`.

Respuestas a los ejercicios de autoevaluación

9.1 a) Flujos. b) Entrada estándar. c) Salida estándar. d) `printf`. e) Especificadores de conversión, bandera, anchos de campo, precisiones y caracteres literales. f) `d`, `i`. g) `o`, `u`, `x` (o `X`). h) `h`, `l`. i) `e` (o `E`). j) `L`. k) 6. l) `s`, `c`. m) `NULL('\0')`. n) asterisco (`*`). o) - (menos). p) + (más). q) `scanf`. r) Conjunto de rastreo. s) `i`. t) `1e`, `1E`, `1f`, `1g`, o bien `1G`. u) Carácter de supresión de asignación (`*`). v) Ancho de campo.

9.2

a) Error: el especificador de conversión `s` espera un argumento del tipo apuntador a `char`.
Corrección: para imprimir el carácter `'c'`, utilice la especificación de conversión %c, o bien cambie `'c'` a `"c"`.
b) Intentar imprimir el carácter literal % sin utilizar la especificador de conversión %%.
Corrección: Utilice %% para imprimir un carácter literal %.
c) Error: el especificador de conversión `c` espera un argumento del tipo `char`.
Corrección: Para imprimir el primer carácter de `"Monday"` utilice el especificador de conversión `%ls`.
d) Error: tratar de imprimir el carácter literal ", sin utilizar la secuencia de escape \".
Corrección: Remplace cada comilla en el conjunto interior de comillas, con \".
e) Error: la cadena de control de formato no está encerrada entre comillas dobles.
Corrección: Encierre `%d%d` entre comillas dobles.
f) El carácter `x` está encerrado entre comillas dobles.
Corrección: los constantes de caracteres a ser impresos utilizando %c deben estar encerrados entre comillas sencillas.
g) Error: la cadena a imprimirse está encerrada entre comillas sencillas.
Corrección: utilice comillas dobles en vez de sencillas para representar a una cadena.

9.3

a) `printf("%10d\n", 1234);`
b) `printf("%+.3e\n", 123.456789);`
c) `scanf("%lf", &number);`
d) `printf("%#o\n", 100);`
e) `scanf("%s", string);`
f) `scanf("%[^0123456789]", n);`
g) `printf("%*.*f\n", x, y, 87.4573);`
h) `scanf("%f%%", &percent);`
i) `printf("%+20.3Lf\n", 3.333333);`

Ejercicios

9.4 Escriba un enunciado `printf` o bien `scanf` para cada uno de los siguientes:

a) Imprima el entero no signado `40000` justificado a la izquierda, en un campo de `15` dígitos, con `8` dígitos.
b) Lea un valor hexadecimal a la variable `hex`.
c) Imprima `200` con y sin un signo.
d) Imprima `100` en forma hexadecimal precedido por `0x`.
e) Lea caracteres al arreglo `s`, hasta que se encuentre con la letra `p`.
f) Imprima `1.234` en un campo de `9` dígitos, con ceros a la izquierda.
g) Lea una hora de la forma `hh:mm:ss` almacenando las partes de la hora en las variables enteras `hour`, `minute` y `second`. Omita los dobles puntos (:) del flujo de entrada. Utilice el carácter de supresión de asignación.
h) Lea una cadena de la forma `"characters"` de la entrada estándar. Almacene la cadena en el arreglo de carácter `s`. Elimine las comillas del flujo de entrada.
i) Lea una hora de la forma `hh:mm:ss` almacenando las partes de la hora en las variables enteras `hour`, `minute` y `second`. Omita los dobles puntos (:) en el flujo de entrada. No utilice el carácter de supresión de asignación.

9.5 Muestre lo que se imprime mediante cada uno de los enunciados siguientes. Si un enunciado es incorrecto, indique porqué.

a) `printf("%-10d\n", 10000);`
b) `printf("%c\n", "This is a string");`
c) `printf("%*.*lf\n", 8, 3, 1024.987654);`
d) `printf("%#o\n%#X\n%#e\n", 17, 17, 1008.83689);`
e) `printf("% ld\n%+ld\n", 1000000, 1000000);`
f) `printf("%10.2E\n", 444.93738);`
g) `printf("%10.2g\n", 444.93738);`
h) `printf("%d\n", 10.987);`

9.6 Encuentre el o los errores en cada uno de los siguientes segmentos de programa. Explique cómo pueden ser corregidos cada uno de ellos.

a) `printf("%s\n", 'Happy Birthday');`
b) `printf("%c\n", 'Hello');`
c) `printf("%c\n", "This is a string");`
d) El siguiente enunciado deberá imprimir `"Bon Voyage"`
```
printf(""%s"", "Bon Voyage");
```
e)
```
char day [] = "Sunday";
printf("%s\n", day [3]);
```
f) `printf('Enter your name: ');`
g) `printf(%f, 123.456);`
h) El siguiente enunciado debería de imprimir los caracteres `'O' y 'K'`.
```
printf("%s%s\n", 'O', 'K');
```
i)
```
char s[10];
scanf("%c", s[7]);
```

9.7 Escriba un programa que cargue el arreglo de 10 elementos `number` con enteros al azar, desde 1 hasta 1000. Para cada uno de los valores, imprima el valor y el total acumulado del número de caracteres impresos. Utilice la especificación de conversión %n para determinar el número de caracteres ya extraídos para cada valor. Imprima el número total de caracteres extraídos para todos los valores incluyendo el valor actual, cada vez que éste sea impreso. La salida deberá tener el formato siguiente:

```
Value       Total characters
342         3
1000        7
963         10
6           11
etc.
```

9.8 Escriba un programa para probar la diferencia entre los especificadores de conversión `%d` y `%i` al ser utilizados en enunciados `scanf`. Utilice los enunciados

```
scanf("%i%d", &x, &y);
printf("%d%d\n", x, y);
```

para introducir e imprimir los valores. Pruebe el programa con los siguientes conjuntos de datos de entrada:

```
 10    10
-10   -10
010   010
0x10  0x10
```

9.9 Escriba un programa que imprima los valores de apuntador utilizando todos los especificadores de conversión enteros y la especificación de conversión `%p`. ¿Cuáles son los que imprimen valores raros? ¿Cuáles son los que causan errores? ¿En cuál de los formatos la especificación de conversión `%p` despliega en su sistema la dirección?

9.10 Escriba un programa para probar el resultado de imprimir el valor entero `12345` y el de punto flotante `1.2345` en varios tamaños de campo. ¿Qué pasa cuando se imprimen los valores en campos que contienen menos dígitos que los valores mismos?

9.11 Escriba un programa que imprima el valor `100.453627` redondeado al dígito, décima, centésima, milésima y decenas de millar más cercano.

9.12 Escriba un programa que desde el teclado introduzca una cadena y determine la longitud de la misma. Imprima la cadena utilizando como ancho de campo dos veces su longitud.

9.13 Escriba un programa que convierta temperaturas Fahrenheit enteras desde `0` hasta `212` grados a temperaturas Celsius de punto flotante con `3` dígitos de precisión. Utilice la fórmula

```
celsius = 5.0 / 9.0 * (fahrenheit - 32);
```

para llevar a cabo el cálculo. La salida deberá ser impresa en dos columnas justificadas a la derecha, cada una de 10 caracteres, y las temperaturas Celsius deberán ser antecedidas por un signo, tanto para valores positivos como negativos.

9.14 Escriba un programa para probar todas las secuencias de escape de la figura 9.16. Para aquellas secuencias de escape que mueven el cursor, imprima un carácter, antes y después de imprimir la secuencia de escape, a fin de que resulte claro dónde se ha movido el cursor.

9.15 Escriba un programa que determine si `?` puede ser impreso como un carácter literal como parte de una cadena de control de formato `printf`, en vez de utilizar la secuencia de escape `\?`

9.16 Escriba un programa que introduzca el valor `437` utilizando cada uno de los especificadores de conversión enteros `scanf`. Imprima cada valor introducido, utilizando todos los especificadores de conversión enteros.

9.17 Escriba un programa que utilice cada uno de los especificadores de conversión `e`, `f` y `g` para introducir el valor `1.2345`. Imprima los valores de cada variable para probar que cada especificador de conversión puede ser utilizado para introducir este mismo valor.

9.18 En algunos lenguajes de programación, las cadenas pueden ser introducidas entre comillas sencillas o dobles. Escriba un programa que lea las tres cadenas `suzy`, `"suzy"`, y `'suzy'`. ¿Son las comillas sencillas y dobles ignoradas o leídas por C, como parte de la cadena?

9.19 Escriba un programa que determine si `?` puede ser impreso como la constante de carácter `'?'`, en vez de la secuencia de escape de carácter constante `'\?'` mediante el uso de especificador de conversión `%c` en la cadena de control de formato de un enunciado `printf`.

9.20 Escriba un programa que utilice el especificador de conversión `g` para extraer el valor `9876.12345`. Imprima el valor con precisiones que vayan desde `1` hasta `9`.

10

Estructuras, uniones, manipulaciones de bits y enumeraciones

Objetivos

- Ser capaz de crear y utilizar estructuras, uniones y enumeraciones.
- Ser capaz de pasar estructuras a funciones en llamada por valor y en llamada por referencia.
- Ser capaz de manipular datos con operadores a nivel de bits.
- Ser capaz de crear campos de bits para almacenar datos en forma compacta.

Nunca pude comprender lo que esos malditos puntos significaban.
Winston Churchill

Pero otra vez una unión en desunión;
William Shakespeare

Puedes excluirme.
Samuel Goldwyn

La misma caritativa y vieja mentira
Repetida conforme los años pasan
Siempre con el mismo éxito—
"¡Realmente no has cambiado nada!"
Margaret Fishback

Sinopsis

10.1 introducción

Las *estructuras* son colecciones de variables relacionadas —a veces denominadas *agregados*— bajo un nombre. Las estructuras pueden contener variables de muchos tipos diferentes de datos —a diferencia de los arreglos, que contienen únicamente elementos de un mismo tipo de datos. Generalmente las estructuras se utilizan para definir registros a almacenarse en archivos (vea el capítulo 11, "Procesamiento de archivos"). Los apuntadores y las estructuras facilitan la formación de estructuras de datos de mayor complejidad, como son listas enlazadas, colas de espera, pilas y árboles (vea el capítulo 12, "Estructuras de datos").

10.2 Definiciones de estructuras

Las estructuras son *tipos de datos derivados* —están construidas utilizando objetos de otros tipos. Considere la siguiente definición de estructura:

```
struct card {
   char *face;
   char *suit;
};
```

La palabra reservada **struct** presenta la definición de estructura. El identificador **card** es el *rótulo de la estructura*. El rótulo de la estructura da nombre a la definición de la misma, y se utiliza con la palabra reservada **struct** para declarar variables del *tipo estructura*. En este ejemplo, el tipo estructura es **struct card** . Las variables declaradas dentro de las llaves de la definición de estructura son los *miembros* de la estructura. Los miembros de la misma estructura

deben tener nombres únicos, pero dos estructuras diferentes pueden contener miembros con el mismo nombre sin entrar en conflicto (pronto veremos por qué). Cada definición de estructura debe terminar con un punto y coma.

Error común de programación 10.1

Olvidar el punto y coma que da por terminada una definición de estructura.

La definición de **struct card** contiene dos miembros del tipo **char * face** y **suit**. Los miembros de la estructura pueden ser variables de los tipos de datos básicos (es decir, **int**, **float**, etcétera), o agregados, como son los arreglos y otras estructuras. Como ya vimos en el capítulo 6, cada elemento de un arreglo debe ser del mismo tipo. Los miembros de una estructura, sin embargo, pueden ser de una variedad de tipos de datos. Por ejemplo, un **struct employee** pudiera contener miembros de cadenas de caracteres correspondientes a los nombres y apellidos, un miembro **int**, para la edad del empleado, un miembro **char**, que contenga **'M'** o bien **'F'** para el sexo del empleado, un miembro **float** para el salario horario del empleado, y así sucesivamente. Una estructura no puede contener una instancia de sí misma. Por ejemplo, una variable del tipo **struct card** no puede ser declarada dentro de la definición correspondiente a **struct card**. Sin embargo, pudiera ser incluido un apuntador a **estruct card**. Una estructura, que contenga un miembro que es un apuntador al mismo tipo de estructura, se conoce como una *estructura autorreferenciada*. Las estructuras autorreferenciadas se utilizan en el capítulo 12 para construir varios tipos de estructuras de datos enlazadas.

La anterior definición de estructura no reserva ningún espacio en memoria, más bien genera un nuevo tipo de datos, que se utiliza para declarar variables. Las variables de estructura se declaran como se declaran las variables de otros tipos. La declaración

```
struct card a, deck[52], *c;
```

declara **a** ser una variable del tipo **struct card**, declara **deck** como un arreglo con 52 elementos del tipo **struct card**, y declara **c** como un apuntador a **struct card**. Las variables de un tipo dado de estructura, pudieran también ser declaradas colocando una lista, separada por comas, de los nombres de las variables, entre la llave de cierre de la definición de la estructura y el punto y coma que termina la definición de la misma. Por ejemplo, la declaración anterior podía haberse incorporado en la definición de estructura **struct card** como sigue:

```
struct card {
      char *face;
      char *suit;
} a, deck[52], *c;
```

El nombre del rótulo de la estructura es opcional. Si la definición de una estructura no contiene un nombre de rótulo de estructura, las variables de ese tipo de estructura pueden únicamente ser declaradas dentro de la definición de estructura —y no en una declaración por separado.

Práctica sana de programación 10.1

Al crear un tipo de estructura proporcione un nombre de rótulo de estructura. El nombre de rótulo de estructura es conveniente más adelante en el programa para la declaración de nuevas variables de este tipo de estructura.

Práctica sana de programación 10.2

Seleccionar un nombre de rótulo de estructura significativo ayuda a autodocumentar el programa.

Las únicas operaciones válidas que pueden ejecutarse sobre estructuras son: asignar variables de estructura a variables de estructura del mismo tipo, tomando la dirección (**&**) de una variable de estructura, obteniendo acceso a los miembros de una variable de estructura (vea la Sección 10.4), y utilizando el operador **sizeof**, a fin de determinar el tamaño de la variable de estructura.

Error común de programación 10.2

Asignar una estructura de un tipo a una estructura de un tipo distinto.

Las estructuras no pueden compararse entre sí, porque los miembros de las estructuras no están necesariamente almacenados en bytes de memoria consecutivos. Algunas veces en una estructura existen "huecos" porque las computadoras pudieran almacenar tipos de datos específicos en ciertos límites de memoria, como son límites de media palabra, de palabra o de dobles palabras. Una palabra es una unidad estándar de memoria, utilizada para almacenar datos en una computadora — 2 o 4 bytes, normalmente. Considere la siguiente definición de estructura, en la cual se declaran **sample1** y **sample2**, del tipo **struct example**:

```
struct example {
      char c;
      int i;
} sample1, sample2;
```

Una computadora con palabras de 2 bytes pudiera requerir que cada uno de los miembros de **struct example** fuesen alineados en un límite de palabras, es decir, al principio de una palabra (esto depende de la máquina). En la Figura 10.1 se muestra una alineación de almacenamiento para una variable del tipo **struct example**, que ha sido asignado al carácter **'a'**, y el entero **97** (se muestran las representaciones de los valores en bits). Si los miembros se almacenan empezando en los límites de palabras, aparece un hueco de 1 byte (byte **1** en la figura) en el almacenamiento para variables del tipo **struct example**. El valor en el hueco de un byte se queda sin definir. Si los valores de miembros de **sample1** y **sample2** son de hecho iguales, la comparación de las estructuras no será necesariamente igual, porque los huecos no definidos de un byte probablemente no contendrán valores idénticos.

Error común de programación 10.3

Es un error de sintaxis comparar estructuras, debido a diferentes requisitos de alineación en los diferentes sistemas.

Sugerencia de portabilidad 10.1

Dado que depende de la máquina el tamaño de los elementos de datos de un tipo particular, y debido a que las consideraciones de alineación de almacenamiento también son dependientes de la máquina, entonces también lo será la representación de una estructura.

Fig. 10.1 Una posible alineación de almacenamiento para una variable del tipo **struct example** mostrando en la memoria un área no definida.

10.3 Cómo inicializar estructuras

Las estructuras pueden ser inicializadas mediante listas de inicialización como con los arreglos. Para inicializar una estructura, escriba en la declaración de la estructura, a continuación del nombre de la variable, un signo igual, con inicializadores encerrados entre llaves y separados por comas. Por ejemplo, la declaración

```
struct card a = ("Three", "Hearts");
```

crea la variable **a** del tipo **struct card** (como fue definida anteriormente) e incializa el miembro **face** a **"Three"**, y el miembro **suit** a **"Hearts"**. Si en la lista aparecen menos inicializadores que en la estructura, los miembros restantes automáticamente quedarán inicializados a **0** (o **NULL** si el miembro es un apuntador). Las variables de estructura declarados por fuera de una definición de función (es decir, en forma externa) se inicializan a **0** o **NULL** si en la declaración externa no se inicializan en forma explícita. Las variables de estructura también pueden ser inicializadas en enunciados de asignación, asignándoles una variable de estructura del mismo tipo, o asignando valores a los miembros individuales de la misma.

10.4 Cómo tener acceso a los miembros de estructuras

Para tener acceso a miembros de estructuras se utilizan dos operadores: el *operador de miembro de estructura* (.) —también conocido como *operador punto*— y el *operador de apuntador de estructura* (->) —también conocido como el *operador de flecha*. El operador de miembro de estructura tiene acceso a un miembro de estructura mediante el nombre de la variable de estructura. Por ejemplo, para imprimir el miembro **suit** de la estructura **a** correspondiente a la declaración anterior, utilice el enunciado

```
printf ("%s", a.suit);
```

El operador de apuntador de estructura —que consiste de un signo menos (-) y de un signo mayor que (>), sin espacios intermedios— tiene acceso a un miembro de estructura vía un apuntador a la estructura. Suponga que el apuntador **aPtr** se ha declarado para apuntar a **struct card**, y que la dirección de la estructura **a** ha sido asignada a **aPtr**. Para imprimir el miembro **suit** de la estructura **a** utilizando el apuntador **aPtr**, utilice el enunciado

```
printf ("%s", aPtr->suit);
```

La expresión **aPtr->suit** es equivalente **(*aPtr).suit** que desreferencía el apuntador y tiene acceso al miembro **suit** utilizando el operador de miembro de estructura. Se requiere aquí de los paréntesis, porque el operador de miembro de estructura (.) tiene una precedencia mayor que el operador de desreferenciación de apuntador (*). El operador de apuntador de estructura y el operador de miembro de estructura, junto con los paréntesis y los corchetes ([]) utilizados para los subíndices de arreglos, tienen la precedencia de operadores más alta y se asocian de izquierda a derecha.

Práctica sana de programación 10.3

Evite utilizar los mismos nombres para miembros de estructura de distintos tipos. Ello es permitido, pero podría causar confusión.

Práctica sana de programación 10.4

No deje espacios alrededor de los operadores -> y . ya que ayuda a enfatizar que las expresiones en las cuales los operadores están contenidos son esencialmente nombres individuales de variables.

Error común de programación 10.4

Insertar un espacio entre el signo de - y el signo de > del operador de apuntador de estructura, (o insertar espacios entre los componentes de cualquier otro operador múltiple de teclado, a excepción de ?:).

Error común de programación 10.5

Intentar referirse a un miembro de una estructura utilizando únicamente el nombre de dicho miembro.

Error común de programación 10.6

No utilizar paréntesis al referirse a un miembro de estructura utilizando un apuntador y el operador de miembro de estructura (por ejemplo `*aPtr.suit`, es un error de sintaxis).

El programa de la figura 10.2 pone de manifiesto el uso de los operadores de miembro de estructura y de apuntador de estructura. Mediante el uso del operador de miembro de estructura, los miembros de la estructura **a** son asignados los valores **"Ace"** y **"Spades"** respectivamente. Al apuntador **aPtr** se le asigna la dirección de la estructura **a**. Un enunciado **printf** imprime los miembros de la variable de estructura **a**, utilizando el operador de miembro de estructura con el nombre de variable **a**, el operador de apuntador de estructura con el apuntador **aPtr**, y el operador de miembro de estructura con el apuntador desreferenciado **aPtr**.

```
/* Using the structure member and
   structure pointer operators */
#include <stdio.h>

struct card {
   char *face;
   char *suit;
};

main()
{
   struct card a;
   struct card *aPtr;

   a.face = "Ace";
   a.suit = "Spades";
   aPtr = &a;
   printf("%s%s%s\n%s%s%s\n%s%s%s\n",
          a.face, " of ", a.suit,
          aPtr->face, " of ", aPtr->suit,
          (*aPtr).face, " of ", (*aPtr).suit);
   return 0;
}
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

Fig. 10.2 Cómo utilizar el operador de miembro de estructura y el operador de apuntador de estructura.

10.5 Cómo utilizar estructuras con funciones

Las estructuras pueden ser pasadas a funciones pasando miembros de estructura individuales, pasando toda la estructura, o pasando un apuntador a una estructura. Cuando se pasan estructuras o miembros individuales de estructura a una función, se pasan en llamada por valor. Por lo tanto, los miembros de la estructura de un llamador no podrán ser modificadas por la función llamada.

Para pasar una estructura en llamada por referencia, pase la dirección de la variable de estructura. Los arreglos de estructura —como todos los demás otros arreglos— son automáticamente pasados en llamada por referencia.

En el capítulo 6, indicamos que un arreglo podía ser pasado en llamada por valor mediante el uso de una estructura. Para pasar un arreglo en llamada por valor, origine una estructurza con el arreglo como un miembro. Dado que las estructuras se pasan en llamada por valor, el arreglo será pasado en llamada por valor.

Error común de programación 10.7

Suponer que las estructuras, como los arreglos, se pasan automáticamente en llamada por referencia, e intentar modificar los valores de estructura del llamador en la función llamada.

Sugerencia de rendimiento 10.1

Es más eficaz pasar estructuras en llamada por referencia que pasar estructuras en llamada por valor (ya que esto último requiere que toda la estructura se copie).

10.6 Typedef

La palabra reservada **typedef** proporciona un mecanismo para la creación de sinónimos (o alias) para tipos de datos anteriormente definidos. Los nombres de los tipos de estructura se definen a menudo utilizando **typedef**, a fin de crear nombres de tipo más breves. Por ejemplo, el enunciado

```
typedef struct card Card;
```

define el nuevo nombre de tipo **Card** como un sinónimo para el tipo **struct card**. Los programadores en C utilizan a menudo **typedef** para definir un tipo de estructura de tal forma que un rótulo de estructura no sea requerido. Por ejemplo, la definición siguiente

```
typedef struct {
   char *face;
   char *suit;
} Card;
```

crea el tipo de estructura **Card**, sin necesidad de un enunciado por separado **typedef**.

Práctica sana de programación 10.5

*Ponga los nombres **typedef** en mayúsculas, para enfatizar que esos nombres son sinónimos de otros nombres de tipo.*

Card puede ahora ser utilizado para declarar variables del tipo **struct card**. La declaración

```
Card deck[52];
```

declara un arreglo de 52 estructuras **Card** (es decir, variables del tipo **struct card**). Al crear un nuevo nombre utilizando **typedef** no se crea un nuevo tipo; **typedef** simplemente crea un

nuevo nombre de tipo, que puede ser utilizado como un seudónimo para un nombre de tipo existente. Un nombre significativo auxilia a autodocumentar el programa. Por ejemplo, cuando leemos la declaración anterior, sabemos que "**deck** es un arreglo de 52 **Cards**"

typedef se utiliza a menudo para crear seudónimos para los tipos de datos básicos. Por ejemplo, un programa que requiera de enteros de 4 bytes, pudiera utilizar el tipo **int** en un sistema y el tipo **long** en otro. Los programas diseñados para portabilidad, a menudo utilizan **typedef** para crear un alias o seudónimo para los enteros de 4 bytes como sería **Integer**. Una vez dentro del programa el alias **Integer** puede ser modificado, para hacer que el programa funcione en ambos sistemas.

Sugerencia de portabilidad 10.2

*Utilice **typedef** para ayudar a hacer más portátil un programa.*

10.7 Ejemplo: simulación de barajar y distribuir cartas de alto rendimiento

El programa en la figura 10.3 se basa en la simulación de barajar y distribuir cartas analizado en el capítulo 7. El programa representa el mazo de cartas o de naipes como un arreglo de estructuras. El programa utiliza algoritmos de alto rendimiento para barajar y distribuir. La salida del programa de alto rendimiento para barajar y distribuir se muestra en la figura 10.4.

En el programa, la función **fillDeck** inicializa el arreglo **Card** en orden desde Ace hasta King de cada uno de los palos. El arreglo **Card** se pasa a la función **shuffle**, donde se pone en operación el algoritmo de alto rendimiento de barajar. La función **shuffle** toma como argumento un arreglo de 52 estructuras **Card**. La función cicla a través de las 52 cartas (subíndices de arreglo 0 a 51) mediante una estructura **for**. Para cada una de las cartas, es tomado al azar un número entre 0 y 51. A continuación, en el arreglo son intercambiadas la estructura actual **Card** y la estructura seleccionada al azar **Card**. En una sola pasada de todo el arreglo se llevan a cabo un total de 52 intercambios, y ¡el arreglo de estructuras **Card** queda barajado! Este algoritmo no puede sufrir por posposición indefinida, como sufría el algoritmo de barajar presentado en el capítulo 7. Dado que en el arreglo las estructuras **Card** fueron intercambiadas en su lugar, el algoritmo de distribución de alto rendimiento puesto en marcha en la función **deal** requerirá de únicamente una pasada del arreglo para distribuir las cartas barajadas.

Error común de programación 10.8

Olvidar incluir el subíndice de arreglo al referirse a estructuras individuales de un arreglo de estructuras.

10.8 Uniones

Una *unión* es un tipo de datos derivado —como lo es una estructura— cuyos miembros comparten el mismo espacio de almacenamiento. Para distintas situaciones en un programa, algunas variables pudieran no ser de importancia, pero otras variables lo son —por lo que una unión comparte el espacio, en vez de desperdiciar almacenamiento en variables que no están siendo utilizadas. Los miembros de una unión pueden ser de cualquier tipo. El número de bytes utilizados para almacenar una unión, deben ser por lo menos suficientes para contener el miembro más grande. En la mayor parte de los casos, las uniones contienen dos o más tipos de datos. Unicamente un miembro y, por lo tanto, únicamente un tipo de datos, puede ser referenciado en un momento dado. Es responsabilidad del programador asegurarse que en una unión los datos están referenciados con el tipo de dato apropiado.


```
/* The card shuffling and dealing program using structures */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct card {
   char *face;
   char *suit;
};

typedef struct card Card;

void fillDeck(Card *, char *[], char *[]);
void shuffle(Card *);
void deal(Card *);

main()
{
   Card deck[52];
   char *face[] = {"Ace", "Deuce", "Three", "Four", "Five",
                   "Six", "Seven", "Eight", "Nine", "Ten",
                   "Jack", "Queen", "King"};
   char *suit[] = {"Hearts", "Diamonds", "Clubs", "Spades"};

   srand(time(NULL));

   fillDeck(deck, face, suit);
   shuffle(deck);
   deal(deck);
   return 0;
}

void fillDeck(Card *wDeck, char *wFace[], char *wSuit[])
{
   int i;

   for (i = 0; i <= 51; i++) {
      wDeck[i].face = wFace[i % 13];
      wDeck[i].suit = wSuit[i / 13];
   }
}

void shuffle(Card *wDeck)
{
   int i, j;
   Card temp;

   for (i = 0; i <= 51; i++) {
      j = rand() % 52;
      temp = wDeck[i];
      wDeck[i] = wDeck[j];
      wDeck[j] = temp;
   }
}
```

```
void deal(Card *wDeck)
{
   int i;

   for (i = 0; i < 52; i++)
      printf("%5s of %-8s%c", wDeck[i].face, wDeck[i].suit,
             (i + 1) % 2 ? '\t' : '\n');
}
```

Fig. 10.3 Simulación de barajar y distribuir cartas de alto rendimiento (parte 2 de 2).

```
 Eight of Diamonds        Ace of Hearts
 Eight of Clubs          Five of Spades
 Seven of Hearts        Deuce of Diamonds
   Ace of Clubs           Ten of Diamonds
Deuce of Spades           Six of Diamonds
Seven of Spades         Deuce of Clubs
Jacks of Clubs            Ten of Spades
 King of Hearts          Jack of Diamonds
Three of Hearts         Three of Diamonds
Three of Clubs           Nine of Clubs
  Ten of Hearts         Deuce of Hearts
  Ten of Clubs          Seven of Diamonds
  Six of Clubs          Queen of Spades
  Six of Hearts         Three of Spades
 Nine of Diamonds         Ace of Diamonds
 Jack of Spades          Five of Clubs
 King of Diamonds       Seven of Clubs
 Nine of Spades          Four of Hearts
  Six of Spades         Eight of Spades
Queen of Diamonds        Five of Diamonds
  Ace of Spades          Nine of Hearts
 King of Clubs           Five of Hearts
 King of Spades          Four of Diamonds
Queen of Hearts         Eight of Hearts
 Four of Spades          Jack of Hearts
 Four of Clubs          Queen of Clubs
```

Fig. 10.4 Salida de la simulación de barajar y distribuir cartas de alto rendimiento.

Error común de programación 10.9

Es un error lógico referenciar con el tipo equivocado, datos en una unión almacenados con un tipo distinto.

Sugerencia de portabilidad 10.3

Si en una unión los datos se almacenan como de un tipo y se referencían como de otro tipo, los resultados serán dependientes de la instalación.

Una unión se declara con la palabra reservada **union** en el mismo formato que una estructura. La declaración **union**

```
union number{
   int x;
   float y;
};
```

indica que **number** es un tipo **union** con miembros **int x** y **float y**. En un programa normalmente la definición de unión antecede a **main**, por lo que ésta puede ser utilizada para declarar variables en todas las funciones del programa.

Observación de ingeniería de software 10.1

Al igual que en una declaración `struct`*, una declaración* `union` *simplemente crea un tipo nuevo. Colocar una declaración* `union` *o* `struct` *fuera de cualquier función no crea una variable global.*

Las operaciones que pueden ser ejecutadas en una unión son: asignar una unión a otra unión del mismo tipo, tomar la dirección (**&**) de una unión, y tener acceso a los miembros de una unión utilizando el operador de miembro de estructura y el operador de apuntador de estructura. Las uniones no pueden ser comparadas entre sí, por las mismas razones que no pueden compararse las estructuras.

En una declaración, una unión puede ser inicializada únicamente con un valor del mismo tipo que el primer miembro de la unión. Por ejemplo, en la unión anterior, la declaración

```
union number value = {10};
```

es una inicialización válida de la variable de unión **value**, porque la unión está inicializada con un **int**, pero la siguiente declaración no sería válida:

```
union number value = {1.43};
```

Error común de programación 10.10

Es un error de sintaxis comparar uniones, debido a los diferentes requisitos de alineación en varios sistemas.

Error común de programación 10.11

Inicialización de una uión en una declaración con un valor cuyo tipo es distinto al del primer miembro de la unión.

Sugerencia de portabilidad 10.4

La cantidad de almacenamiento requerido para almacenar una unión es dependiente de la instalación.

Sugerencia de portabilidad 10.5

Quizá no sea fácil portar algunas uniones a otros sistemas de computación. El que una unión sea portable o no depende a menudo de los requerimientos de alineación de almacenamiento para los tipos de miembro de unión de información en un sistema particular.

Sugerencia de rendimiento 10.2

Las uniones ahorran almacenamiento.

El programa de la figura 10.5 utiliza la variable **value** del tipo **union number**, para desplegar el valor almacenado en la unión, tanto como un **int** como como un **float**. La salida del programa depende de la instalación. La salida del programa muestra que la representación interna de un valor **float** puede resultar bastante distinta de la representación de **int**.

10.9 Operadores a nivel de bits

En las computadoras en forma interna todos los datos se representan como secuencias de bits. Cada bit puede asumir un valor de **0** o un valor de **1**. En la mayor parte de los sistemas, una secuencia de 8 bits forma un byte —la unidad estándar de almacenamiento para una variable del tipo **char**. Otros tipos de datos son almacenados en números de bits más grandes. Los operadores

```
/* An example of a union */
#include <stdio.h>

union number {
    int x;
    float y;
};

main()
{
    union number value;

    value.x = 100;
    printf("%s\n%s\n%s%d\n%s%f\n\n",
           "Put a value in the integer member",
           "and print both members.",
           "int:   ", value.x,
           "float: ", value.y);

    value.y = 100.0;
    printf("%s\n%s\n%s%d\n%s%f\n",
           "Put a value in the floating member",
           "and print both members.",
           "int:   ", value.x,
           "float: ", value.y);
   return 0;
}
```

```
Put  value in the integer member
and print both members.
int:   100
float: 0.000000

Put a value in the floating member
and print both members.
int:   17096
float: 100.000000
```

Fig. 10.5 Cómo imprimir el valor de una unión en ambos tipos de datos de miembro.

a nivel de bits se utilizan para manipular los bits de operandos integrales (**char**, **short**, **int** y **long**; tanto **signed** como **unsigned**). Los enteros no signados (**unsigned**) son utilizados normalmente con los operadores a nivel de bits.

Sugerencia de portabilidad 10.6

Las manipulaciones de datos a nivel de bits son dependientes de la máquina.

Advierta que los análisis de los operadores a nivel de bits de esta sección, muestran las representaciones binarias de los operandos enteros. Para una explicación detallada del sistema numérico binario (también conocido como de base 2) vea el Apéndice E, "Sistemas numéricos". También, los programas de las Secciones 10.9 y 10.10 fueron probados en un Macintosh de Apple, utilizando Think C y en una PC compatible, utilizando Borland C++. Ambos sistemas utilizan enteros de 16 bits (2 bytes). Dada la naturaleza de dependencia de la máquina de las manipulaciones a nivel de bits, estos programas pudieran no funcionar en su sistema.

Los operadores a nivel de bits son: *AND a nivel de bits (&), OR inclusivo a nivel de bits (|), OR exclusivo a nivel de bits (^), desplazamiento a la izquierda (<<), desplazamiento a la derecha (>>)*, y *complemento (~)*. El AND a nivel de bits, el OR inclusivo a nivel de bits y el OR exclusivo a nivel de bits son operadores que comparan sus dos operandos bit por bit. El operador AND a nivel de bits establece en el resultado cada bit a 1, si el bit correspondiente en ambos operandos es 1. El operador OR inclusivo a nivel de bits, establece en el resultado cada bit a 1, si el bit correspondiente en cada o en (ambos) operandos es 1. El operador OR exclusivo a nivel de bits establece en el resultado cada bit a 1 si el bit correspondiente en exactamente un operando es 1. El operador de desplazamiento a la izquierda desplaza los bits de su operando izquierdo hacia la izquierda por el número de bits especificado en su operando derecho. El operador de desplazamiento a la derecha desplaza los bits de su operando izquierdo hacia la derecha en el número de bits especificado por su operando derecho. El operador de complemento a nivel de bits define en el resultado todos los bits **0** en su operando a 1, y define todos los bits **1** a **0** en el resultado. En

Operador	Descripción
& AND a nivel de bits	Los bits en el resultado se establecen a **1** si los bits correspondientes en ambos operandos son ambos **1**.
| OR inclusivo a nivel de bits	Los bits en el resultado se establecen a **1** si por lo menos uno de los bits correspondientes en los dos operandos es **1**.
^ OR exclusivo a nivel de bits	Los bits en el resultado se definen a **1** si exactamente uno de los bits correspondientes en los dos operandos es **1**.
<< desplazamiento a la izquierda	Desplaza los bits del primer operando hacia la izquierda en el número de bits especificado por el segundo operando; rellena a partir de la derecha con bits **0**.
>> desplazamiento a la derecha	Desplaza los bits del primer operando hacia la derecha en el número de bits especificado por el segundo operando; el método de rellenar a partir de la izquierda depende de la máquina.
~ complemento a uno	Todos los bits **0** se definen a **1** y todos los bits **1** se definen a cero.

Fig. 10.6 Los operadores a nivel de bits.

los ejemplos que siguen aparecen análisis detallados de cada operador a nivel de bits. Los operadores a nivel de bits se resumen en la figura 10.6.

Al utilizar los operadores a nivel de bits, es útil imprimir los valores en su representación binaria, para ilustrar los efectos precisos de estos operadores. El programa de la figura 10.7 imprime un entero **unsigned** en su representación binaria en grupos de ocho bits cada uno. La función **displayBits** utiliza el operador AND a nivel de bits para combinar la variable **value** con la variable **displayMask**. A menudo, el operador AND a nivel de bits se utiliza con un operando conocido como una *máscara* —un valor entero con bits específicos establecidos a 1. Las máscaras se utilizan para ocultar algunos bits en un valor, mientras otros bits se seleccionan. En la función **displayBits**, la variable de máscara **displayMask** es asignada el valor **1 << 15 (10000000 00000000)**. El operador de desplazamiento a la izquierda desplaza el valor

```
/* Printing an unsigned integer in bits */
#include <stdio.h>

main()
{
   unsigned x;
   void displayBits(unsigned);

   printf("Enter an unsigned integer: ");
   scanf("%u", &x);
   displayBits(x);
   return 0;
}

void displayBits(unsigned value)
{
   unsigned c, displayMask = 1 << 15;

   printf("%7u = ", value);

   for (c = 1; c <= 16; c++) {
      putchar(value & displayMask ? '1' : '0');
      value <<= 1;

      if (c % 8 == 0)
         putchar(' ');
   }

   putchar('\n');
}
```

```
Enter an unsigned integer: 65000
  65000 = 11111101 11101000
```

Fig. 10.7 Cómo imprimir un entero no signado en bits.

1 de la posición inferior (más a la derecha) hacia el bit de orden superior (más a la izquierda) en **displayMask**, y rellena con bits 0 a partir de la derecha. El enunciado

```
putchar(value & displayMask ? '1' : '0');
```

determina si deberá de imprimirse un 1 o un 0 para el bit actual más a la izquierda de la variable **value**. Suponga que la variable **value** contiene **65000 (11111101 11101000)**. Cuando se combinan **value** y **displayMask** utilizando **&**, todos los bits, a excepción del bit de orden superior, en la variable **value**, son "enmascarados" (ocultos) porque cualquier bit "manipulado por AND" con 0 da como resultado 0. Si el bit más a la izquierda es 1, **value & displayMask** se evalúa a 1, y 1 se imprime —de lo contrario se imprime 0. La variable **value** es después desplazada un bit a la izquierda, mediante la expresión **value << = 1** (este es equivalente a **value = value << = 1**). Estos pasos se repiten para cada uno de los bits en la variable **unsigned value**. En la figura 10.8 se resumen los resultados de combinar dos bits con el operador AND a nivel de bits.

Error común de programación 10.12

Usar el operador lógico AND (&&), en lugar del operador AND a nivel de bits (&), y viceversa.

El programa de la figura 10.9 demuestra el uso del operador AND a nivel de bits, del operador OR inclusivo a nivel de bits, del operador OR exclusivo a nivel de bits, y del operador de complemento a nivel de bits. El programa utiliza la función **display Bits** para imprimir los valores enteros **unsigned**. La salida se muestra en la figura 10.10.

En la figura 10.9, la variable entera **mask** es asignada al valor **1 (00000000 00000001)**, y a la variable **number1** se le asigna el valor **65535 (11111111 11111111)**. Cuando se combinan **mask** y **number1** utilizando el operador AND a nivel de bits (**&**) en la expresión **number1 & mask**, el resultado es **00000000 00000001**. Todos los bits, salvo el bit de orden inferior en la variable **number1** quedan "enmascarados" (ocultos), mediante la operación con el operador "AND" con la variable **mask**.

El operador OR inclusivo a nivel de bits se utiliza para definir en un operando bits específicos a 1. En la figura 10.9, la variable **number1** es asignada **15 (00000000 00001111)**, y la variable **setBits** es asignada **241 (00000000 11110001)**. Cuando se combinan **number1** y **setBits**, utilizando el operador OR a nivel de bits en la expresión **number1 | setBits**, el resultado es **255 (00000000 11111111)**. En la figura 10.11 se resumen los resultados de combinar dos bits con el operador OR inclusivo a nivel de bits.

Error común de programación 10.13

Usar el operador OR lógico (||), en lugar del operador OR a nivel de bits (|), y viceversa.

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

Fig. 10.8 Resultados de combinar dos bits mediante el operador AND a nivel de bits **&**.

```
/* Using the bitwise AND, bitwise inclusive OR, bitwise
   exclusive OR, and bitwise complement operators */

#include <stdio.h>

void displayBits(unsigned);

main()
{
   unsigned number1, number2, mask, setBits;

   number1 = 65535;
   mask = 1;
   printf("The result of combining the following\n");
   displayBits(number1);
   displayBits(mask);
   printf("using the bitwise AND operator & is\n");
   displayBits(number1 & mask);

   number1 = 15;
   setBits = 241;
   printf("\nThe result of combining the following\n");
   displayBits(number1);
   displayBits(setBits);
   printf("using the bitwise inclusive OR operator | is\n");
   displayBits(number1 | setBits);

   number1 = 139;
   number2 = 199;
   printf("\nThe result of combining the following\n");
   displayBits(number1);
   displayBits(number2);
   printf("using the bitwise exclusive OR operator ^ is\n");
   displayBits(number1 ^ number2);

   number1 = 21845;
   printf("\nThe one's complement of\n");
   displayBits(number1);
   printf("is\n");
   displayBits(~number1);

   return 0;
}
```

Fig. 10.9 Cómo utilizar el AND a nivel de bits, el OR inclusivo a nivel de bits, el OR exclusivo a nivel de bits, y el operador de complemento a nivel de bits (parte 1 de 2).

El operador OR exclusivo a nivel de bits (^) define cada bit en el resultado a 1, si *exactamente* uno de los bits, correspondiente en sus dos operandos, es 1. En la figura 10.9, las variables **number1** y **number2** se les asigna a los valores `139 (00000000 10001011)` y `199 (00000000 11000111)` respectivamente. Cuando se combinan estas variables con el operador OR exclusivo, en la expresión **number1 ^ number2**, el resultado es `00000000 01001100`. En la figura 10.12 se resumen los resultados de combinar dos bits utilizando el operador OR exclusivo a nivel de bits.


```
void displayBits(unsigned value)
{
   unsigned c, displayMask = 1 << 15;

   printf("%7u = ", value);

   for (c = 1; c <= 16; c++) {
      putchar(value & displayMask ? '1' : '0');
      value <<= 1;

      if (c % 8 == 0)
         putchar(' ');
   }

   putchar('\n');
}
```

Fig. 10.9 Cómo utilizar el AND a nivel de bits, OR inclusivo a nivel de bits, el OR exclusivo a nivel de bits, y el operador de complemento a nivel de bits (parte 2 de 2).

```
The result of combining the following
  65535 = 11111111 11111111
      1 = 00000000 00000001
using the bitwise AND operator & is
      1 = 00000000 00000001

The result of combining the following
     15 = 00000000 00001111
    241 = 00000000 11110001
using the bitwise inclusive OR opertor | is
    255 = 00000000 11111111

The result of combining the following
    139 = 00000000 10001011
    199 = 00000000 11000111
using the bitwise exclusive OR operator ^ is
     76 = 00000000 01001100

The one's complement of
  21845 = 01010101 01010101
is
  43690 = 10101010 10101010
```

Fig. 10.10 Salida correspondiente al programa de la figura 10.9.

El operador de complemento *a nivel de bits* (~) define todos los bits `1` existentes en su operando a `0` en el resultado y define todos los bits `0` a `1` en el resultado —o de otra forma conocido como "tomar el *complemento a uno* del valor". En la figura 10.9 la variable **number1** es asignada al valor `21845 (01010101 01010101)`. Cuando se evalúa la expresión **~number1** el resultado es `(10101010 10101010)`.

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Fig. 10.11 Resultados de combinar dos bits mediante el operador OR inclusivo a nivel de bits |.

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

Fig. 10.12 Resultados de combinar dos bits mediante el operador OR exclusivo a nivel de bits ^.

El programa de la figura 10.13 demuestra el operador de desplazamiento a la izquierda (<<) así como el operador de desplazamiento a la derecha (>>). La función **displayBits** se utiliza para imprimir los valores enteros **unsigned**.

El operador de desplazamiento a la izquierda (<<) desplaza los bits de su operando izquierdo hacia la izquierda, en el número de bits especificados en su operando derecho. Los bits desalojados a la derecha serán remplazados con 0s; los 1s que se desplazan hacia la izquierda se pierden. En el programa de la figura 10.13, la variable **number1** es asignada al valor **960 (00000011 11000000)**. El resultado de desplazar a la izquierda la variable **number1** 8 bits en la expresión **number1 << 8** es **49152 (11000000 00000000)**.

El operador de desplazamiento a la derecha (>>) desplaza los bits de su operando izquierda hacia la derecha, en el número de bits especificado por su operador derecho. Ejecutar un desplazamiento a la derecha en un entero **unsigned** hace que los bits desalojados a la izquierda sean remplazados por 0s; los 1s desplazados hacia la derecha se pierden. En el programa de la figura 10.13, el resultado de desplazar hacia la derecha **number1** en la expresión **number1>> 8** es **3 (00000000 00000011)**.

Error común de programación 10.14

El resultado de desplazar un valor queda indefinido si el operando derecho es negativo o si el operando derecho es más grande que el número de bits en el cual se almacena el operando izquierdo.

Sugerencia de portabilidad 10.7

El desplazamiento a la derecha es dependiente de la máquina. Desplazar a la derecha un entero signado, en algunas máquinas llena los bits desalojados con ceros y en otras con 1s.

Cada operador a nivel de bits (a excepción del operador de complemento a nivel de bits) tiene un operador de asignación correspondiente. Estos *operadores de asignación* a nivel de bits se muestran en la figura 10.14, y se utilizan de forma similar a los operadores de asignación aritméticos, presentados en el capítulo 3.


```
/* Using the bitwise shift operators */
#include <stdio.h>

void displayBits(unsigned);

main()
{
   unsigned number1 = 960;

   printf("\nThe result of left shifting\n");
   displayBits(number1);
   printf("8 bit positions using the ");
   printf("left shift operator << is\n");
   displayBits(number1 << 8);

   printf("\nThe result of right shifting\n");
   displayBits(number1);
   printf("8 bit positions using the ");
   printf("right shift operator >> is\n");
   displayBits(number1 >> 8);
   return 0;
}

void displayBits(unsigned value)
{
   unsigned c, displayMask = 1 << 15;

   printf("%7u = ", value);

   for (c = 1; c <= 16; c++) {
      putchar(value & displayMask ? '1' : '0');
      value <<= 1;

      if (c % 8 == 0)
         putchar(' ');
   }

   putchar('\n');
}
```

```
The result of left shifting
    960 = 00000011 11000000
8 bit positions using the left shift opertor << is
  49152 = 11000000 00000000

The result of right shifting
    960 = 00000011 11000000
8 bit positions using the right shift operator >> is
      3 = 00000000 00000011
```

Fig. 10.13 Cómo utilizar los operadores de desplazamiento a nivel de bits.

Operadores de asignación a nivel de bits	
`&=`	Operador de asignación AND a nivel de bits.
`|=`	Operador de asignación OR inclusivo a nivel de bits.
`^=`	Operador de asignación OR exclusivo a nivel de bits.
`<<=`	Operador de asignación de desplazamiento a la izquierda.
`>>=`	Operador de asignación de desplazamiento a la derecha.

Fig. 10.14 Los operadores de asignación a nivel de bits.

En la figura 10.15 se muestra la precedencia y asociatividad de los varios operadores presentados hasta este punto en el texto. Se muestran de arriba hacia abajo, en orden decreciente de precedencia.

10.10 Campos de bits

C proporciona la capacidad de especificar o definir el número de bits en el cual se almacena un miembro **unsigned** o **int** de una estructura o de una unión —conocidos como un *campo de bits*. Los campos de bits le permiten una mejor utilización de la memoria, al almacenar datos en el mínimo número de bits requeridos. Los miembros de campos de bits *deben* ser declarados como **int** o **unsigned**.

Operador	Asociatividad	Tipo
`() [] . ->`	de izquierda a derecha	el mas alto
`+ - ++ -- ! (tipo) & * ~ sizeof`	de derecha a izquierda	unario
`* / %`	de izquierda a derecha	multiplicativo
`+ -`	de izquierda a derecha	aditivo
`<< >>`	de izquierda a derecha	de desplazamiento
`< <= > >=`	de izquierda a derecha	relacional
`== !=`	de izquierda a derecha	igualdad
`&`	de izquierda a derecha	AND a nivel de bits
`^`	de izquierda a derecha	negación
`|`	de izquierda a derecha	OR a nivel de bits
`&&`	de izquierda a derecha	AND lógico
`||`	de izquierda a derecha	OR lógico
`?:`	de derecha a izquierda	condicional
`= += -= *= /= %= &= |= ^= <<= >>=`	de derecha a izquierda	asignación
	de izquierda a derecha	coma

Fig. 10.15 Precedencia y asociatividad de operadores.

Sugerencia de rendimiento 10.3

Los campos de bits ayudan a ahorrar almacenamiento.

Considere la siguiente definición de estructura:

```
struct bitCard {
   unsigned face : 4;
   unsigned suit : 2;
   unsigned color : 1;
};
```

La definición contiene tres campos de bits **unsigned** —**face**, **suit**, y **color**— utilizadas para representar una carta de un mazo de 52 cartas. Se declara un campo de bits, haciendo seguir a un nombre de miembro **unsigned** o **int** con un signo de dos puntos (**:**) y una constante entera que representa el *ancho* del campo, es decir, el número de bits en el cual queda almacenado el miembro. La constante que representa el ancho debe ser un entero entre 0 y el número total de bits utilizados para almacenar un **int** en su sistema. Nuestros ejemplos fueron probados en una computadora con enteros de dos bytes (16 bits).

La definición de estructura anterior indica que el miembro **face** está almacenada en 4 bits, el miembro **suit** en 2 bits y el miembro **color** en 1 bit. El número de bits se basa en el rango deseado de valores correspondiente a cada miembro de estructura. El miembro **face** almacena valores entre **0** (Ace) y **12** (Rey) —4 bits pueden almacenar un valor entre 0 y 15. El miembro **suit** almacena valores entre **0** y **3** (**0** = Diamantes, **1** = Corazones, **2** = Tréboles y **3** = Espadas)— 2 bits pueden almacenar un valor entre **0** y **3**. Finalmente, el miembro **color** almacena ya sea **0** (Rojo) o **1** (Negro) —1 bit puede almacenar ya sea **0** ó **1**.

El programa de la figura 10.16 (cuya salida se muestra en la figura 10.17), crea el arreglo **deck**, que contiene 52 estructuras **struct bitCard**. La función **fillDeck** inserta las 52 cartas en el arreglo **deck**, y la función **deal** imprime las 52 cartas. Note que se tiene acceso a los miembros de campos de bits de las estructuras exactamente como cualquier otro miembro de estructura. El miembro **color** se incluye para tener la posibilidad de indicar el color de la carta en un sistema que permita despliegues en color.

Es posible especificar un *campo de bits sin nombre*, en cuyo caso el campo se utiliza en la estructura como un *relleno*. Por ejemplo, la definición de estructura

```
struct example {
   unsigned a : 13;
   unsigned   : 3;
   unsigned b : 4;
};
```

usa como relleno un campo de 3 bits sin nombre —en estos tres bits no se puede almacenar nada. El miembro **b** (en nuestra computadora de palabras de 2 bytes) se almacena en otra unidad de almacenamiento.

Un *campo de bits sin nombre con ancho cero*, se utiliza para alinear el siguiente campo de bits en el límite de la nueva unidad de almacenamiento. Por ejemplo, la definición de estructura

```
struct example {
   unsigned a : 13;
   unsigned   : 0;
   unsigned b : 4;
};
```

utiliza un campo sin nombre de `0` bits para saltarse los bits restantes (tantos como existan) de la unidad de almacenamiento en la cual está almacenado **a**, y alinear **b** con el límite de la siguiente unidad de almacenamiento.

Sugerencia de portabilidad 10.8

Las manipulaciones de campos de bits son dependientes de la máquina. Por ejemplo, algunas computadoras permiten que los campos de bits crucen límites de palabras, en tanto que otras no lo permiten.

Error común de programación 10.15

Intentar tener acceso a bits individuales de un campo de bits como si fueran elementos de un arreglo. Los campos de bits no son "arreglos de bits".

Error común de programación 10.16

Intentar tomar la dirección de un campo de bits (el operador & no puede ser utilizado en conjunción con campos de bits, porque éstos no tienen direcciones).

Sugerencia de rendimiento 10.4

Aunque los campos de bits ahorran espacio, su uso puede hacer que el compilador genere código en lenguaje de máquina de ejecución más lenta. Esto ocurre debido a que tener acceso a sólo porciones de una unidad de almacenamiento direccionable toma más operaciones en lenguaje de máquina. Esto es uno de los muchos ejemplos de los tipos de intercambios espacio-tiempo que ocurren en la ciencia de la computación.

10.11 Constantes de enumeración

C proporciona un tipo final, definido por el usuario, conocido como una *enumeración*. Una enumeración, introducida por la palabra reservada **enum**, es un conjunto de constantes enteras representadas por identificadores. Estas *constantes de enumeración* son, en efecto, constantes simbólicas, cuyos valores pueden ser definidos automáticamente. Los valores de un **enum** se inician con `0`, a menos de que se defina de otra manera, y se incrementan en `1`. Por ejemplo, la enumeración

```
enum months {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,
             OCT, NOV, DEC};
```

crea un nuevo tipo en **enum months**, en el cual los identificadores son definidos automáticamente a los enteros `0` a `11`. Para numerar los meses 1 a 12, utilice la enumeración siguiente:

```
enum months {JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,
             SEP, OCT, NOV, DEC};
```

Dado que el primer valor de la enumeración anterior se define explícitamente en `1`, los valores subsiguientes se incrementan en `1` dando como resultado los valores `1` hasta `12`. Los identificadores en una enumeración deben ser únicos. En una enumeración el valor de cada constante de numeración puede ser establecido explícitamente en la definición, mediante la asignación de un valor al identificador. Varios miembros de una enumeración pueden tener el mismo valor entero. En el programa de la figura 10.18, la variable de enumeración **month** se utiliza en una estructura **for** para imprimir los meses del año del arreglo **monthName**. Note que hemos hecho **monthName[0]** la cadena vacía `""`. Algunos programadores pudieran preferir definir **monthName[0]** a un valor como `***ERROR***` para indicar que ocurrió un error lógico.


```
/* Example using a bit field */

#include <stdio.h>

struct bitCard {
   unsigned face : 4;
   unsigned suit : 2;
   unsigned color : 1;
};

typedef struct bitCard Card;

void fillDeck(Card *);
void deal(Card *);

main()
{
   Card deck[52];

   fillDeck(deck);
   deal(deck);

   return 0;
}

void fillDeck(Card *wDeck)
{
   int i;

   for (i = 0; i <= 51; i++) {
      wDeck[i].face = i % 13;
      wDeck[i].suit = i / 13;
      wDeck[i].color = i / 26;
   }
}

/* Function deal prints the cards in two column format */
/* Column 1 contains cards 0-25 subscripted with k1 */
/* Column 2 contains cards 26-51 subscripted with k2 */

void deal(Card *wDeck)
{
   int k1, k2;

   for (k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++) {
      printf("Card:%3d  Suit:%2d  Color:%2d   ",
             wDeck[k1].face, wDeck[k1].suit, wDeck[k1].color);
      printf("Card:%3d  Suit:%2d  Color:%2d\n",
             wDeck[k2].face, wDeck[k2].suit, wDeck[k2].color);
   }
}
```

Fig. 10.16 Cómo utilizar campos de bits para almacenar un mazo de cartas.

```
Card:  0 Suit: 0 Color: 0 Card:  0 Suit: 2 Color: 1
Card:  1 Suit: 0 Color: 0 Card:  1 Suit: 2 Color: 1
Card:  2 Suit: 0 Color: 0 Card:  2 Suit: 2 Color: 1
Card:  3 Suit: 0 Color: 0 Card:  3 Suit: 2 Color: 1
Card:  4 Suit: 0 Color: 0 Card:  4 Suit: 2 Color: 1
Card:  5 Suit: 0 Color: 0 Card:  5 Suit: 2 Color: 1
Card:  6 Suit: 0 Color: 0 Card:  6 Suit: 2 Color: 1
Card:  7 Suit: 0 Color: 0 Card:  7 Suit: 2 Color: 1
Card:  8 Suit: 0 Color: 0 Card:  8 Suit: 2 Color: 1
Card:  9 Suit: 0 Color: 0 Card:  9 Suit: 2 Color: 1
Card: 10 Suit: 0 Color: 0 Card: 10 Suit: 2 Color: 1
Card: 11 Suit: 0 Color: 0 Card: 11 Suit: 2 Color: 1
Card: 12 Suit: 0 Color: 0 Card: 12 Suit: 2 Color: 1
Card:  0 Suit: 1 Color: 0 Card:  0 Suit: 3 Color: 1
Card:  1 Suit: 1 Color: 0 Card:  1 Suit: 3 Color: 1
Card:  2 Suit: 1 Color: 0 Card:  2 Suit: 3 Color: 1
Card:  3 Suit: 1 Color: 0 Card:  3 Suit: 3 Color: 1
Card:  4 Suit: 1 Color: 0 Card:  4 Suit: 3 Color: 1
Card:  5 Suit: 1 Color: 0 Card:  5 Suit: 3 Color: 1
Card:  6 Suit: 1 Color: 0 Card:  6 Suit: 3 Color: 1
Card:  7 Suit: 1 Color: 0 Card:  7 Suit: 3 Color: 1
Card:  8 Suit: 1 Color: 0 Card:  8 Suit: 3 Color: 1
Card:  9 Suit: 1 Color: 0 Card:  9 Suit: 3 Color: 1
Card: 10 Suit: 1 Color: 0 Card: 10 Suit: 3 Color: 1
Card: 11 Suit: 1 Color: 0 Card: 11 Suit: 3 Color: 1
Card: 12 Suit: 1 Color: 0 Card: 12 Suit: 3 Color: 1
```

Fig. 10.17 Salida del programa de la figura 10.16.

Error común de programación 10.17

Es un error de sintaxis asignar un valor a una constante de numeración después de haber sido definida.

Práctica sana de programación 10.6

Utilice sólo letras mayúsculas en los nombres de las constantes de numeración. Esto hace que estas constantes destaquen en un programa y le recuerdan al programador que las constantes de numeración no son variables.

Resumen

- Las estructuras son colecciones de variables relacionadas, algunas veces conocidas como agregados, bajo un solo nombre.
- Las estructuras pueden contener variables de varios tipos de datos.
- La palabra reservada **`struct`** empieza toda definición de estructura. Dentro de las llaves de la definición de estructura, están las declaraciones de los miembros de la estructura.
- Los miembros de la misma estructura deben de tener nombres únicos.


```
/* Using an enumeration type */
#include <stdio.h>

enum months {JAN = 1, FEB, MAR, APR, MAY, JUN,
             JUL, AUG, SEP, OCT, NOV, DEC};

main()
{
   enum months month;
   char *monthName[] = {"", "January", "February", "March",
                        "April", "May", "June", "July",
                        "August", "September", "October",
                        "November", "December"};

   for (month = JAN; month <= DEC; month++)
      printf("%2d%11s\n", month, monthName[month]);

   return 0;
}
```

```
 1    January
 2   February
 3      March
 4      April
 5        May
 6       June
 7       July
 8     August
 9  September
10    October
11   November
12   December
```

UNIVERSIDAD DE LA REPUBLICA
FACULTAD DE INGENIERIA
DEPARTAMENTO DE
DOCUMENTACION Y BIBLIOTECA

Fig. 10.18 Cómo utilizar una enumeración.

- Una definición de estructura crea un nuevo tipo de datos que puede ser utilizado para declarar variables.
- Existen dos métodos para declarar variables de estructura. El primer método es declarar las variables en una declaración, como se hace con las variables de otros tipos de datos, utilizando **`struct tagName`** como el tipo. El segundo método es incluir las variables encerradas en las llaves de la definición de estructura y en el punto y coma que termina la definición de estructura.
- El nombre de rótulo de la estructura es opcional. Si la estructura se define sin un nombre de rótulo, las variables del tipo de datos derivados deben de ser declarados en la definición de estructura, y no se pueden declarar otras variables del nuevo tipo de estructura.
- Una estructura puede ser inicializada con una lista de incialización, siguiendo el nombre de la variable en la declaración de estructura con un signo igual y una lista de inicializadores, separados por comas y encerrados en llaves. Si en la lista existen menos inicializadores que

miembros en la estructura, los miembros restantes serán automáticamente inicializados a cero (o a **NULL** si el miembro es un apuntador).

- Estructuras completas pueden ser asignadas a variables de estructura del mismo tipo.
- Una variable de estructura puede ser inicializada con una variable de estructura del mismo tipo.
- El operador de miembro de estructura se utiliza al tener acceso a un miembro de una estructura vía el nombre de la variable de estructura.
- El operador de apuntador de estructura —creado con un signo de menos (-) y un signo de mayor que (>)— se utiliza al tener acceso a un miembro de una estructura vía un apuntador a la estructura.
- Las estructuras y los miembros individuales de las estructuras se pasan a las funciones en llamada por valor.
- Para pasar a una estructura llamada por referencia, pase la dirección de la variable de estructura.
- Un arreglo de estructura se pasa automáticamente en llamada por referencia.
- Para pasar un arreglo en llamada por valor, cree una estructura con el arreglo como un miembro.
- Crear un nuevo nombre utilizando **typedef** no crea un nuevo tipo; crea un nombre que es un seudónimo del tipo anteriormente definido.
- Una unión es un tipo de datos derivado, cuyos miembros comparten el mismo espacio de almacenamiento. Los miembros pueden ser de cualquier tipo.
- El almacenamiento reservado para una unión debe ser lo suficientemente grande para almacenar su miembro mayor. En la mayoría de los casos, las uniones contienen dos o más tipos de datos. Solamente un miembro y, por lo tanto, un solo tipo de datos, pueden ser referenciados en un momento dado.
- Una unión se declara con la palabra reservada **union**, en el mismo formato que una estructura.
- Una unión puede ser inicializada únicamente con el valor del tipo de su primer miembro.
- El operador AND a nivel de bits (**&**) toma dos operandos integrales. Un bit en el resultado se define a 1 si los bits correspondientes en cada uno de los operandos son 1.
- Se utilizan máscaras para ocultar algunos bits mientras se conservan otros.
- El operador OR inclusivo a nivel de bits (|) toma dos operandos. Un bit en el resultado se define a 1 si el bit correspondiente en cualquier operando está definido a 1.
- Cada uno de los operadores a nivel de bits (a excepción del operador de complemento a nivel de bits unario) tiene un operador de asignación correspondiente.
- El operador OR exclusivo a nivel de bits (^) toma dos operandos. Un bit en el resultado se define a 1 si exactamente 1 de los bits correspondientes en los dos operandos está definido a 1.
- El operador de desplazamiento de izquierda (<<) desplaza los bits de su operando izquierdo hacia la izquierda en el número de bits especificados por su operando derecho. Los bits desalojados a la derecha se remplazan con 0s.
- El operador de desplazamiento a la derecha (>>) desplaza los bits de su operando izquierdo hacia la derecha en el número de bits especificado en su operando derecho. El ejecutar un desplazamiento a la derecha en un entero no signado hace que los bits desocupados a la izquierda sean remplazados por cero. Los bits desocupados en enteros signados podrían ser remplazados con 0s o con 1s —esto dependerá de la máquina.

- El operador de complemento a nivel de bits (~) toma un operando e invierte sus bits —esto produce el complemento a uno del operando.
- Los campos de bits reducen la utilización del almacenamiento al almacenar datos en el número mínimo de bits requeridos.
- Los miembros de campos de bits deben de ser declarados como **int** o **unsigned.**
- Un campo de bits se declara haciendo seguir un nombre de miembro **unsigned** o **int** con un punto y coma, y con el ancho del campo de bits.
- El ancho del campo de bits debe ser una constante entera, entre 0 y el número total de bits utilizados para almacenar una variable **int** en su sistema.
- Si un campo de bits se especifica sin nombre, el campo se utilizará como relleno en la estructura.
- Un campo de bits sin nombre con ancho **0** se utiliza para alinear el siguiente campo de bits en el límite de la siguiente palabra de máquina.
- Una enumeración, designada con la palabra reservada **enum**, es un conjunto de enteros que se representan mediante identificadores. Los valores de un **enum** se inician con **0**, a menos de que se especifique lo contrario, y son siempre incrementados en **1**.

Terminología

^ operador OR exclusivo a nivel de bits
^= operador de asignación OR exclusivo a nivel de bits
~ operador de complemento a uno
& operador AND a nivel de bits
&= operador de asignación AND a nivel de bits
| operador OR inclusivo a nivel de bits.
|= operador de asignación OR inclusivo a nivel de bits.
<< operador de desplazamiento a la izquierda
<<= operador de asignación de desplazamiento a la izquierda
>> operador de desplazamiento a la derecha
>>= operador de asignación de desplazamiento a la derecha
como tener acceso a miembros de estructuras
agregados
arreglo de estructuras
campo de bits
operador a nivel de bits
complementar
tipo derivado
enumeración
constante de enumeración
inicialización de estructuras
desplazamiento a la izquierda
máscara
enmascarar bits
miembro
nombre de miembro
estructuras anidadas
complemento a uno
relleno
apuntador a una estructura
tipos de datos definidos por el programador
registro
desplazamiento a la derecha
estructura autorreferenciada
desplazar
intercambios espacio-tiempo
struct
asignación de estructura
declaración de estructura
definición de estructura
inicialización de estructura
operador de miembro de estructura (punto) (.)
nombre de estructura
operador de apuntador de estructura (flecha) (->)
etiqueta de estructura
tipo de estructura
nombre de etiqueta
typedef
union
campo de bits sin nombre
ancho de un campo de bits
campo de bits de ancho cero

Errores comunes de programación

10.1 Olvidar el punto y coma que da por terminada una definición de estructura.
10.2 Asignar una estructura de un tipo a una estructura de un tipo distinto.
10.3 Es un error de sintaxis comparar estructuras, debido a diferentes requisitos de alineación en los diferentes sistemas.
10.4 Insertar un espacio entre el signo de - y el signo de > del operador de apuntador de estructura (o insertar espacios entre los componentes de cualquier otro operador múltiple de teclado, a excepción de `?:`).
10.5 Intentar referirse a un miembro de una estructura utilizando únicamente el nombre de dicho miembro.
10.6 No utilizar paréntesis al referirse a un miembro de estructura utilizando un apuntador y el operador de miembro de estructura (por ejemplo `*aptr.suit`, es un error de sintaxis).
10.7 Suponer que las estructuras, como los arreglos, se pasan automáticamente en llamada por referencia, e intentar modificar los valores de estructura del llamador en la función llamada.
10.8 Olvidar incluir el subíndice de arreglo al referirse a estructuras individuales de un arreglo de estructuras.
10.9 Es un error lógico referenciar con el tipo equivocado, datos en una unión almacenados con un tipo distinto.
10.10 Es un error de sintaxis comparar uniones, debido a los diferentes requisitos de alineación en varios sistemas.
10.11 Inicializar una unión en una declaración con un valor cuyo tipo es diferente del tipo del primer miembro de la unión.
10.12 Usar el operador lógico AND (`&&`), en lugar del operador AND a nivel de bits (`&`), y viceversa.
10.13 Usar el operador OR lógico (`||`), en lugar del operador OR a nivel de bits (`|`), y viceversa.
10.14 El resultado de desplazar un valor queda indefinido si el operando derecho es negativo o si el operando derecho es más grande que el número de bits en el cual se almacena el operando izquierdo.
10.15 Intentar tener acceso a bits individuales de un campo de bits como si fueran elementos de un arreglo. Los campos de bits no son "arreglos de bits".
10.16 Intentar tomar la dirección de un campo de bits (el operador `&` no puede ser utilizado en conjunción con campos de bits, porque estos no tienen direcciones).
10.17 Es un error de sintaxis asignar un valor a una constante de numeración después de haber sido definida.

Prácticas sanas de programación

10.1 Al crear un tipo de estructura proporcione un nombre de rótulo de estructura. El nombre de rótulo de estructura es conveniente más adelante en el programa para la declaración de nuevas variables de este tipo de estructura.
10.2 Seleccionar un nombre de rótulo de estructura significativo ayuda a autodocumentar el programa.
10.3 Evite utilizar los mismos nombres para miembros de estructura de distintos tipos. Ello es permitido, pero podría causar confusión.
10.4 No deje espacios alrededor de los operadores `->` y `.` ya que ayuda a enfatizar que las expresiones en las cuales los operadores están contenidos son esencialmente nombres individuales de variables.
10.5 Ponga los nombres `typedef` en mayúsculas, para enfatizar que esos nombres son sinónimos de otros nombres de tipo.
10.6 Utilice sólo letras mayúsculas en los nombres de las constantes de numeración. Esto hace que estas constantes destaquen en un programa y le recuerdan al programador que las constantes de numeración no son variables.

Sugerencia de portabilidad

10.1 Dado que depende de la máquina el tamaño de los elementos de datos de un tipo particular, y debido a que las consideraciones de alineación de almacenamiento también son dependientes de la máquina, entonces también lo será la representación de una estructura.
10.2 Utilice `typedef` para ayudar a hacer más portátil un programa.
10.3 Si en una unión los datos se almacenan como de un tipo y se referencían como de otro tipo, los resultados serán dependientes de la instalación.
10.4 La cantidad de almacenamiento requerido para almacenar una unión es dependiente de la instalación.
10.5 En algunas uniones no pueden aplicarse fácilmente otros sistemas de computación. Si una unión es portable o no, a menudo depende de la alineación de almacenamiento requerida para los tipos de datos de miembros de unión en un sistema dado.
10.6 Las manipulaciones de datos a nivel de bits son dependientes de la máquina.
10.7 El desplazamiento a la derecha es dependiente de la máquina. Desplazar a la derecha un entero signado, en algunas máquinas llena los bits desalojados con ceros y en otras con `1s`.
10.8 Las manipulaciones de campos de bits son dependientes de la máquina. Por ejemplo, algunas computadoras permiten que los campos de bits crucen límites de palabras, en tanto que otras no lo permiten.

Sugerencia de rendimiento

10.1 Es más eficaz pasar estructuras en llamada por referencia que pasar estructuras en llamada por valor (ya que esto último requiere que toda la estructura se copie).
10.2 Las uniones ahorran almacenamiento.
10.3 Los campos de bits ayudan a ahorrar almacenamiento.
10.4 Aunque los campos de bits ahorran espacio, su uso puede hacer que el compilador genere código en lenguaje de máquina de ejecución más lenta. Esto ocurre debido a que tener acceso a sólo porciones de una unidad de almacenamiento direccionable toma más operaciones en lenguaje de máquina. Esto es uno de los muchos ejemplos de los tipos de intercambios espacio-tiempo que ocurren en la ciencia de la computación.

Observación de ingeniería de software

10.1 Al igual que en una declaración `struct`, una declaración `union` simplemente crea un tipo nuevo. Colocar una declaración `union` o `struct` fuera de cualquier función no crea una variable global.

Ejercicios de autoevaluación

10.1 Llene los espacios en blanco con cada uno de los siguientes:
a) Una _____ es una colección de variables relacionadas bajo un nombre.
b) Una _____ es una colección de variables bajo un nombre en la cual las variables comparten el mismo almacenamiento.
c) Los bits en el resultado de una expresión utilizando el operador _____ se definen a 1 si los bits correspondientes en cada operando están establecidos en 1. De lo contrario, los bits se definen a cero.
d) Las variables declaradas en una definición de estructura se conocen como sus _____.
e) Los bits en el resultado de una expresión utilizando el operador _____ se definen a 1, si por lo menos uno de los bits correspondientes en cualquiera de los operandos está definido a 1. De lo contrario los bits se establecen a cero.
f) La palabra reservada _____ introduce una declaración de estructura.

g) La palabra reservada _____ se utiliza para crear un seudónimo de un tipo de datos previamente definido.
h) Los bits en el resultado de una expresión utilizando el operador _____ se definen a 1, si exactamente uno de los bits correspondientes en cada uno de los operandos está definido a uno. De lo contrario, los bits se definen a cero.
i) El operador AND a nivel de bits `&`, se utiliza a menudo para _____ a los bits, esto es para seleccionar ciertos bits a partir de una cadena de bits, en tanto que se ponen a cero los demás.
j) La palabra reservada _____ se utiliza para introducir una definición de unión.
k) El nombre de la estructura se conoce como el _____ de la estructura.
l) Se tiene acceso a un miembro de estructura ya sea con el operador _____ o con el operador _____.
m) Los operadores _____ y _____ se utilizan para desplazar los bits de un valor hacia la izquierda o hacia la derecha, respectivamente.
n) Una _____ es un conjunto de enteros representados por identificadores.

10.2 Indique si cada uno de los siguientes es verdadero o falso. Si es falso, explique por qué.
a) Las estructuras pueden contener únicamente un tipo de datos.
b) Dos uniones pueden ser comparadas entre sí para determinar si son iguales.
c) El nombre del rótulo de una estructura es opcional.
d) Los miembros de diferentes estructuras deben tener nombres únicos.
e) La palabra reservada `typedef` se utiliza para definir nuevos tipos de datos.
f) Las estructuras se pasan siempre a las funciones en llamada por referencia.
g) Las estructuras no pueden ser comparadas.

10.3 Escriba un solo enunciado o un conjunto de enunciados para llevar a cabo cada uno de los siguientes:
a) Defina una estructura llamada `part` conteniendo la variable `int partNumber`, y el arreglo `char partName` cuyos valores pudieran ser de hasta 25 caracteres de largo.
b) Defina `Part` como un sinónimo para el tipo `struct part`.
c) Utilice `Part` para declarar la variable `a` que sea de tipo `struct part`, el arreglo `b[10]` que sea del tipo `struct part` y la variable `ptr` que sea del tipo apuntador a `struct part`.
d) Lea un número de parte y un nombre de parte del teclado a los miembros individuales de la variable `a`.
e) Asigne los valores del miembro de la variable `a` al elemento 3 del arreglo `b`.
f) Asigne las direcciones del arreglo `b` a la variable de apuntador `ptr`.
g) Imprima los valores de miembros del elemento 3 del arreglo `b`, utilizando la variable `ptr` y el operador de apuntador de estructura para referirse a los miembros.

10.4 Encuentre el error en cada uno de los siguientes:
a) Suponga que `struct car` ha sido definido conteniendo dos apuntadores al tipo `char`, es decir, `face` y `suit`. También, la variable `c` ha sido declarada del tipo `struct card` y la variable `cPtr` ha sido declarada ser del tipo apuntador a `struct card`. La variable `cPtr` ha sido asignada a la dirección de `c`.

```
printf("%s\n", *cPtr->face);
```

b) Suponga que `struct card` ha sido definida conteniendo dos apuntadores de tipo `char`, es decir `face` y `suit`. También, el arreglo `hearts[13]` ha sido declarado ser del tipo `struct card`. El siguiente enunciado debería imprimir el miembro `face` del elemento 10 del arreglo.

```
printf("%s\n", hearts.face);
```

c)
```
union values {
   char w;
   float x;
   double y;
} v = {1.27};
```


d)
```
struct person {
   char last Name[15]
   char firstName[15]
   int age;
}
```

e) Suponga que `struct person` ha sido definido como en la parte (d) pero con la corrección apropiada.

```
person d;
```

f) Suponga que la variable `p` ha sido declarada como del tipo `struct person` y la variable `c` ha sido declarada del tipo `struct card`.

```
p = c;
```

Respuestas a los ejercicios de autoevaluación

10.1 a) estructura. b) unión. c) AND a nivel de bits (`&`). d) miembros. e) OR inclusivo a nivel de bits (`|`). f) `struct`. g) `typedef`. h) OR exclusivo a nivel de bits (`^`). i) máscara. j) `union`. k) etiqueta. l) miembro de estructura, apuntador de estructura. m) operador de desplazamiento a la izquierda (`<<`), operador de desplazamiento a la derecha (`>>`). n) enumeración.

10.2 a) Falso. Una estructura puede contener muchos tipos de datos.
b) Falso. Las uniones no pueden ser comparadas, debido a los mismos problemas de alineación asociados con las estructura.
c) Verdadero.
d) Falso. Los miembros de estructuras separadas pueden tener los mismos nombres, pero los miembros de una misma estructura deben tener nombres únicos.
e) Falso. La palabra reservada `typedef` se utiliza para definir nuevos nombres (sinónimos) para tipos de datos definidos previamente.
f) Falso. Las estructuras son siempre pasadas a las funciones en llamada por valor.
g) Verdadero, debido a los problemas de alineación.

10.3 a)
```
struct part {
int partNumber;
char partName[25];
};
```
b) `typedef struct part Part;`
c) `Part a, b[10], *ptr;`
d `scanf("%d%s", &a.partNumber, &a.partName};`
e) `b[3] = a;`
f) `ptr = b;`
g)
```
printf("%d %s\n", (ptr + 3)->partNumber,
(ptr + 3)-partName);
```

10.4 a) Error: los paréntesis que deberían de encerrar a `*cPtr` han sido omitidos, causando que sea incorrecto el orden de evaluación de la expresión.
b) Error: El subíndice del arreglo ha sido omitido. La expresión debería ser `hearts[10].face`.
c) Error: Una unión solamente puede ser inicializada con un valor que tenga el mismo tipo que el primer miembro de la misma.
d) Error: se requiere de un punto y coma para determinar una definición de estructura.
e) Error: la palabra reservada `struct` fue omitida de la declaración de variable.
f) Error: las variables de tipos de estructuras diferentes no pueden ser asignadas unos a los otros.

Ejercicios

10.5 Dé la definición de cada una de las siguientes estructuras y uniones:

a) La estructura `inventory` que contiene el arreglo de caracteres `partName[30]`, en entero `partNumber`, el punto flotante `price`, el entero `stock`, y el entero `reorder`.
b) La unión `data` que contiene `char c, short s, long l, float f` y `double d`.
c) Una estructura llamada `address` que contiene los arreglos de caracteres `streetAddress[25], city[20], state[3]`, y `zipCode[6]`.
d) La estructura `student` que contiene los arreglos `firstName[15]` y `lastName[15]`, y la variable `homeAddress` del tipo `struct address` correspondiente a la parte (c).
e) La estructura `test` que contenga 16 campos de bits con anchos de 1 bit. Los nombres de los campos de bits son las letras `a` a la `p`.

10.6 Dadas las siguientes definiciones de estructuras y las declaraciones de variables,

```
struct customer {
    char lastName[15];
    char firstName[15];
    int customerNumber;

    struct {
       char phoneNumber[11];
       char address[50];
       char city[15];
       char state[3];
       char zipCode[6];
    } personal;

} customerRecord, *customerPtr;

customerPtr = &customerRecord;
```

escriba una expresión por separado que pueda ser utilizada para tener acceso a los miembros de la estructura en cada una de las partes siguientes.

a) El miembro `lastName` de la estructura `customerRecord`.
b) El miembro `lastName` de la estructura a la cual apunta `customerPtr`.
c) El miembro `firstName` de la estructura `customerRecord`.
d) El miembro `firstName` de la estructura a la cual apunta `customerPtr`.
e) El miembro `customerNumber` de la estructura `customerRecord`.
f) El miembro `customerNumber` de la estructura a la cual apunta `customerPtr`.
g) El miembro `phoneNumber` del miembro `personal` de la estructura `customerRecord`.
h) El miembro `phoneNumber` del miembro `personal` de la estructura a la cual apunta `customerPtr`.
i) El miembro `address` del miembro `personal` de la estructura `customerRecord`.
j) El miembro `address` del miembro `personal` de la estructura apuntada por `customerPtr`.
k) El miembro `city` del miembro `personal` de la estructura `customerRecord`.
l) El miembro `city` del miembro `personal` de la estructura a la cual apunta `customerPtr`.
m) El miembro `state` del miembro `personal` de la estructura `customerRecord`.
n) El miembro `state` del miembro `personal` de la estructura a la cual apunta `customerPtr`.
o) El miembro `zipCode` del miembro `personal` de la estructura `customerRecord`.
p) El miembro `zipCode` del miembro `personal` de la estructura a la cual apunta `customerPtr`.

10.7 Modifique el programa de la figura 10.16 para barajar las cartas utilizando un algoritmo de barajar de alto rendimiento (como se muestra en la figura 10.3). Imprima el mazo resultante en un formato de dos columnas, como en la figura 10.4. Anteceda cada carta con su color.

10.8 Crear la unión `integer` con miembros `char c, short s, int i`, y `long l`. Escriba un programa que introduzca el valor del tipo `char, short, int`, y `long`, y que almacene los valores en las variables de unión del tipo `union integer`. Cada variable de unión deberá ser impresa como un `char`, un `short`, un `int` y un `long`. ¿Se imprimen siempre los valores en forma correcta?

10.9 Crear la unión `floatingPoint` con los miembros `float f, double d`, y `long double l`. Escriba un programa que introduzca valor del tipo `float, double` y `long double`, y almacene los valores en variables de unión del tipo `unionfloatingPoint`. Cada variable de unión deberá imprimirse como un `float`, un `double` y un `long double`. ¿Se imprimen siempre los valores correctamente?

10.10 Escriba un programa que desplace una variable entera 4 bits hacia la derecha. El programa deberá imprimir el entero en bits antes y después de la operación de desplazamiento. ¿Su sistema coloca ceros, o bien unos en los bits desalojados?

10.11 Si su computadora utiliza enteros de 4 bytes, modifique el programa de la Figura 10.7, de tal forma que funcione con enteros de 4 bytes.

10.12 El desplazar a la izquierda un entero `unsigned` en 1 bit es equivalente a multiplicar el valor por 2, Escriba la función `power2` que toma dos argumentos enteros `number` y `pow` y calcule

```
number * 2^pow
```

Utilice el operador de desplazamiento para calcular el resultado. El programa deberá imprimir los valores como enteros y como bits.

10.13 El operador de desplazamiento a la izquierda puede ser utilizado para empacar dos valores de caracteres en una variable entera no signada de 2 bytes. Escriba un programa que introduzca dos caracteres del teclado y que los pase a la función `packCharacters`. Para empacar dos caracteres en una variable entera `unsigned`, asigne el primer carácter a la variable `unsigned`, desplace la variable a la izquierda en 8 posiciones de bits, y combine la variable `unsigned` con el segundo carácter utilizando el operador OR inclusivo a nivel de bits. El programa deberá extraer los caracteres en su formato de bits, antes y después de haber sido empacados en el entero `unsigned`, para probar que los caracteres de hecho han sido empacados correctamente en la variable `unsigned`.

10.14 Utilizando el operador de desplazamiento a la derecha, el operador AND a nivel de bits y una máscara, escriba la función `unpackCharacters` que toma el entero `unsigned` del Ejercicio 10.13 y lo desempaca en dos caracteres. Para desempacar dos caracteres de un entero `unsigned` de 2 bytes, combine el entero `unsigned` con la máscara `65280 (11111111 00000000)` y desplace hacia la derecha el resultado en 8 bits. Asigne el valor resultante a una variable `char`. A continuación combine el entero `unsigned` con la máscara `255 (00000000 11111111)`. Asigne el resultado a otra variable `char`. El programa deberá imprimir el entero `unsigned` en bits, antes de ser desempacado, y a continuación imprimir los caracteres en bits para confirmar que fueron desempacados correctamente.

10.15 Si su sistema utiliza enteros de 4 bytes, vuelva a escribir el programa de Ejercicio 10.13 para empacar 4 caracteres.

10.16 Si su sistema utiliza enteros de 4 bytes, vuelva a escribir la función `unpackCharacters` del Ejercicio 10.14 para desempacar 4 caracteres. Crear las máscaras que necesite para desempacar los 4 caracteres desplazando hacia la izquierda el valor 255 en la variable de enmascaramiento en 8 bits 0, 1, 2 o 3 veces, (dependiendo del byte que está desempacando).

10.17 Escriba un programa que invierta el orden de los bits de un valor entero no signado. El programa deberá introducir el valor proveniente del usuario y llamar a la función **reverseBits** para imprimir los bits en orden inverso. Imprima el valor en bits tanto antes como después de la inversión de bits, para confirmar que los bits hayan sido invertidos correctamente.

10.18 Modifique la función **displayBits** de la Figura 10.7, de tal forma que resulte portátil entre sistemas, utilizando enteros de 2 bytes y sistemas de enteros de 4 bytes. *Sugerencia:* utilice el operador **sizeof** para determinar el tamaño de un entero en una máquina en particular.

10.19 El programa siguiente utiliza la función **multiple** para determinar si el entero introducido desde el teclado es un múltiplo de algún entero **x**. Examine la función múltiple y a continuación determine el valor de **x**.

```
/* This program determines if a value is a multiple of X */

#include <stdio.h>

int multiple(int);

main()
{
   int y;

   printf("Enter an integer between 1 and 32000: ");
   scanf("%d", &y);

   if (multiple(y))
      printf("%d is a multiple of X\n", y);
   else
      printf("%d is not a multiple of X\n", y);

   return 0;
}

int multiple(int num)
{
   int i, mask = 1, mult = 1;

   for (i = 1; i <= 10; i++, mask <<= 1)
      if ((num & mask) != 0) {
         mult = 0;
         break;
      }

    return mult;
}
```


10.20 ¿Qué es lo que ejecuta el siguiente programa?

```
#include <stdio.h>

int mystery(unsigned);

main()
{
   unsigned x;

   printf("Enter an integer: ");
   scanf("%u", &x);
   printf("The result is %d\n", mystery(x));
   return 0;
}

int mystery(unsigned bits)
{
   unsigned i, mask = 1 << 15, total = 0;

   for (i = 1; i <= 16; i++, bits <<= 1)
      if ((bits & mask) == mask)
         ++total;

    return total % 2 == 0 ? 1 : 0;;
}
```

11

Procesamiento de archivos

Objetivos

- Ser capaz de crear, leer, escribir y actualizar archivos.
- Familiarizarse con el proceso de archivos de acceso secuencial.
- Familiarizarse con el proceso de archivos de acceso directo.

Leo una parte en su totalidad.
Samuel Goldwyn

¡Saluden!
La bandera está pasando.
Henry Holcomb Bennett

La conciencia ... no se presenta a sí misma dividida en pedazos ... Los símiles mediante los cuales se le puede describir más naturalmente serían un "río" o una "corriente".
William James

Debo suponer que un documento "No archivar" se archivará en un archivo "No archivar".
Senador Frank Church
Senate Intelligence Subcommittee Hearing, 1975

Sinopsis

Resumen • Terminología • Errores comunes de programación • Prácticas sanas de programación • Sugerencias de rendimiento • Sugerencias de portabilidad • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios.

11.1 Introducción

El almacenamiento de datos en variables y en arreglos es temporal; al terminar un programa todos estos datos se pierden. Para la conservación permanente de grandes cantidades de datos se utilizan los *archivos*. Las computadoras almacenan los archivos en dispositivos de almacenamiento secundario, especialmente en dispositivos de almacenamiento en disco. En este capítulo, explicaremos como los programas en C crean, actualizan y procesan los archivos de datos. Analizaremos tanto archivos de acceso secuencial como archivos de acceso directo.

11.2 La jerarquía de datos

En último término, todos los elementos de datos procesados por una computadora se reducen a combinaciones de ceros y de unos. Es así porque es simple y económico construir dispositivos electrónicos que puedan asumir dos estados estables —uno de los estados representando `0` y el otro `1`. Es verdaderamente asombroso que las impresionantes funciones ejecutadas por las computadoras sólo involucren el manejo más básico de `0`s y de `1`s.

En una computadora el elemento de datos más pequeño puede asumir el valor `0` o el valor `1`. Este elemento de datos se conoce como un *bit* (abreviatura de *"binary digit"* —un dígito que puede asumir uno de dos valores). Los circuitos de la computadora ejecutan varias manipulaciones simples de bits, como es determinar el valor de un bit, establecer el valor de un bit, e invertir un bit (de `1` a `0` o de `0` a `1`).

Para los programadores resulta muy engorroso trabajar con datos bajo su forma de nivel bajo, es decir los bits. En vez de ello, los programadores prefieren trabajar con datos en forma de *dígitos decimales* (es decir 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9), *letras* (es decir, de la A a la Z, y a hast a la z) y *símbolos especiales* (es decir, $, @, %, &, *, (,), -, +, ", :, ?, /, y muchos otros). Los *dígitos, las letras y los símbolos especiales se conocen como caracteres*. El conjunto de todos los caracteres que pudieran

ser utilizados para escribir programas y representar elementos de datos de una computadora en particular se llama el *conjunto de caracteres* de dicha computadora. Dado que las computadoras sólo pueden procesar `1`s y `0`s, todo carácter de un conjunto de caracteres en una computadora es representado como un patrón de `1`s y `0`s (llamados un *byte*). Hoy día, los bytes están comúnmente formados por ocho bits. Los programadores crean programas y elementos de datos como caracteres; a continuación, las computadoras manipulan y procesan dichos caracteres como patrones de bits.

Al igual que los caracteres están formados por bits, los *campos* se componen de caracteres. Un campo es un grupo de caracteres que contiene un significado. Por ejemplo, un campo que consista únicamente de letras mayúsculas y minúsculas, puede ser utilizado para representar el nombre de una persona.

Los elementos de datos procesados por las computadoras forman una *jerarquía de datos,* en la cual los elementos de datos se convierten en más grandes y más complejos en cuanto a estructura conforme progresamos desde los bits, hacia los caracteres (bytes), hacia los campos y así sucesivamente.

Un *registro* (es decir, un `struct` en C) se compone de varios campos. En un sistema de nómina, por ejemplo, un registro para un empleado en particular pudiera estar formado por los campos siguientes:

1. Número de seguridad social
2. Nombre
3. Dirección
4. Tasa horaria de salario
5. Número de excepciones reclamadas
6. Ganancias acumuladas año a la fecha
7. Cantidades retenidas de impuestos federales, etcétera

Entonces, un registro es un grupo de campos relacionados. En el ejemplo anterior, cada uno de los campos corresponde al mismo empleado. Naturalmente, una compañía particular pudiera tener muchos empleados, y para cada uno de ellos tendrá un registro de nómina. Un *archivo* es un grupo de registros relacionados. El archivo de nóminas de una empresa normalmente contiene un registro para cada empleado. Entonces, un archivo de nóminas para una pequeña compañía pudiera contener únicamente 22 registros, en tanto que un archivo de nómina para una compañía grande pudiera contener 100,000 registros. No es desusado para una organización tener cientos e inclusive miles de archivos, muchos de ellos conteniendo millones y aún miles de millones de caracteres de información. Con la creciente popularidad de los discos laser ópticos y la tecnología de multimedios, pronto inclusive serán comunes archivos de billones de bytes. En la figura 11.1 se ilustra la jerarquía de los datos.

Para facilitar la recuperación de registros específicos a partir de un archivo, por lo menos un campo de cada registro es seleccionado como *registro clave.* Un registro clave identifica a un registro como perteneciente a una persona o entidad en particular. Por ejemplo, en el registro de nóminas descrito en esta sección, el número de seguridad social normalmente sería seleccionado como registro clave.

Existen muchas formas de organizar los registros dentro de un archivo. El tipo más popular de organización se conoce como *archivo secuencial,* en el cual típicamente los registros se almacenan en orden, en relación con el campo de registro clave. En un archivo de nóminas, los

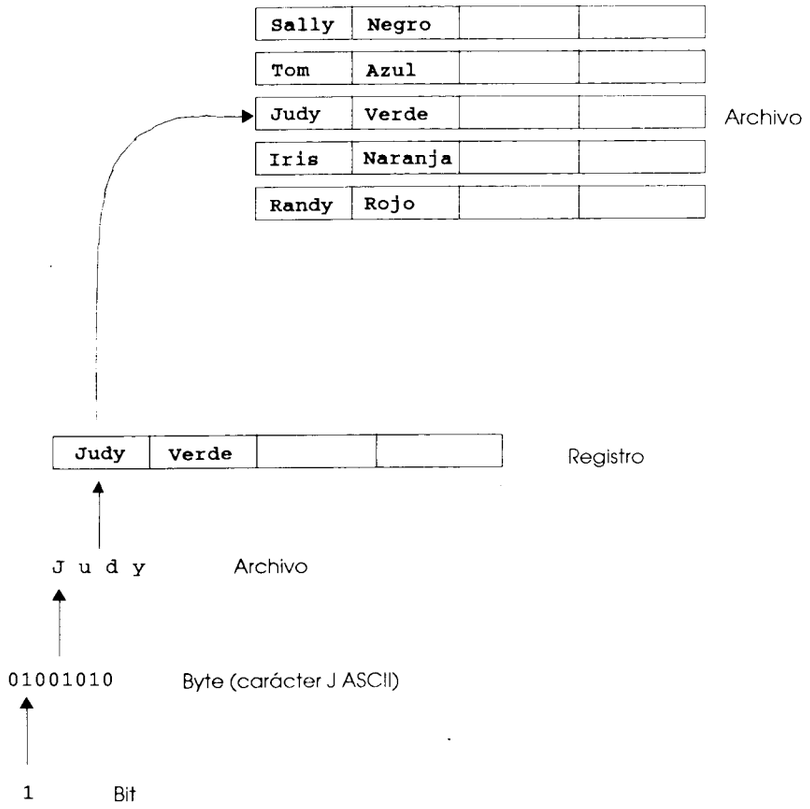


Fig. 11.1 La jerarquía de datos.

registros normalmente se ordenan por número de seguridad social. El primer registro de empleado del archivo contiene el número más bajo de seguridad social, y los registros subsiguientes contienen números de seguridad social cada vez más altos.

Para almacenar datos la mayor parte de los negocios utilizan muchos archivos distintos. Por ejemplo, las empresas pueden tener archivos de nóminas, archivos de cuentas por cobrar (enlistando el dinero que los clientes les deben), archivos de cuentas por pagar (enlistando dinero que se les debe a los proveedores), archivos de inventarios (enlistando hechos relacionados con todos los elementos manejados por el negocio) y muchos otros tipos de archivos. A veces un grupo de archivos relacionados se conoce como una *base de datos*. Una colección de programas diseñado para crear y administrar bases de datos se conoce como un *sistema de administración de bases de datos* (DBMS, por *database management system*).

11.3 Archivos y flujos

C ve cada uno de los archivos simplemente como un flujo secuencial de bytes (figura 11.2). Cada archivo termina con un *marcador de fin de archivo* o en un número de bytes específico registrado en una estructura administrativa de datos, mantenida por el sistema. Cuando un archivo *se abre*, se asocia un flujo con el archivo. Al empezar la ejecución de un programa automáticamente se

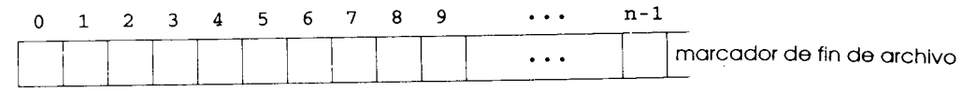


Fig. 11.2 Vista en C de un archivo de *n* bytes.

abren tres archivos y sus flujos asociados —la *entrada estándar*, la *salida estándar* y el *error estándar*. Los flujos proporcionan canales de comunicación entre archivos y programas. Por ejemplo, el flujo de entrada estándar permite que un programa lea datos del teclado, el flujo de salida estándar permite que un programa imprima datos a la pantalla. Abrir un archivo regresa un apuntador a una estructura **FILE** (definida en **<stdio.h>**) que contiene información utlizada para procesar dicho archivo. Esta estructura incluye un *descriptor de archivo*, es decir un índice a un arreglo del sistema operativo, conocido como una *tabla de archivo abierto*. Cada elemento del arreglo contiene un *bloque de control de archivo* (*FCB*, por *file control block*) utilizado por el sistema operativo para administrar el archivo particular. La entrada estándar, salida estándar y error estándar son manejados utilizando los apuntadores de archivo **stdin**, **stdout** y **stderr**.

La biblioteca estándar proporciona muchas funciones para leer datos de los archivos y para escribir datos a los archivos. La función **fgetc**, al igual que **getchar**, lee un carácter de un archivo. La función **fgetc** recibe como un argumento un apuntador **FILE** para el archivo del cual se leerá un carácter. La llamada **fgetc(stdin)** lee un carácter de **stdin** —la entrada estándar. Esta llamada es equivalente a la llamada **getchar()**. La función **fputc**, al igual que **putchar**, escribe un carácter a un archivo. La función **fputc** recibe como argumentos un carácter para ser escrito, y un apuntador al archivo hacia el cual el carácter será escrito. La llamada de función **fputc('a', stdout)** escribe el carácter **'a'**, a **stdout** —la salida estándar. Esta llamada es equivalente a **putchar('a')**.

Varias otras funciones, utilizadas para leer datos de la entrada estándar y para escribir datos a la salida estándar, tienen funciones de procesamiento de archivo similarmente identificados. Las funciones **fgets** y **fputs**, por ejemplo, pueden ser utilizadas para leer una línea de un archivo y para escribir una línea a un archivo, respectivamente. Sus contrapartidas para leer de la estrada estándar y para escribir a la salida estándar, **gets** y **puts**, ya fueron analizadas en el capítulo 8. En varias de las siguientes secciones, presentamos los equivalentes en procesamiento de archivo de las funciones **scanf** y **printf**—**fscanf** y **fprintf**. Más adelante en el capítulo analizaremos las funciones **fread** y **fwrite**.

11.4 Cómo crear un archivo de acceso secuencial

C no impone estructuras a un archivo. Por lo tanto, como parte del lenguaje C no existen conceptos como registro de un archivo. Por lo tanto, para que cumpla con los requisitos de cada aplicación en particular, el programador deberá proporcionar alguna estructura de archivo. En el ejemplo siguiente, vemos como un programador puede imponer una estructura de registro en un archivo.

El programa de la figura 11.3 crea un archivo simple de acceso secuencial, que podría ser utilizado en cualquier sistema de cuentas por cobrar para ayudar a llevar control de las cantidades que deben los clientes a crédito de una empresa. Para cada cliente, el programa obtiene un número de cuenta, el nombre del cliente y el saldo del mismo (es decir, la cantidad que el cliente le debe a la empresa debido a bienes y servicios recibidos en el pasado). Los datos obtenidos de cada cliente constituyen un "registro" para cada uno de estos clientes. En esta aplicación el número de cuenta se utiliza como registro clave —el archivo será creado y mantenido por orden de número

```
/* Create a sequential file */
#include <stdio.h>

main()
{
   int account;
   char name[30];
   float balance;
   FILE *cfPtr;   /* cfPtr = clients.dat file pointer */

   if ((cfPtr = fopen("clients.dat", "w")) == NULL)
      printf("File could not be opened\n");
   else {
      printf("Enter the account, name, and balance.\n");
      printf("Enter EOF to end input.\n");
      printf("? ");
      scanf("%d%s%f", &account, name, &balance);

      while (!feof(stdin)) {
         fprintf(cfPtr, "%d %s %.2f\n",
                 account, name, balance);
         printf("? ");
         scanf("%d%s%f", &account, name, &balance);
      }

      fclose(cfPtr);
   }

   return 0;
}
```

```
Enter the account, name, and balance.
Enter the EOF character to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
?
```

Fig. 11.3 Cómo crear un archivo secuencial.

de cuenta. Este programa supone que el usuario escribe los registros en orden por número de cuenta. En un sistema de cuentas por cobrar más completo, se incluiría una capacidad de ordenamiento, de tal forma que el usuario pudiera introducir los registros en cualquier orden. Los registros a continuación serían clasificados y escritos en el archivo.

Examinemos ahora este programa. El enunciado

```
FILE *cfPtr;
```

establece que **cfPtr** es un apuntador a una estructura **FILE**. El programa C administra cada archivo con una estructura **FILE** por separado. Para utilizar archivos el programador no necesita

saber los detalles específicos de la estructura **FILE**. Pronto veremos precisamente cómo la estructura **FILE** lleva indirectamente al bloque de control de archivos del sistema operativo (FCB) correspondiente a un archivo.

Sugerencia de portabilidad 11.1

La estructura **FILE** *depende del sistema operativo (es decir, los miembros de la estructura varían de un sistema a otro, según la forma en que cada sistema maneja sus archivos).*

Cada archivo abierto debe tener un apuntador declarado por separado del tipo **FILE**, que es utilizado para referirse al archivo. La línea

```
if ((cfPtr = fopen("clients.dat", "w")) == NULL)
```

nombra el archivo —**"clients.dat"**— para ser utilizado por el programa y establece una "línea de comunicación" con el archivo. El apuntador de archivo **cfPtr** es asignado a un apuntador a la estructura **FILE** para el archivo abierto con **fopen**. La función **fopen** toma dos argumentos: un nombre de archivo y un *modo de archivo abierto*. El modo de archivo abierto **"w"** indica que el archivo debe de ser abierto para *escritura*. Si el archivo no existe y es abierto para escritura, **fopen** crea el archivo. Si un archivo existente es abierto para escritura, el contenido del archivo es descartado sin advertencia. En el programa, la estructura **if** se utiliza para determinar si el apuntador al archivo **cfPtr** es **NULL** (es decir, el archivo no está abierto). Si es **NULL**, se imprime un mensaje de error y el programa termina. De lo contrario, la entrada es procesada y escrita al archivo.

Error común de programación 11.1

*Abrir un archivo existente para escritura (***"w"***) cuando, de hecho, el usuario desea conservar el archivo; el contenido del archivo se descartará sin advertencia.*

Error común de programación 11.2

Olvidar abrir un archivo antes de intentar hacer referencia a él en un programa.

El programa le solicita al usuario que introduzca los varios campos de cada registro, o que introduzca fin de archivo cuando esté completa la entrada de datos. La figura 11.4 enlista las combinaciones de teclas para introducir fin de archivo en varios sistemas de computación.

La línea

```
while (!feof(stdin))
```

utiliza la función **feof** para determinar si el *indicador de fin de archivo* está definido para el archivo al que se refiere **stdin**. El indicador de fin de archivo le informa al programa que ya no hay más datos a procesarse. En el programa de la figura 11.3, el indicador de fin de archivo está

Sistema de computación	Combinación de teclas
Sistemas UNIX	*<return><ctrl>d*
IBM PC y compatibles	*<ctrl>z*
Macintosh	*<ctrl>d*
VAX (VMS)	*<ctrl>z*

Fig. 11.4 Combinaciones de teclas de fin de archivo correspondientes a varios sistemas populares de computación.

definido para la entrada estándar cuando el usuario introduce la combinación de teclas de fin de archivo. El argumento de la función **feof** es un apuntador al archivo bajo prueba en relación con el indicador de fin de archivo (en este caso **stdin**). La función regresa un valor no cero (verdadera) una vez definido el indicador de fin de archivo; de lo contrario regresa cero. La estructura **while**, que en este programa incluye la llamada **feof**, se continuará ejecutando en tanto no se defina el indicador de fin de archivo.

El enunciado

```
fprintf(cfPtr, "%d %s %.2f\n", account, name, balance);
```

escribe datos al archivo **clients.dat**. Los datos pueden ser recuperados más tarde mediante un programa diseñado para leer el archivo (vea la sección 11.5). La función **fprintf** es equivalente a **printf**, excepto que **fprintf** también recibe como argumento un apuntador de archivo para el archivo al cual se escribirán los datos.

Error común de programación 11.3

Usar el apuntador de archivo incorrecto para referirse a un archivo.

Práctica sana de programación 11.1

Asegúrese que en un programa las llamadas a las funciones de procesamiento de archivos contienen los apuntadores de archivo correctos.

Después de que el usuario haya introducido el fin de archivo, el programa cierra el archivo **clients.dat** utilizando **fclose** y termina. La función **fclose** también recibe el apuntador de archivo (en vez del nombre del archivo) como un argumento. Si no se llama a la función **fclose** en forma explícita, normalmente cuando la ejecución del programa termine el sistema operativo cerrará el archivo. Esto es un ejemplo de "buena administración interna" del sistema operativo.

Práctica sana de programación 11.2

Cierre cada archivo en forma explícita, tan pronto sepa que el programa ya no hará otra vez referencia al archivo.

Sugerencia de rendimiento 11.1

Cerrar un archivo puede liberar recursos que están siendo esperados por otros usuarios o programas.

En la ejecución de muestra del programa de la figura 11.3, el usuario introduce información correspondiente a cinco cuentas, y a continuación escribe el fin de archivo, para indicar que la entrada de datos está completa. La ejecución de muestra no pone de manifiesto cómo aparecen realmente los registros de datos dentro del archivo. A fin de verificar que el archivo haya sido creado exitosamente, en la siguiente sección introducimos un programa que lee el archivo e imprime su contenido.

En la figura 11.5 se ilustra la relación entre los apuntadores **FILE**, las estructuras **FILE** y los FCB en memoria. Cuando se abre el archivo **"clients.dat"**, se copia un FCB para dicho archivo en la memoria. La figura muestra la conexión entre el apuntador de archivo regresado por **fopen** y el FCB utilizado por el sistema operativo para la administración del archivo.

Los programas no pueden procesar ningún archivo, un archivo o varios archivos. Cada archivo utilizado en un programa deberá tener un nombre único, y deberá tener un apuntador de archivo distinto regresado por **fopen**. Todas las funciones de procesamiento de archivos subsecuentes después de su apertura deberán referirse al archivo utilizando el apuntador de archivo apropiado.

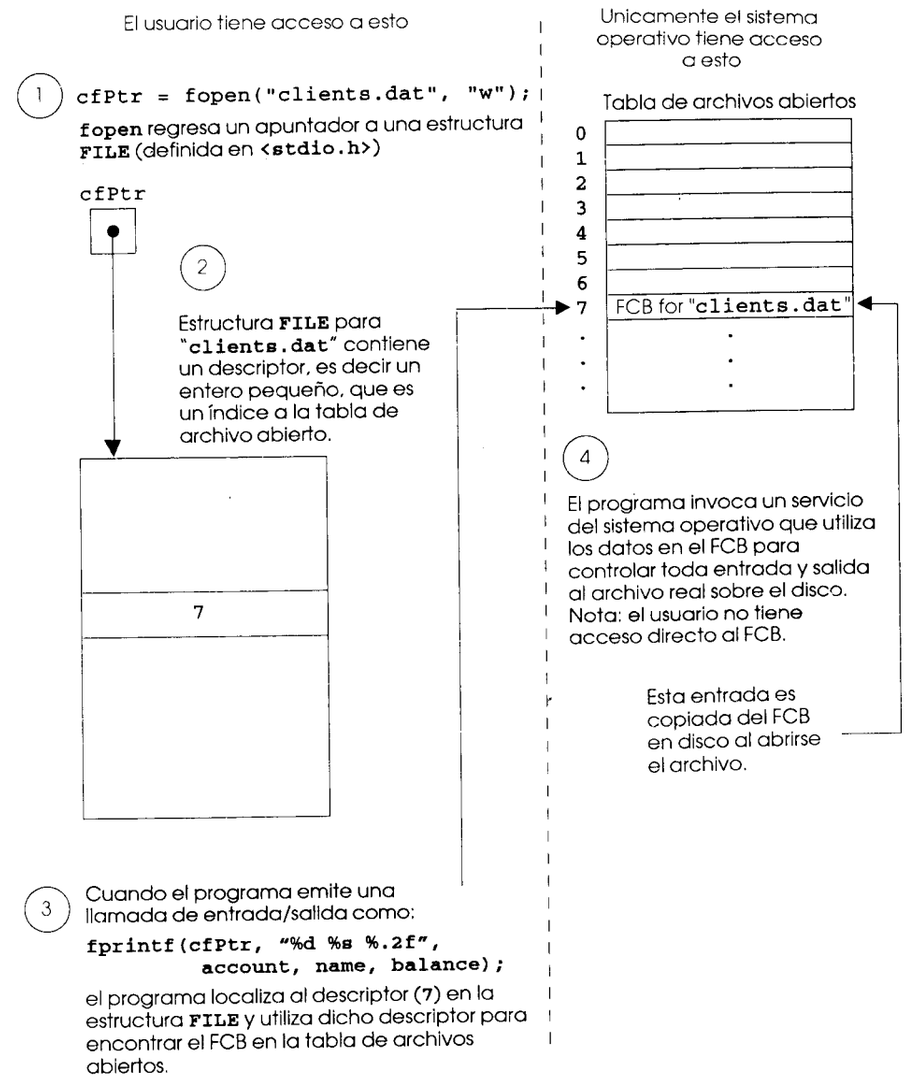


Fig. 11.5 Relación entre los apuntadores **FILE** las estructuras **FILE** y los FCB.

Los archivos pueden ser abiertos en uno de varios modos. Para crear un archivo, o para descartar el contenido de un archivo antes de escribir datos, abra el archivo para escritura (**"w"**). Para leer un archivo existente, ábralo para lectura (**"r"**). Para añadir registros al final de un archivo existente, abra el archivo para agregar (**"a"**). Para abrir a un archivo de tal forma que pueda ser escrito y leído, abra el archivo para actualizar en uno de los tres modos de actualización —**"r+"**, **"w+"** o **"a+"**. El modo **"r+"** abre un archivo para lectura y escritura. El modo **"w+"** genera un

archivo para lectura y escritura. Si el archivo ya existe, el archivo es abierto y el contenido actual de dicho archivo se descarta. El modo **"a+"** abre un archivo para lectura y escritura —toda escritura se efectuará al final del archivo. Si el archivo no existe será creado.

Si al abrir un archivo en cualquiera de los modos anteriores ocurre un error, **fopen** regresará **NULL**. Algunos errores posibles son:

Error común de programación 11.4

Abrir para lectura un archivo no existente.

Error común de programación 11.5

Abrir un archivo para lectura o escritura sin haber obtenido los derechos de acceso al archivo apropiados (esto depende del sistema operativo).

Error común de programación 11.6

Abrir un archivo para escritura cuando no hay espacio disponible en disco. En la figura 11.6 se enlistan los modos de apertura de archivos.

Error común de programación 11.7

Puede llevar a errores devastadores abrir un archivo utilizando el modo de archivo incorrecto. Por ejemplo, abrir un archivo en el modo de escribir ("w") cuando debería haberse abierto en modo de actualizar ("r+") hace que sea descartado todo el contenido del archivo.

Práctica sana de programación 11.3

Abra un archivo únicamente para lectura (y no para actualizar), si el contenido del archivo no debe modificarse. Esto evitará cambios no intencionales en el contenido del archivo. Este es otro ejemplo del principio del mínimo privilegio.

11.5 Cómo leer datos de un archivo de acceso secuencial

Los datos se almacenan en archivos, de tal forma que cuando sea necesario puedan ser recuperados para su proceso. La sección anterior demostró cómo crear un archivo para acceso secuencial. En esta sección, analizamos como leer secuencialmente los datos de un archivo.

Modo	Descripcion
r	Abrir una archivo para lectura.
w	Crear un archivo para escritura. Si el archivo ya existe, se descarta el contenido actual.
a	Agregar; abrir o crear un archivo para escribir al final del mismo.
r+	Abrir un archivo para actualizar (leer y escribir).
w+	Crear un archivo para actualizar. Si el archivo ya existe, se descarta el contenido actual.
a+	Agregar; abrir o crear un archivo para actualizar; la escritura se efectuará al final del archivo.

Fig. 11.6 Modos de apertura de archivo.

El programa de la figura 11.7 lee registros del archivo **"clients.dat"** creados por el programa de la figura 11.3 e imprime el contenido de los registros. El enunciado

```
FILE *cfPtr;
```

indica que **cfPtr** es un apuntador a un **FILE**. La línea

```
if ((cfPtr = fopen("clients.dat", "r")) ==NULL)
```

intenta abrir el archivo **"clients.dat"** para lectura (**'r'**) y determina si el archivo se ha abierto con éxito (es decir, que **fopen** no regresa **NULL**). El enunciado

```
fscanf(cPtr, "%d%s%f, &account, name, &balance);
```

lee un "registro" del archivo. La función **fscanf** es equivalente a la función **scanf**, salvo que **fscanf** recibe como argumento un apuntador a un archivo para el archivo del cual se van a leer datos. Después de que el enunciado anterior es ejecutado por primera vez, **account** tendrá el valor **100**, **name** tendrá el valor **"Jones"**, y **balance** tendrá el valor **24.98**. Cada vez que se ejecute el segundo enunciado **fscanf**, se leerá otro registro del archivo y **account**, **name** y **balance** tomarán nuevos valores. Cuando se llegue al final del archivo, éste se cerrará y el programa terminará.

```
/* Reading and printing a sequential file */
#include <stdio.h>

main()
{
   int account;
   char name[30];
   float balance;
   FILE *cfPtr;    /* cfPtr = clients.dat file pointer */

   if ((cfPtr = fopen("clients.dat", "r")) == NULL)
      printf("File could not be opened\n");
   else {
      printf("%-10s%-13s%s\n", "Account", "Name", "Balance");
      fscanf(cfPtr, "%d%s%f", &account, name, &balance);

      while (!feof(cfPtr)) {
         printf("%-10d%-13s%7.2f\n", account, name, balance);
         fscanf(cfPtr, "%d%s%f", &account, name, &balance);
      }

      fclose(cfPtr);
   }

   return 0;
}
```

Fig. 11.7 Cómo leer e imprimir un archivo secuencial (parte 1 de 2).

```
Account  Name      Balance
100      Jones       24.98
200      Doe        345.67
300      White        0.00
400      Stone      -42.16
500      Rich       224.62
```

Fig. 11.7 Cómo leer e imprimir un archivo secuencial (parte 2 de 2).

Para recuperar secuencialmente datos de un archivo, un programa normalmente empieza a leer a partir del principio del archivo, y lee todos los datos en forma consecutiva, hasta que encuentra los datos deseados. Durante la ejecución de un programa pudiera ser deseable procesar los datos secuencialmente en un archivo varias veces (a partir del principio del archivo). Un enunciado como

```
rewind(cfPtr);
```

genera un *apuntador de posición de archivo* del programa —que indica el número del siguiente byte del archivo a leerse o a escribirse— a que se recoloque al principio del archivo (es decir en el byte 0) al cual apunta **cfPtr**. El apuntador de posición de archivo no es realmente un apuntador. Más bien es un valor entero que especifica la posición de byte en el archivo, en el cual ocurrirá la siguiente lectura o escritura. Esto a veces se denomina el *desplazamiento de archivo*. El apuntador de posición de archivo es un miembro de la estructura **FILE** asociado con cada archivo.

Ahora introducimos un programa (figura 11.8) que le permite a un gerente de crédito obtener listas de clientes con saldo 0 (es decir, clientes que no deben ningún dinero), clientes con saldos acreedores (es decir, clientes a los cuales la empresa les debe dinero), clientes con saldos deudores (es decir, clientes que le deben dinero a la empresa por bienes y servicios recibidos). Un saldo acreedor es una cantidad negativa; un saldo deudor es una cantidad positiva.

El programa despliega un menú y le permite al gerente de crédito introducir una de tres opciones para obtener información de crédito. La opción 1 produce una lista de cuentas con saldos en cero. La opción 2 produce una lista de cuentas con saldos acreedores. La opción 3 produce una lista de cuentas con saldos deudores. La opción 4 termina la ejecución del programa. Una salida de muestra se despliega en la figura 11.9.

Note que los datos en este tipo de archivo secuencial no pueden ser modificados sin riesgo de destruir otros datos dentro del archivo. Por ejemplo, si el nombre "**White**" necesitara ser modificado a "**Worthington**", el nombre antiguo simplemente no puede ser sobrescrito. El registro para **White** fue escrito al archivo como

```
300 White 0.00
```

Si el registro se reescribe utilizando el nuevo nombre empezando en la misma posición en el archivo, el registro sería

```
300 Worthington 0.00
```

El nuevo registro es más largo que el original. Los caracteres más allá de la segunda "**o**" de "**Worthington**" sobreescribirían el principio del siguiente registro secuencial en el archivo. El


```
/* Credit inquiry program */
#include <stdio.h>

main()
{
   int request, account;
   float balance;
   char name[30];
   FILE *cfPtr;

   if ((cfPtr = fopen("clients.dat", "r")) == NULL)
      printf("File could not be opened\n");
   else {
      printf("Enter request\n"
             " 1 - List accounts with zero balances\n"
             " 2 - List accounts with credit balances\n"
             " 3 - List accounts with debit balances\n"
             " 4 - End of run\n? ");
      scanf("%d", &request);

      while (request != 4) {
         fscanf(cfPtr, "%d%s%f", &account, name, &balance);

         switch (request) {
            case 1:
               printf("\nAccounts with zero balances:\n");

               while (!feof(cfPtr)) {

                  if (balance == 0)
                     printf("%-10d%-13s%7.2f\n",
                            account, name, balance);

                  fscanf(cfPtr, "%d%s%f",
                         &account, name, &balance);
               }

               break;
            case 2:
               printf("\nAccounts with credit balances:\n");

               while (!feof(cfPtr)) {

                  if (balance < 0)
                     printf("%-10d%-13s%7.2f\n",
                            account, name, balance);

                  fscanf(cfPtr, "%d%s%f",
                         &account, name, &balance);
               }

               break;
```

Fig. 11.8 Programa de consulta de crédito (parte 1 de 2).

```
            case 3:
               printf("\nAccounts with debit balances:\n");

               while (!feof(cfPtr)) {

                  if (balance > 0)
                     printf("%-10d%-13s%7.2f\n",
                            account, name, balance);

                  fscanf(cfPtr, "%d%s%f",
                         &account, name, &balance);
               }

               break;
         }

         rewind(cfPtr);
         printf("\n? ");
         scanf("%d", &request);
      }

      printf("End of run.\n");
      fclose(cfPtr);
   }

   return 0;
}
```

Fig. 11.8 Programa de consulta de crédito (parte 2 de 2).

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 1

Accounts with zero balances:
300        White                0.00

? 2

Accounts with credit balances:
400        Stone              -42.16

? 3

Accounts with debit balances:
100        Jones               24.98
200        Doe                345.67
500        Rich               224.62

? 4
End of run.
```

Fig. 11.9 Salida de muestra del programa de consulta de crédito de la figura 11.8.

problema aquí es que en el modelo de entrada/salida con formato utilizando **`fprintf`** y **`fscanf`**, los campos —y por lo tanto los registros— pueden variar de tamaño. Por ejemplo, 7, 14, -117, 2074, y 27383 son todos **`int`** almacenados internamente en el mismo número de bytes, pero imprimen en la pantalla o **`fprintf`** en el disco como campos de tamaño diferente.

Por lo tanto, el acceso secuencial mediante **`fprint`** y **`fscanf`** normalmente no se utiliza para actualizar registros en su sitio. En vez de ello, usualmente la totalidad del archivo se vuelve a escribir. Para llevar a cabo la modificación de nombre anteriormente citada, los registros anteriores a **`300 White 0.00`** en un archivo como éste, de acceso secuencial, serían copiados a un nuevo archivo, sería escrito el nuevo registro, y los registros existentes después de **`300 White 0.00`** serían vueltos a copiar al nuevo archivo. Esto significa el procesamiento de todos los registros en el archivo para simplemente actualizar un registro.

11.6 Archivos de acceso directo

Como hemos indicado anteriormente, los registros en un archivo creados con la función de salida **`fprintf`** con formato, no necesariamente son de la misma longitud. Sin embargo, los registros individuales de un *archivo de acceso directo* normalmente son de longitud fija y se puede tener acceso a ellos directamente (y por lo tanto rápidamente) sin tener que buscar a través de otros registros. Esto hace que los archivos de acceso directo sean apropiados para sistemas de reservación de aerolíneas, sistemas bancarios, sistemas de punto de venta y otros tipos de *sistemas de procesamiento de transacciones,* que requieren de acceso rápido a datos específicos. Existen otras formas para poner en funcionamiento archivos de acceso directo, pero limitaremos nuestro análisis a este enfoque sencillo de utilización de registros de longitud fija.

Dado que en un archivo de acceso directo todos los registros normalmente tienen la misma longitud, la posición exacta de un registro en relación con el principio del archivo puede ser calculada como una función del registro clave. Pronto veremos como esto facilita el acceso inmediato a registros específicos, inclusive en archivos grandes.

En la figura 11.10 se ilustra una forma para poner en marcha un archivo de acceso directo. Un archivo como éste es similar a un tren de carga con muchos vagones —algunos vacíos y algunos con carga. Cada vagón en el tren tiene la misma longitud.

Los datos pueden ser insertos en un archivo de acceso directo sin destruir otros datos en el archivo. Los datos almacenados anteriormente también pueden ser actualizados o borrados, sin tener que reescribir todo el archivo. En las secciones que siguen explicaremos como se crea un archivo de acceso directo, como se introduce información, como se leen los datos tanto secuencial como directamente, como se actualizan los datos y como se borran aquellos datos que ya no se requieran.

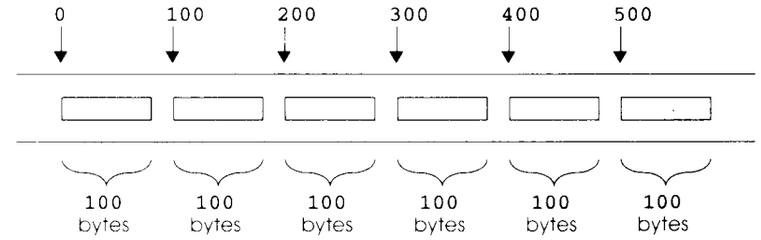


Fig. 11.10 Vista de un archivo de acceso directo con registros de longitud fija.

11.7 Cómo crear un archivo de acceso directo

La función **fwrite** transfiere a un archivo un número especificado de bytes empezando en una posición especificada de memoria. Los datos se escriben al principio de la posición en el archivo indicada mediante el apuntador de posición de archivo. La función **fread** transfiere un número especificado de bytes de la posición en el archivo, especificado por el apuntador de posición de archivo, a un área en memoria empezando a partir de una dirección especificada. Ahora, al escribir un entero, en vez de utilizar

```
fprintf(fPtr, "%d", number);
```

podría imprimir desde 1 dígito hasta un máximo de 11 dígitos (10 dígitos más un signo, cada uno de los cuales requiere 1 byte de almacenamiento) para un entero de 4 bytes, podemos utilizar

```
fwrite(&number, sizeof(int), 1, fPtr);
```

que siempre escribirá 4 bytes (o 2 bytes en un sistema con enteros de 2 bytes) de la variable **number** al archivo representado por **fPtr** (explicaremos en breve el argumento **1**). Más adelante, **fread** podrá ser utilizado para leer 4 de estos bytes a la variable entera **number**. Aunque **fread** y **fwrite** leen y escriben datos, como son enteros, en tamaño fijo en vez de en formato de tamaño variable, los datos que manejan se procesan en formato "en bruto" de computadora (es decir, bytes de datos), en vez de en el formato legible para los seres humanos **printf** y **scanf**.

Las funciones **fwrite** y **fread** son capaces de leer y de escribir arreglos de datos hacia y desde el disco. El tercer argumento, tanto de **fread** como **fwrite** es el número de elementos en el arreglo que deberá ser leído del disco, o escrito al disco. La llamada anterior de función **fwrite** escribe un solo entero al disco, por lo que el tercer argumento es **1** (como si un elemento de un arreglo fuera escrito).

Los programas de procesamiento de archivos rara vez escriben un solo campo a un archivo. Normalmente, escriben un **struct** a la vez, como veremos en los ejemplos siguientes.

Considere el siguiente enunciado de problema:

Crear un sistema de procesamiento de crédito capaz de almacenar hasta 100 registros de longitud fija. Cada registro deberá estar formado de un número de cuenta, que será utilizado como registro clave, un apellido, un nombre y un saldo. El programa resultante deberá ser capaz de actualizar una cuenta, insertar un nuevo registro de cuenta, borrar una cuenta y enlistar todos los registros de cuenta en un archivo de texto con formato para su impresión. Utilice un archivo de acceso directo.

En las siguientes varias secciones se introducen las técnicas necesarias para crear el programa de procesamiento de crédito. El programa de la figura 11.11 muestra como abrir un archivo de acceso directo, definir un formato de registro utilizando un **struct**, escribir datos al disco, y cerrar el archivo. Este programa inicializa los 100 registros del archivo **"credit.dat"**, con **structs** vacíos utilizando la función **fwrite**. Cada **struct** vacío contiene **0** para el número de cuenta, **NULL** (representado por comillas vacías) para el apellido, **NULL** para el nombre y **0.0** para el saldo. El archivo se inicializa de esta forma para crear espacio en el disco en el cual se almacenará el archivo, y para que sea posible determinar si un registro contiene datos.

La función **fwrite** escribe un bloque (un número específico de bytes) de datos a un archivo. En nuestro programa, el enunciado

```
fwrite(&blankClient, sizeof(struct clientData), 1, cfPtr);
```



```
/* Creating a randomly accessed file sequentially */
#include <stdio.h>

struct clientData {
   int acctNum;
   char lastName[15];
   char firstName[10];
   float balance;
};

main()
{
   int i;
   struct clientData blankClient = {0, "", "", 0.0};
   FILE *cfPtr;

   if ((cfPtr = fopen("credit.dat", "w")) == NULL)
      printf("File could not be opened.\n");
   else {

      for (i = 1; i <= 100; i++)
         fwrite(&blankClient,
                sizeof(struct clientData), 1, cfPtr);

      fclose (cfPtr);
   }

   return 0;
}
```

Fig. 11.11 Cómo crear un archivo de acceso directo en forma secuencial.

hace que la estructura **blankClient** de tamaño **sizeof(struct clientData)** se escriba al archivo al cual apunta **cfPtr**. El operador **sizeof** regresa el tamaño en bytes del objeto contenido en los paréntesis (en este caso **struct clientData**). El operador **sizeof** es un operador unario en tiempo de compilación que regresa un entero no signado. El operador **sizeof** puede ser utilizado para determinar el tamaño en bytes de cualquier tipo o expresión de datos. Por ejemplo, **sizeof(int)** se utiliza para determinar si en una computadora en particular un entero está almacenado en 2 o en 4 bytes.

Sugerencia de rendimiento 11.2

*Muchos programadores piensan erróneamente que **sizeof** es una función, y que utilizándolo se genera una sobrecarga en tiempo de ejecución de una llamada de función. No existe tal sobrecarga, porque **sizeof** es un operador en tiempo de compilación.*

La función **fwrite** puede de hecho ser utilizada para escribir varios elementos de un arreglo de objetos. Para escribir varios elementos de arreglo, el programador proporciona un apuntador a un arreglo como primer argumento en la llamada **fwrite**, y especifica el número de elementos a escribirse como el tercer argumento en la llamada **fwrite**. En el enunciado anterior, **fwrite** fue utilizado para escribir un solo objeto que no era un elemento de arreglo. Escribir un solo objeto es el equivalente a escribir un elemento de un arreglo, de ahí el **1** en la llamada **fwrite**.

11.8 Cómo escribir datos directamente a un archivo de acceso directo

El programa de la figura 11.2 escribe datos al archivo **"credit.dat"**. Utiliza la combinación de **fseek** y **fwrite** para almacenar datos en posiciones específicas dentro del archivo. La función **fseek** define el apuntador de posición de archivo a una posición específica dentro del archivo, y a continuación **fwrite** escribe los datos. Una ejecución de muestra aparece en la figura 11.13.

El enunciado

```
fseek(cfPtr, (accountNum - 1) * sizeof(struct clientData);
      SEEK_SET);
```

```
/* Writing to a random access file */
#include <stdio.h>

struct clientData {
   int acctNum;
   char lastName[15];
   char firstName[10];
   float balance;
};

main()
{
   FILE *cfPtr;
   struct clientData client;

   if ((cfPtr = fopen("credit.dat", "r+")) == NULL)
      printf("File could not be opened.\n");
   else {
      printf("Enter account number"
             " (1 to 100, 0 to end input)\n? ");
      scanf("%d", &client.acctNum);

      while (client.acctNum != 0) {
         printf("Enter lastname, firstname, balance\n? ");
         scanf("%s%s%f", &client.lastName,
               &client.firstName, &client.balance);
         fseek(cfPtr, (client.acctNum - 1) *
               sizeof(struct clientData), SEEK_SET);
         fwrite(&client, sizeof(struct clientData), 1, cfPtr);
         printf("Enter account number\n? ");
         scanf("%d", &client.acctNum);
      }
   }

   fclose(cfPtr);

   return 0;
}
```

Fig. 11.12 Cómo escribir datos directamente a un archivo de acceso directo.


```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0
```

Fig. 11.13 Ejecución de muestra del programa de la figura 11.12.

posiciona el apuntador de posición de archivo para el archivo referenciado por **cfPtr**, a la posición de bytes calculada por **(accountNum - 1) * sizeof(struct clientData)**; el valor de esta expresión se conoce como el *desplazamiento*. Dado que el número de cuenta está entre 1 y 100, pero las posiciones de bytes en el archivo empiezan con **0**, al calcular la posición de bytes dentro del registro se resta 1 del número de cuenta. Entonces, para el registro 1, el apuntador de posición de archivo se define al byte 0 del archivo. La constante simbólica **SEEK_SET** indica que el apuntador de posición de archivo está colocado en relación con el principio del archivo, en la cantidad del desplazamiento. Como indica el enunciado anterior, una búsqueda para el número de cuenta 1 en el archivo define el apuntador de posición de archivo al principio del archivo, porque la posición de bytes calculada es 0. La figura 11.14 ilustra el apuntador de archivo que hace referencia a una estructura **FILE** en memoria. El apuntador de posición de archivo indica que el siguiente byte a leerse o escribirse está a 5 bytes del principio del archivo.

El estándar ANSI muestra la función prototipo para **fseek** como

```
int fseek(FILE *stream, long int offset, int whence);
```

donde **offset** es el número de bytes a partir de la posición **whence** en el archivo al cual apunta **stream**. El argumento **whence** puede tener uno de tres valores —**SEEK_SET**, **SEEK_CUR** o **SEEK_END**— indicando la posición en el archivo a partir del cual se inicia la búsqueda. **SEEK_SET** indica que la búsqueda se inicia al principio del archivo; **SEEK_CUR** indica que la búsqueda se inicia en la posición actual en el archivo; y **SEEK_END** indica que la búsqueda se inicia en el final del archivo. Estas tres constantes simbólicas se definen en el archivo de cabecera **stdio.h**.

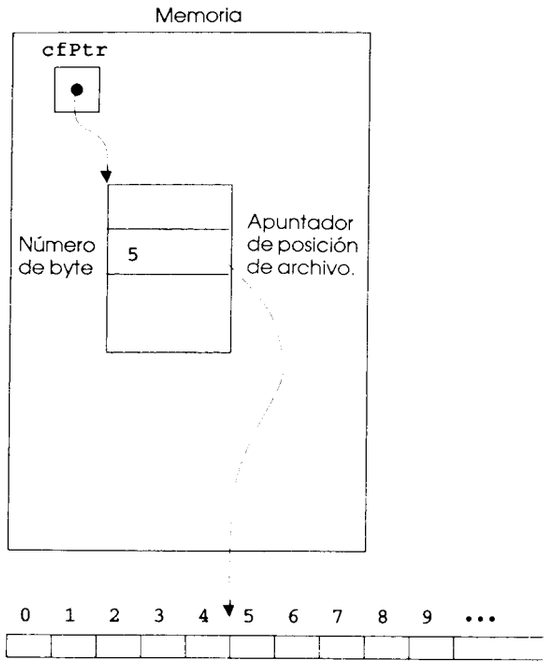


Fig. 11.14 Apuntador de posición de archivo, indicando un desplazamiento de 5 bytes a partir del principio del archivo.

11.9 Cómo leer datos directamente de un archivo de acceso directo

La función **fread** lee un número especificado de bytes de un archivo a la memoria. Por ejemplo, el enunciado

```
fread(&client, sizeof(struct clientData), 1, cfPtr);
```

lee el número de bytes determinado por **sizeof(struct clientData)** correspondiente al archivo referenciado por **cfPtr** y almacena el dato en la estructura **client**. Los bytes son leídos de la posición en el archivo especificado por el apuntador de posición de archivo. La función **fread** puede ser utilizada para leer varios elementos de arreglo de tamaño fijo, proporcionando un apuntador al arreglo en el cual los elementos se almacenarán, e indicando el número de elementos a leerse. El enunciado anterior especifica que un elemento deberá ser leído. Para poder leer más de un elemento, especifique el número de elementos en el tercer argumento del enunciado **fread**.

El programa de la figura 11.15 lee en forma secuencial todos los registros en el archivo **"credit.dat"**, determina si cada uno de dichos registros contiene datos, e imprime los datos con formato correspondientes a los registros que contengan datos. La función **feof** determina cuando se alcanza el fin de archivo, y la función **fread** transfiere los datos del disco a la estructura **client** de **clientData**.


```
/* Reading a random access file sequentially */
#include <stdio.h>

struct clientData {
   int acctNum;
   char lastName[15];
   char firstName[10];
   float balance;
};

main()
{
   FILE *cfPtr;
   struct clientData client;

   if ((cfPtr = fopen("credit.dat", "r")) == NULL)
      printf("File could not be opened.\n");
   else {
      printf("%-6s%-16s%-11s%10s\n", "Acct", "Last Name",
             "First Name", "Balance");

      while (!feof(cfPtr)) {
         fread(&client, sizeof(struct clientData), 1, cfPtr);

         if (client.acctNum != 0)
            printf("%-6d%-16s%-11s%10.2f\n",
                   client.acctNum, client.lastName,
                   client.firstName, client.balance);
      }
   }

   fclose(cfPtr);

   return 0;
}
```

```
Acct  Last Name      First Name  Balance
29    Brown          Nancy        -24.54
33    Dunn           Stacey       314.33
37    Barker         Doug           0.00
88    Smith          Dave         258.34
96    Stone          Sam           34.98
```

Fig. 11.5 Cómo leer secuencialmente un archivo de acceso directo.

11.10 Estudio de caso: un programa de procesamiento de transacciones

Ahora presentamos un programa sustancial de procesamiento de transacciones, que utiliza archivos de acceso directo. El programa mantiene información de cuentas de un banco. El programa actualiza las cuentas existentes, añade cuentas nuevas, borra cuentas y almacena un enlistado de todas las cuentas actuales, en un archivo de texto para su impresión. Suponemos que el programa de la figura 11.11 ha sido ejecutado para crear el archivo **credit.dat**.

El programa tiene cinco opciones. La opción 1 llama a la función **textFile** para almacenar una lista con formato de todas las cuentas en un archivo de texto llamado **accounts.txt**, que pudiera ser impreso más adelante. La función utiliza **fread** y las técnicas de acceso secuencial a archivo utilizadas en el programa de la figura 11.15. Después de seleccionar la opción 1 el archivo **accounts.txt** contiene:

```
Acct  Last Name      First Nam      Balance
29    Brown          Nancy           -24.54
33    Dunn           Stacey          314.33
37    Barker         Doug              0.00
88    Smith          Dave            258.34
96    Stone          Sam              34.98
```

La opción 2 llama a la función **updateRecord** para actualizar una cuenta. La función únicamente actualizará un registro ya existente, por lo que la función primero averiguará si el registro especificado por el usuario está vacío. El registro se lee a la estructura **client** utilizando a **fread**, y a continuación se utiliza **strcmp** para determinar si es **NULL** el miembro **lastName** de la estructura **client**. De ser así, el registro no contiene información, y se imprime un mensaje indicando que el registro está vacío. A continuación se despliegan las selecciones de menú. Si el registro contiene información, la función **updateRecord** introduce la cantidad de transacción, calcula el nuevo saldo, y vuelve a escribir el registro al archivo. Una salida típica correspondiente a la opción 2 es:

```
Enter account to update (1 - 100): 37
37  Barker          Doug            0.00

Enter charge (+) or payment (-): +87.99
37  Barker          Doug           87.99
```

La opción 3 llama a la función **newRecord** a fin de añadir una nueva cuenta al archivo. Si el usuario introduce un número de cuenta correspondiente a una cuenta existente, **newRecord** despliega un mensaje de error, indicando que el registro ya contiene información, y las selecciones de menú se vuelven a imprimir. Esta función utiliza para añadir una nueva cuenta el mismo proceso que en el programa de la figura 11.12. Una salida típica para la opción 3 es

```
Enter new account number (1 - 100): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

La opción 4 llama a la función **deleteRecord** para borrar un registro del archivo. El borrado se lleva a cabo solicitándole al usuario el número de cuenta y volviendo a inicializar el registro. Si la cuenta no contiene información, **deleteRecord** despliega un mensaje de error, indicando que la cuenta no existe. La opción 5 termina la ejecución del programa. El programa se muestra en la figura 11.16. Advierta que el archivo **"credit.dat"** se abre para actualizar (lectura y escritura) mediante el modo **"r+"**.


```
/* This program reads a random access file sequentially, *
 * updates data already written to the file, creates new *
 * data to be placed in the file, and deletes data       *
 * already in the file.                                  */

#include <stdio.h>

struct clientData {
   int acctNum;
   char lastName[15];
   char firstName[10];
   float balance;
};

int enterChoice(void);
void textFile(FILE *);
void updateRecord(FILE *);
void newRecord(FILE *);
void deleteRecord(FILE *);

main()
{
   FILE *cfPtr;
   int choice;

   if ((cfPtr = fopen("credit.dat", "r+")) == NULL)
      printf("File could not be opened.\n");
   else {

      while ((choice = enterChoice()) != 5) {

         switch (choice) {
            case 1:
               textFile(cfPtr);
               break;
            case 2:
               updateRecord(cfPtr);
               break;
            case 3:
               newRecord(cfPtr);
               break;
            case 4:
               deleteRecord(cfPtr);
               break;
         }

      }
   }

   fclose(cfPtr);
   return 0;
}
```

Fig. 11.16 Programa de cuentas de banco (parte 1 de 4).

```
void textFile(FILE *readPtr)
{
   FILE *writePtr;
   struct clientData client;

   if ((writePtr = fopen("accounts.txt", "w")) == NULL)
      printf("File could not be opened.\n");
   else {
      rewind(readPtr);
      fprintf(writePtr, "%-6s%-16s%-11s%10s\n",
              "Acct", "Last Name", "First Name","Balance");

      while (!feof(readPtr)) {
         fread(&client, sizeof(struct clientData), 1, readPtr);

         if (client.acctNum != 0)
            fprintf(writePtr, "%-6d%-16s%-11s%10.2f\n",
                    client.acctNum, client.lastName,
                    client.firstName, client.balance);
      }
   }

   fclose(writePtr);
}

void updateRecord(FILE *fPtr)
{
   int account;
   float transaction;
   struct clientData client;

   printf("Enter account to update (1 - 100): ");
   scanf("%d", &account);
   fseek(fPtr, (account - 1) * sizeof(struct clientData),
         SEEK_SET);
   fread(&client, sizeof(struct clientData), 1, fPtr);

   if (client.acctNum == 0)
      printf("Acount #%d has no information.\n", account);
   else {
      printf("%-6d%-16s%-11s%10.2f\n\n",
             client.acctNum, client.lastName,
             client.firstName, client.balance);
      printf("Enter charge (+) or payment (-): ");
      scanf("%f", &transaction);
      client.balance += transaction;
      printf("%-6d%-16s%-11s%10.2f\n",
             client.acctNum, client.lastName,
             client.firstName, client.balance);
      fseek(fPtr, (account - 1) * sizeof(struct clientData),
            SEEK_SET);
      fwrite(&client, sizeof(struct clientData), 1, fPtr);
   }
}
```

Fig. 11.16 Programa de cuentas de banco (parte 2 de 4).


```
void deleteRecord(FILE *fPtr)
{
   struct clientData client, blankClient = {0, "", "", 0};
   int accountNum;

   printf("Enter account number to delete (1 - 100): ");
   scanf("%d", &accountNum);
   fseek(fPtr, (accountNum - 1) * sizeof(struct clientData),
         SEEK_SET);
   fread(&client, sizeof(struct clientData), 1, fPtr);

   if (client.acctNum == 0)
      printf("Account %d does not exist.\n", accountNum);
   else {
      fseek(fPtr, (accountNum - 1) * sizeof(struct clientData),
            SEEK_SET);
      fwrite(&blankClient, sizeof(struct clientData), 1, fPtr);
   }
}

void newRecord(FILE *fPtr)
{
   struct clientData client;
   int accountNum;
   printf("Enter new account number (1 - 100): ");
   scanf("%d", &accountNum);
   fseek(fPtr, (accountNum - 1) * sizeof(struct clientData),
         SEEK_SET);
   fread(&client, sizeof(struct clientData), 1, fPtr);

   if (client.acctNum != 0)
      printf("Account #%d already contains information.\n",
             client.acctNum);
   else {
      printf("Enter lastname, firstname, balance\n? ");
      scanf("%s%s%f", &client.lastName, &client.firstName,
            &client.balance);
      client.acctNum = accountNum;
      fseek(fPtr, (client.acctNum - 1) *
            sizeof(struct clientData), SEEK_SET);
      fwrite(&client, sizeof(struct clientData), 1, fPtr);
   }
}
```

Fig. 11.16 Programa de cuentas de banco (parte 3 de 4).

Resumen

- Todos los elementos de datos procesados por una computadora se reducen a combinaciones de ceros y de unos.
- En una computadora el elemento más pequeño de datos puede asumir el valor `0` o el valor `1`. Este elemento de datos se conoce como un bit (abreviatura de "binary digit" —un dígito que puede asumir uno de dos valores).

```
int enterChoice(void)
{
   int menuChoice;

   printf("\nEnter your choice\n"
          "1 - store a formatted text file of acounts called\n"
          "    \"accounts.txt\" for printing\n"
          "2 - update an account\n"
          "3 - add a new account\n"
          "4 - delete an account\n"
          "5 - end program\n? ");
   scanf("%d", &menuChoice);
   return menuChoice;
}
```

Fig. 11.16 Programa de cuentas de banco (parte 4 de 4).

- Los dígitos, las letras y los símbolos especiales se conocen como caracteres. El conjunto de todos los caracteres que pueden ser utilizados para escribir programas y representar elementos de datos en una computadora en particular se conoce como el conjunto de caracteres de la computadora. Cada uno de los caracteres en el conjunto de caracteres de la computadora se representa como un patrón de ocho `1`s y `0`s (conocido como un byte).
- Un campo es un grupo de caracteres que contiene significado.
- Un registro es un grupo de campos relacionados.
- Por lo menos un campo es seleccionado normalmente en cada registro como registro clave. El registro clave identifica un registro como perteneciente a una persona o entidad particular.
- El tipo de organización más popular para registros en un archivo, se conoce como archivo de acceso secuencial, en el cual se tiene acceso a registros en forma consecutiva hasta que son localizados los datos deseados.
- Un grupo de archivos relacionados a veces se llama una base de datos. Un conjunto o colección de programas diseñados para crear y administrar bases de datos se conoce como un sistema de administración de bases de datos (DBMS, por database management system).
- C considera cada archivo como simplemente un flujo secuencial de bytes.
- Al empezar la ejecución de un programa C abre automáticamente tres archivos y sus flujos asociados —entrada estándar, salida estándar y error estándar.
- Los apuntadores de archivo asignados a la entrada estándar, a la salida estándar y al error estándar son **`stdin`**, **`stdout`** y **`stderr`**, respectivamente.
- La función **`fgetc`** lee un carácter a partir de un archivo especificado.
- La función **`fputc`** escribe un carácter a un archivo especificado.
- La función **`fgets`** lee una línea de un archivo especificado.
- La función **`fputs`** escribe una línea en un archivo especificado.
- **`FILE`** es un tipo de estructura definida en el archivo de cabecera **`stdio.h`**. Para poder utilizar archivo el programador no necesita saber los detalles específicos de esta estructura. Conforme se abre un archivo, se regresa un apuntador a la estructura **`FILE`** del archivo.

- La función **`fopen`** toma dos argumentos— un nombre de archivo y un modo del archivo abierto— y abre el archivo. Si el archivo existe, el contenido del archivo se descarta sin advertencia. Si el archivo no existe y el archivo está siendo abierto para escribir, **`fopen`** crea el archivo.
- La función **`feof`** determina si ha sido definido el indicador de fin de archivo para el mismo.
- La función **`fprintf`** es equivalente a **`printf`**, excepto que **`printf`** recibe como argumento un apuntador al archivo hacia el cual se escribirán los datos.
- La función **`fclose`** cierra el archivo al cual apunta su argumento.
- Para crear un archivo, o para descartar el contenido de un archivo antes de escribir datos, abra el archivo para escritura (**`"w"`**). Para leer un archivo existente, ábralo para lectura (**`"r"`**). Para añadir registros al final de un archivo existente, abra el archivo para agregar (**`"a"`**). Para abrir un archivo de tal forma que pueda ser escrito y leído, abra el archivo para actualizar en alguno de los tres modos de actualización —**`"r+"`**, **`"w+"`** o **`"a+"`**. El modo **`"r+"`** simplemente abre el archivo para lectura y escritura. El modo **`"w+"`** crea el archivo si no existe, y si existe descarta el contenido actual del archivo. En el modo **`"a+"`** se crea el archivo si no existe, y la escritura se efectúa al final del archivo.
- La función **`fscanf`** es equivalente a **`scanf`**, excepto que **`fscanf`** recibe como argumento un apuntador al archivo (normalmente distinto a **`stdin`**) a partir del cual se leerán los datos.
- La función **`rewind`** hace que el programa recoloque el apuntador de posición del archivo correspondiente al archivo especificado al principio del mismo.
- Se utiliza el procesamiento de acceso directo de archivos para tener acceso.directo a registros.
- Para facilitar el acceso directo, los datos se almacenan en registros de longitud fija. Dado que todos los registros son de la misma longitud, la computadora puede rápidamente calcular (como una función del registro clave) la posición exacta de un registro en relación con el principio del archivo.
- Los datos pueden ser añadidos fácilmente a un archivo de acceso directo sin destruir otros datos en el archivo. Los datos previamente almacenados en un archivo con registros de longitud fija, también pueden ser modificados y borrados sin tener que reescribir todo el archivo.
- La función **`fwrite`** escribe un bloque (número específico de bytes) de datos a un archivo.
- El operador en tiempo de compilación **`sizeof`** regresa el tamaño en bytes de su operando.
- La función **`fseek`** establece el apuntador de posición de archivo a una posición específica, en un archivo basado en la posición inicial del punto de búsqueda en el archivo. La búsqueda puede iniciarse a partir de una de tres posiciones —**`SEEK_SET`** inicia a partir del principio del archivo, **`SEEK_CUR`** inicia a partir de la posición actual en el archivo y **`SEEK_END`** inicia a partir del fin del archivo.
- La función **`fread`** lee un bloque (número específico de bytes) de datos de un archivo.

Terminología

modo de archivo abierto **`a`**
modo de archivo abierto **`a+`**
campo alfabético
campo alfanumérico
orden alfa
dígito binario
bit
byte

carácter
campo de carácter
conjunto de caracteres
cerrar un archivo
jerarquía de datos
base de datos
sistema de administración de base de datos
dígito decimal
desplazamiento
número de doble precisión
fin de archivo
indicador de fin de archivo
`fclose`
`feof`
`fgetc`
`fgets`
campo
archivo
almacenamiento temporal de archivo
nombre de archivo
modo de archivo abierto
apuntador de archivo
apuntador de posición de archivo
estructura `FILE`
`fopen`
entrada/salida con formato
`fprintf`
`fputc`
`fputs`
`fread`
`fscanf`
`fseek`
`fwrite`
número entero
espacios a la izquierda
letra
campo numérico
desplazamiento
abrir un archivo
modo de archivo abierto `r`
acceso directo
archivo de acceso directo
registro
registro clave
parámetro del número de registro
`rewind`
modo de archivo abierto `r+`
`SEEK_CUR`
`SEEK_END`
`SEEK_SET`
archivo de acceso secuencial
número de una sola precisión
`stderr` (error estándar)
`stdin` (entrada estándar)
`stdout` (salida estándar)
flujo
espacios a la derecha
modo de archivo abierto `w`
modo de archivo abierto `w+`
ceros y unos

Errores comunes de programación

11.1 Abrir un archivo existente para escritura (`"w"`) cuando, de hecho, el usuario desea conservar el archivo; el contenido del archivo se descartará sin advertencia.
11.2 Olvidar abrir un archivo antes de intentar hacer referencia a él en un programa.
11.3 Usar el apuntador de archivo incorrecto para referirse a un archivo.
11.4 Abrir para lectura un archivo no existente.
11.5 Abrir un archivo para lectura o escritura sin haber obtenido los derechos de acceso al archivo apropiados (esto depende del sistema operativo).
11.6 Abrir un archivo para escritura cuando no hay espacio disponible en disco. En la figura 11.6 se listan los modos de apertura de archivos.
11.7 Puede llevar a errores devastadores abrir un archivo utilizando el modo de archivo incorrecto. Por ejemplo, abrir un archivo en el modo de escribir (`"w"`) cuando debería haberse abierto en modo de actualizar (`"r+"`) hace que sea descartado todo el contenido del archivo.

Prácticas sanas de programación

11.1 Asegúrese que en un programa las llamadas a las funciones de procesamiento de archivos contienen los apuntadores de archivo correctos.

11.2 Cierre cada archivo en forma explícita, tan pronto sepa que el programa ya no hará otra vez referencia al archivo.
11.3 Abra un archivo únicamente para lectura (y no para actualizar), si el contenido del archivo no debe modificarse. Esto evitará cambios no intencionales en el contenido del archivo. Este es otro ejemplo del principio del mínimo privilegio.

Sugerencia de rendimiento

11.1 Cerrar un archivo puede liberar recursos que están siendo esperados por otros usuarios o programas.
11.2 Muchos programadores piensan erróneamente que `sizeof` es una función, y que utilizándolo se genera una sobrecarga en tiempo de ejecución de una llamada de función. No existe tal sobrecarga, porque `sizeof` es un operador en tiempo de compilación.

Sugerencia de portabilidad

11.1 La estructura `FILE` depende del sistema operativo (es decir, los miembros de la estructura varían de un sistema a otro, según la forma en que cada sistema maneja sus archivos).

Ejercicios de autoevaluación

11.1 Llene los espacios vacíos en cada uno de los siguientes:
a) En última instancia, todos los elementos de datos procesados en una computadora se reducen a combinaciones de _____ y _____.
b) El elemento de datos más pequeño que puede procesar una computadora se conoce como un _____.
c) Un _____ es un grupo de registros relacionados.
d) Dígitos, letras y símbolos especiales se conocen como _____.
e) Un grupo de archivos relacionados se llama una _____.
f) La función _____ cierra un archivo.
g) El enunciado _____ lee datos de un archivo en una forma similar a la forma en que `scanf` lee a partir de `stdin`.
h) La función _____ lee un carácter de un archivo especificado.
i) La función _____ lee una línea de un archivo especificado.
j) La función _____ abre un archivo.
k) La función _____ se utiliza normalmente para leer datos de un archivo en aplicaciones de acceso directo.
l) La función _____ recoloca el apuntador de posición de archivo a una posición específica dentro del archivo.

11.2 Indique cuáles de los siguientes son verdaderos y cuáles son falsos (para aquellos que son falsos, explique por qué).
a) La función `fscanf` no puede ser utilizada para leer datos de la entrada estándar.
b) El programador debe utilizar `fopen` explícitamente, para abrir los flujos de entrada estándar, salida estándar y error estándar.
c) Para cerrar un archivo un programa debe llamar en forma explícita a la función `fclose`.
d) Si el apuntador de posición de archivo apunta a una posición en un archivo secuencial distinto al principio del mismo, el archivo debe de ser cerrado y vuelto a abrir para leer a partir del principio del mismo.
e) La función `fprintf` puede escribir a la salida estándar.
f) Los datos en los archivos de acceso secuencial se actualizan siempre sin sobreescribir otros datos.

g) No es necesario buscar en todos los registros de un archivo de acceso directo para encontrar un registro específico.
h) Los registros en archivos de acceso directo no son de longitud uniforme.
i) La función **fseek** puede buscar únicamente en relación con el principio de un archivo.

11.3 Escriba un solo enunciado para que se ejecuten cada uno de los siguientes. Suponga que cada uno de estos enunciados se aplica al mismo programa.

a) Escriba un enunciado que abra el archivo `"oldmast.dat"` para lectura y asigne el apuntador de archivo regresado a `ofPtr`.
b) Escriba un enunciado que abra el archivo `"trans.dat"` para lectura y asigne el apuntador de archivo regresado a `tfPtr`.
c) Escriba un enunciado que abra el archivo `"newmast.dat"` para escritura (y creación) y asigne el apuntador de archivo regresado a `nfPtr`.
d) Escriba un enunciado que lea un registro del archivo `"oldmast.dat"`. El registro está formado del entero `accountNum`, de la cadena `Name`, y del punto flotante `currentBalance`.
e) Escriba un enunciado que lea un registro del archivo `"trans.dat"`. El registro está formado del entero `accountNum` y del punto flotante `dollarAmount`.
f) Escriba un enunciado que escriba un registro al archivo `"newmast.dat"`. El registro está formado del entero `accountNum`, de la cadena `Name`, y del punto flotante `currentBalance`.

11.4 Encuentre el error en cada uno de los siguientes segmentos de programa. Explique cómo se puede corregir dicho error.

a) El archivo referido por `fPtr` (`"payables.dat"`) no ha sido abierto.
```
fprintf(fPtr, "%d%s%d\n", account, company, amount);
```
b)
```
open("receive.dat", "r+");
```
c) El siguiente enunciado debería leer un registro del archivo `"payables.dat"`. El apuntador de archivo `payPtr` se refiere a este archivo, y el apuntador de archivo `recPtr` se refiere al archivo `"receive.dat"`
```
fscanf(recPtr, "%d%s%d\n", &account, company, &amount);
```
d) El archivo `"tools.dat"` debería ser abierto para añadir datos al archivo, sin descartar los datos actuales.
```
if ((tfPtr = fopen("tools.dat", "w")) != NULL)
```
e) El archivo `"courses.dat"` debería ser abierto para agregar sin modificar el contenido actual del archivo.
```
if ((cfPtr = fopen("courses.dat", "w+")) != NULL)
```

Respuestas a los ejercicios de autoevaluación

11.1 a) 1s, 0s. b) Bit, c) Archivo. d) Caracteres. e) Bases de datos f) `fclose`. g) `fscanf`. h) `getc` o bien `fgetc`. i) `fgets`. j) `fopen`. k) `fread`. l)`fseek`.

11.2 a) Falso. La función `fscanf` sólo puede ser utilizada para leer a partir de la entrada estándar incluyendo `stdin`, el apuntador al flujo estándar de entrada, en la llamada a `fscanf`.
b) Falso. Estos tres flujos son automáticamente abiertos por C cuando se inicia la ejecución del programa.
c) Falso. Los archivos serán terminados cuando se termine la ejecución del programa, pero todos los archivos deberían ser cerrados en forma explícita utilizando `fclose`.
d) Falso. La función `rewind` puede ser utilizada para volver a colocar el apuntador de posición de archivo al principio del archivo.
e) Verdadero.

f) Falso. En la mayor parte de los casos, los registros de archivos secuenciales no son de longitud uniforme. Por lo tanto, es posible que al actualizar un registro se cause la sobreescritura de otros datos.
g) Verdadero.
h) Falso. Los registros en un archivo de acceso directo normalmente son de longitud uniforme.
i) Falso. Es posible buscar a partir del principio del archivo, a partir del fin del archivo y a partir de la posición actual en el archivo, de acuerdo con el apuntador de posición de archivo.

11.3
a)
```
ofPtr = fopen("oldmast.dat", "r");
```
b)
```
tfPtr = fopen("trans.dat", "r");
```
c)
```
nfPtr = fopen("newmast.dat", "w");
```
d)
```
fscanf(ofPtr, "%d%s%f", &accountNum, name, &currentBalance);
```
e)
```
fscanf(tfPtr, "%d%f", &accountNum, &dollarAmount);
```
f)
```
fprintf(nfPtr, "%d%s%.2f", accountNum, name, currentBalance);
```

11.4 a) Error: el archivo `"payables.dat"` no ha sido abierto antes de hacer referencia a su apuntador de archivo.
Corrección: utilice `fopen` para abrir `"payables.dat"` para escritura, agregar o actualizar.
b) Error: la función `open` no es una función de ANSI C.
Corrección: utilice la función `fopen`.
c) Error: El enunciado `fscanf` utiliza el apuntador de archivo incorrecto para referirse al archivo `"payables.dat"`.
Corrección: utilice el apuntador de archivo `payPtr` para referirse a `"payables.dat"`.
d) Error: el contenido del archivo será descartado, porque el archivo ha sido abierto para escritura (`"w"`).
Corrección: para añadir datos al archivo, abra el archivo ya sea para actualizar (`"r+"`) o abra el archivo para agregar (`"a"`).
e) Error: el archivo `"courses.dat"` está abierto para actualizar en modo `"w+"`, lo que hace que se descarte el contenido actual del archivo.
Corrección: Abra el archivo en modo `"a"`.

Ejercicios

11.5 Llene los espacios vacíos en cada uno de los siguientes:

a) Las computadoras almacenan grandes cantidades de datos en dispositivos de almacenamiento secundarios como son _____.
b) Un _____ está compuesto de varios campos.
c) Un campo que pudiera contener dígitos, letras y espacios se llama un campo de _____.
d) Para facilitar la recuperación de registros específicos a partir de un archivo, se selecciona un campo de cada registro como _____.
e) La gran mayoría de la información almacenada en sistemas de computación se archiva en archivos _____.
f) Un grupo de caracteres relacionados que contienen significados se conocen como un _____.
g) Los apuntadores de archivo para los tres archivos que se abren automáticamente por C al iniciarse la ejecución de un programa se llaman _____, _____, y _____.
h) La función _____ escribe un carácter a un archivo especificado.
i) La función _____ escribe una línea a un archivo especificado.
j) La función _____ se usa generalmente para escribir datos a un archivo de acceso directo.
k) La función _____ recoloca el apuntador de posición de archivo al principio del mismo.

11.6 Indique cuáles de los siguientes son verdaderos y cuáles son falsos (en el caso de los falsos, explique porqué):

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

a) Las funciones impresionantes ejecutadas por las computadoras involucran esencialmente el manejo de ceros y de unos.
b) Las personas prefieren manipular bits en vez de caracteres y campos, porque los bits son más compactos.
c) Las personas especifican elementos de programas y de datos como caracteres; las computadoras a continuación manipulan y procesan dichos caracteres como grupos de ceros y de unos.
d) El código postal de una persona es un ejemplo de un campo numérico.
e) En aplicaciones de computadora, la calle de la dirección de una persona se considera generalmente un campo alfabético.
f) Los elementos de datos procesados por una computadora forman una jerarquía de datos, en la cual los elementos de datos se hacen más grandes y más complejos conforme se avanza desde los campos a los caracteres, a los bits, etcétera.
g) Un registro clave identifica un registro como perteneciente a un campo particular.
h) La mayor parte de las organizaciones almacenan toda su información en un solo archivo a fin de facilitar el procesamiento en computadora.
i) Los archivos son siempre referidos por nombre en los programas de C.
j) Cuando un programa genera un archivo, el archivo es automáticamente conservado por la computadora para referencia futura.

11.7 El ejercicio 11.3 le solicitó al lector que escribiera una serie de enunciados sencillos. De hecho, estos enunciados forman el núcleo de un tipo importante de programas de procesamiento de archivos, es decir, un programa de cotejo de archivos. En el procesamiento comercial de datos, es común tener varios archivos en cada sistema. Por ejemplo, en un sistema de cuentas por cobrar, generalmente existe un archivo maestro que contiene información detallada relativa a cada cliente como es nombre, dirección, número telefónico, saldo actual, límite de crédito, descuentos, arreglos contractuales, y posiblemente una historia condensada de las compras más recientes y de sus pagos.

Conforme ocurren transacciones (es decir, se efectúan ventas y llegan pagos por el correo), se introducen en un archivo. Al final de cada periodo de negocios (es decir, en algunas empresas un mes, una semana en otras y en algunos casos un día) el archivo de transacciones (llamado `"trans.dat"` en el ejercicio 11.3) se aplica al archivo maestro (llamado `"oldmast.dat"` del ejercicio 11.3), actualizando así el registro de cada cuenta en compras y pagos. Después de cada una de estas corridas de actualización, el archivo maestro se vuelve a escribir como un nuevo archivo (`"newmast.dat"`), que entonces se utiliza al fin del siguiente periodo de negocios, para volver a empezar el siguiente proceso de actualización.

Los programas de cotejo de archivos deben enfrentarse a ciertos problemas que no existen en programas de un solo archivo. Por ejemplo, no siempre ocurrirá una coincidencia. Un cliente existente en el archivo maestro pudiera, en el periodo de negocios actual, no haber efectuado ninguna compra o ningún pago y, por lo tanto, en el archivo de transacción no aparecerá en ningún registro para este cliente. Similarmente, un cliente que sí hizo algunas compras o pagos, quizás acaba de trasladarse a esta comunidad, y la compañía todavía no habrá tenido la oportunidad de crear un registro maestro para dicho cliente.

Utilice los enunciados escritos en el ejercicio 11.3 como base para escribir un programa completo para cotejar cuentas por cobrar. Utilice el número de cuenta en cada archivo como registro clave para cotejo. Suponga que cada archivo es un archivo secuencial con registros almacenados en orden creciente de número de cuenta.

Cuando ocurra una coincidencia (es decir, cuando registros con el mismo número de cuenta aparezcan tanto en el archivo maestro como en el archivo de transacción) añada la cantidad en dólares existente en el archivo de transacción al saldo actual existente en el archivo maestro, y escriba el registro `"newmast.dat"`. (Suponga que las adquisiciones o compras están indicadas por cantidades positivas en el archivo de transacción, y que los pagos están indicados por cantidades negativas.) Cuando exista un registro maestro relativo a una cuenta particular sin registro de transacción correspondiente, simplemente

escriba el registro maestro a `"newmast.dat"`. Cuando exista un registro de transacción sin el correspondiente registro maestro, imprima el mensaje `"Unmatched transaction record for account number ..."` (incluya el número de cuenta proveniente del registro de transacción).

11.8 Después de escribir el programa del ejercicio 11.7 escriba un programa sencillo para crear alguna información de prueba para verificar el programa del ejercicio 11.7. Utilice los datos de cuentas de muestras siguientes:

Archivo maestro Número de cuenta	Nombre	Balance
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Suzy Green	-14.22

Archivo de transacción: Número de cuenta	Cantidad en dólares
100	27.14
300	62.11
400	100.56
900	82.17

11.9 Ejecute el programa del ejercicio 11.7 utilizando los archivos de los datos de prueba creados en el ejercicio 11.8 Utilice el programa enlistado de la sección 11.7 para imprimir un nuevo archivo maestro. Verifique cuidadosamente los resultados.

11.10 Es posible (y de hecho común) tener varios registros de transacciones para un mismo registro clave. Esto ocurre porque un cliente en particular durante un periodo de negocios pudiera efectuar varias compras y varios pagos. Vuelva a escribir su programa de cotejo de cuentas por cobrar del ejercicio 11.7 para que incluya la posibilidad de manejar varios registros de transacción con el mismo registro clave. Modifique los datos de prueba del ejercicio 11.8 para incluir los siguientes registros de transacción adicionales:

Número de cuenta	Cantidad en dólares
300	83.89
700	80.78
700	1.53

11.11 Escriba enunciados que lleven a cabo cada uno de los siguientes. Suponga que la estructura

```
struct person {
   char lastName[15];
   char firstName[15];
   char age[2];
}
```

ha sido definida, y que el archivo ya está abierto para escritura.

a) Inicialice el archivo `"nameage.dat"` de tal forma que existan 100 registros con `lastName = "unassigned"`, `firstname = ""`, y `age = "0"`
b) Introduzca `10` apellidos, nombres y edades y escríbalos al archivo.
c) Actualice un registro; si no existe información en el registro, indique al usuario `"No info"`.
d) Borre un registro que tenga información mediante la reinicialización de dicho registro en particular.

11.12 Usted es el propietario de una ferretería y necesita mantener un inventario que le pueda indicar cuáles son las herramientas que tiene, cuántas tiene y el costo de cada una. Escriba un programa que inicialice el archivo `"hardware.dat"` a 100 registros vacíos, que le permita introducir los datos correspondientes a cada herramienta, enliste todas sus herramientas, borrar un registro correspondiente a una herramienta que ya no posea, y le deje actualizar *cualquier* información dentro del archivo. El número de identificación de la herramienta deberá ser el número de registro. Utilice la siguiente información para iniciar su archivo:

Registro #	Nombre de la herramienta	Cantidad	Costo
3	Electric sander	7	57.98
17	Hammer	76	11.99
24	Jig saw	21	11.00
39	Lawn mower	3	79.50
56	Power saw	18	99.99
68	Screwdriver	106	6.99
77	Sledge hammer	11	21.50
83	Wrench	34	7.50

11.13 ***Generador de palabras de números telefónicos.*** Los marcadores telefónicos estándar contienen los dígitos 0 al 9. Los números 2 hasta el 9 cada uno de ellos tiene tres letras asociadas, tal y como se indican en la tabla siguiente:

Dígito	Letras
2	A B C
3	D E F
4	G H I
5	J K L
6	M N O
7	P R S
8	T U V
9	W X Y

Muchas personas encuentran difícil memorizar los números telefónicos, por lo que utilizan la correspondencia existente entre dígitos y letras para desarrollar palabras de siete letras, que correspondan a sus números telefónicos. Por ejemplo, una persona cuyo número telefónico es 686-2377 pudiera utilizar la correspondencia indicada en la tabla precedente, para desarrollar la palabra de siete letras "NUMBERS".

Los negocios frecuentemente intentan obtener números telefónicos que sean fáciles de recordar para sus clientes. Si un negocio puede anunciar una sola palabra para que la marquen sus clientes, entonces sin duda dicho negocio recibirá unas cuantas llamadas más.

Cada palabra de siete letras corresponde a exactamente un número telefónico de siete dígitos. El restaurante que desee aumentar su negocio de venta de comidas para llevar, podría seguramente hacerlo con el número 825-3688 (es decir "TAKEOUT").

Cada número telefónico de 7 dígitos corresponde a muchas palabras distintas de 7 letras. Desafortunadamente, la mayor parte de ellas representan yuxtaposiciones irreconocibles de letras. Es posible, sin embargo, que el propietario de una peluquería se sintiera complacido al saber que el número telefónico de su tienda, 424-7288 corresponde a "HAIRCUT". El propietario de una tienda de licores estaría encantado, sin duda, al averiguar que el número telefónico de la tienda 233-7226, corresponde a "BEERCAN". Un médico veterinario con el número telefónico 738-2273 estaría complacido al aprender que el número corresponde a las letras "PETCARE".

Escriba un programa en C que, dado un número de 7 dígitos, escriba a un archivo todas las palabras posibles de 7 letras que correspondan a dicho número. Existen 2187 (3 a la séptima potencia) palabras posibles. Evite números telefónicos que contengan los dígitos 0 y 1.

11.14 Si tiene disponible un diccionario computarizado, modifique el programa que escribió en el Ejercicio 11.13 para buscar las palabras en el diccionario. Algunas combinaciones de siete letras creadas por este programa consisten de dos o más palabras (el número telefónico 843-2677 produce "THEBOSS").

11.15 Modifique el ejemplo de la figura 8.14 para utilizar las funciones `fgetc` y `fputs`, en vez de `getchar` y `puts`. El programa deberá darle al usuario la opción de leer desde la entrada estándar y escribir a la salida estándar, o de leer de un archivo especificado y escribir a un archivo especificado. Si el usuario se decide por la segunda opción, haga que el usuario introduzca los nombres de archivo correspondientes a los archivos de entrada y de salida.

11.16 Escriba un programa que utilice el operador `sizeof` para determinar los tamaños en bytes de varios tipos de datos en su sistema de computación. Escriba los resultados al archivo `"datasize.dat"` de tal forma que más tarde se puedan imprimir los resultados. El formato para los resultados del archivo deberá ser:

```
Data type             Size
char                     1
unsigned char            1
short int                2
unsigned short int       2
int                      4
unsigned int             4
long int                 4
unsigned long int        4
float                    4
double                   8
long double             16
```

Nota: los tamaños de tipo en su computadora pudieran no ser iguales a los arriba listados.

11.17 En el ejercicio 7.19, usted escribió una simulación en software de una computadora, que utilizaba un lenguaje máquina especial llamado lenguaje de máquina Simpletron (LMS). En la simulación, cada vez que usted deseaba ejecutar un programa LMS, usted introducía el programa en el simulador a partir del teclado. Si al capturar el programa LMS cometía algún error, el simulador se volvía a iniciar y el código LMS se volvía a introducir. Sería agradable, en vez de tener que escribirlo cada vez, poder leer el programa LMS a partir de un archivo. Esto reduciría tiempo y errores en la preparación de la ejecución de programas LMS.

a) Modifique el simulador que escribió en el ejercicio 7.19 para leer programas LMS a partir de un archivo especificado por el usuario desde el teclado.
b) Una vez que Simpletron se ejecuta, saca el contenido de sus registros y de memoria a la pantalla. Sería también deseable capturar la salida en un archivo, por lo que modifique el simulador para escribir su salida a un archivo, además de que despliegue dicha salida en la pantalla.

12

Estructuras de datos

Objetivos

- Ser capaz de asignar y liberar memoria dinámicamente para objetos de datos.
- Ser capaz de formar estructuras de datos enlazadas mediante el uso de apuntadores, estructuras autorreferenciadas y recursión.
- Ser capaz de crear y manipular listas enlazadas, colas, pilas y árboles binarios.
- Comprender varias aplicaciones importantes de las estructuras de datos enlazadas.

Mucho de lo que sujeto, no puedo liberar;
Mucho que puedo liberar regresó a mí.
Lee Wilson Dodd

'¿Quieres andar un poco más rápido?' dijo una merluza a un caracol. 'Hay una tortuga detrás de nosotros, y me está pisando la cola'.
Lewis Carroll

Siempre hay lugar en la cima.
Daniel Webster

Empuja —mantente en movimiento.
Thomas Morton

Pienso que no veré jamás
Tan bello poema como es un árbol.
Joyce Kilmer

Sinopsis

Resumen • Terminología • Errores comunes de programación • Prácticas sanas de programación • Sugerencias de rendimiento • Sugerencias de portabilidad • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios.

12.1 Introducción

Hemos estudiado *estructuras de datos* de tamaño fijo, como arreglos de un solo subíndice, de doble subíndice y **structs**. Este capítulo presenta *estructuras dinámicas de datos,* cuyo tamaño crece y se encoge en tiempo de ejecución. Las *listas enlazadas* son colecciones de elementos de datos "alineados en una fila" —en una lista enlazada las inserciones y las eliminaciones se efectúan en cualquier parte. Las *pilas* son importantes en compiladores y sistemas operativos —en una pila las inserciones y las eliminaciones se efectúan únicamente en un extremo— en su *parte superior.* Las *colas de espera* representan líneas de espera; las inserciones se efectúan en la parte trasera (también conocida como la *cola*) de la misma, y las eliminaciones se hacen de la parte delantera (también conocida como cabeza de la cola). Los *árboles binarios* facilitan la búsqueda y clasificación de los datos a alta velocidad, la eliminación eficiente de elementos duplicados de datos, la representación de sistemas de directorios de archivo y las expresiones de compilación en lenguaje de máquina. Cada una de esas estructuras de datos tiene muchas otras interesantes aplicaciones.

Analizaremos cada uno de los tipos principales de estructuras de datos y pondremos en operación programas que crean y manipulan estas estructuras de datos. En la siguiente parte del libro —la introducción a **C++** y a la programación orientada a objetos, en los capítulos del 15 hasta el 21 —estudiaremos la abstracción de datos. Esta técnica nos permitirá construir estas estructuras de datos de una forma dramáticamente diferente, diseñada para producir software mucho más fácil de mantener y especialmente mucho más fácil de volver a utilizar.

Este es un capítulo retador. Los programas son sustanciales e incorporan la mayor parte de lo que ha aprendido en los capítulos anteriores. Estos programas están llenos de manipulaciones de apuntadores, un tema que muchas personas consideran de entre los temas más difíciles de C. El capítulo está lleno de programas altamente prácticos, que podrá utilizar en cursos más avanzados; el capítulo incluye una valiosa colección de ejercicios que enfatizan aplicaciones prácticas de las estructuras de datos.

Sinceramente esperamos que usted intentará el importante proyecto descrito en la sección especial titulada "Cómo construir su propio compilador". Usted ha estado utilizando un compilador para traducir sus programas C a lenguaje de máquina, a fin de poder ejecutar sus programas en su computadora. En este proyecto, realmente construirá su propio compilador. Este leerá un archivo de enunciados, escritos en un simple, aunque poderoso lenguaje de alto nivel, similar a las versiones primeras del popular lenguaje BASIC. Su compilador convertirá estos enunciados en un archivo de instrucciones de lenguaje de máquina Simpletron. LMS es el lenguaje que aprendió en la sección especial del capítulo 7, "Cómo construir su propia computadora". ¡A continuación su programa simulador Simpletron ejecutará el programa LMS producido por su compilador! Este proyecto le dará la maravillosa oportunidad de practicar la mayor parte de lo que en este curso ha aprendido. La sección especial lo lleva cuidadosamente a través de las especificaciones del lenguaje de alto nivel, y describe los algoritmos que necesitará para convertir cada tipo de enunciado del lenguaje de alto nivel a instrucciones en lenguaje máquina. Si usted es afecto a aceptar retos, pudiera intentar llevar a cabo las muchas mejoras, tanto al compilador como al simulador Simpletron que se sugieren en los ejercicios.

12.2 Estructuras autorreferenciadas

Una *estructura autorreferenciada* contiene un miembro de apuntador que apunta a una estructura del mismo tipo de estructura. Por ejemplo, la definición

```
struct node {
   int data;
   struct node *nextPtr;
};
```

define un tipo, **struct node**. Una estructura del tipo **struct node** tiene dos miembros —el miembro entero **data** y el miembro de apuntador **nextPtr**. El miembro **nextPtr** apunta a una estructura de tipo **struct node** —una estructura del mismo tipo que la que se está declarando aquí, de ahí el "término estructura autorreferenciada". El miembro **nextPtr** se conoce como un *enlace o vínculo* —es decir, **nextPtr** puede ser utilizada para "vincular" una estructura del tipo **struct node** con otra estructura del mismo tipo. Las estructuras autorreferenciadas pueden ser enlazadas juntas para formar útiles estructuras de datos como son las listas, las colas de espera, las pilas y los árboles. En la figura 12.1 se ilustran dos estructuras autorreferenciadas, enlazadas juntas para formar una lista. Note que se coloca una diagonal —que representa un apuntador **NULL**— en el miembro enlazado de la segunda estructura autorreferenciada, para indicar que el enlace no apunta a otra estructura. La diagonal aparece sólo para fines de ilustración; no corresponde al carácter de diagonal de C. Igual que el carácter **NULL** indica el final de una cadena, normalmente un apuntador **NULL** indica el fin de una estructura de datos.

Error común de programación 12.1

No establecer a NULL el enlace en el último nodo de una lista.

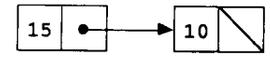


Fig. 12.1 Dos estructuras autorreferencladas enlazadas juntas.

12.3 Asignación dinámica de memoria

La creación y mantenimiento de estructuras dinámicas de datos, requiere de la *asignación dinámica de memoria* —la capacidad por parte de un programa de obtener, en tiempo de ejecución, más espacio de memoria para contener nuevos nodos, y de poder liberar espacio ya no requerido. El límite de la asignación dinámica de memoria pude ser tan grande como la totalidad de memoria física disponible en la computadora, o la cantidad de memoria virtual disponible, en un sistema de memoria virtual. A menudo, los límites son mucho menores, porque la memoria disponible deberá ser compartida entre muchos usuarios.

Las funciones **malloc** y **free** y el operador **sizeof**, son esenciales a la asignación dinámica de memoria. La función **malloc** toma como argumento el número de bytes a asignarse, y regresa un apuntador del tipo **void*** (*apuntador a void*) a la memoria asignada. Un apuntador **void*** puede asignarse a una variable de cualquier tipo de apuntador. Normalmente la función **malloc** se utiliza conjuntamente con el operador **sizeof**. Por ejemplo, el enunciado

```
newPtr = malloc(sizeof(struct node));
```

evalúa **sizeof(struct node)** para determinar el tamaño en bytes de una estructura del tipo **struct node**, asigna en memoria una nueva área de tamaño **sizeof(struct node)** bytes, y almacena en la variable **newPtr** un apuntador a la memoria asignada. Si no existe memoria disponible, **malloc** regresa un apuntador **NULL**.

La función **free** cancela la asignación de la memoria —es decir, se regresa la memoria al sistema, de tal forma que en el futuro ésta pueda ser vuelta a asignar. Para liberar memoria asignada dinámicamente mediante una llamada **malloc** previa, utilice el enunciado

```
free(newPtr);
```

En las siguientes secciones se analizan listas, pilas, colas de espera y árboles. Cada una de estas estructuras de datos se crea y se mantiene utilizando la asignación dinámica de memoria y las estructuras autorreferenciadas.

Sugerencia de portabilidad 12.1

El tamaño de una estructura no es necesariamente la suma de los tamaños de sus miembros. Esto es debido a varios requisitos de alineación de límites, que son dependientes de la máquina (vea el capítulo 10).

Error común de programación 12.2

Suponer que el tamaño de una estructura es simplemente la suma de los tamaños de sus miembros.

Práctica sana de programación 12.1

Para determinar el tamaño de una estructura utilice el operador `sizeof`.

Práctica sana de programación 12.2

Al utilizar **malloc**, *compruebe si es* **NULL** *el valor de regreso de apuntador. Si la memoria solicitada no ha sido asignada imprima un mensaje de error.*

Error común de programación 12.3

No regresar memoria dinámicamente asignada cuando ésta ya no es necesaria, puede hacer que el sistema se quede sin memoria prematuramente. Esto se conoce a veces como "fuga de memoria".

Práctica sana de programación 12.3

Cuando ya no se requiera memoria que fue dinámicamente asignada, utilice **free**, *para regresar esta memoria inmediatamente al sistema.*

Error común de programación 12.4

Utilizando **malloc**, *liberar memoria no dinámicamente asignada.*

Error común de programación 12.5

Referirse a memoria que ya ha sido liberada.

12.4 Listas enlazadas

Una *lista enlazada* es una colección lineal de estructuras autorreferenciadas llamadas *nodos*, conectadas por *enlaces* de apuntador —de ahí el término lista "enlazada". Se tiene acceso a una lista enlazada vía un apuntador al primer nodo de la lista. Se puede tener acceso a los nodos subsecuentes vía el apuntador de enlace almacenado en cada nodo. Por regla convencional, para marcar el fin de la lista, el apuntador de enlace, en el último nodo de una lista, se define a **NULL**. En una lista enlazada los datos se almacenan dinámicamente —cada nodo se crea conforme sea necesario. Un nodo puede contener datos de cualquier tipo, incluyendo otras **struct**. Las pilas y las colas de espera también son estructuras lineales de datos, y como veremos, son versiones restringidas de listas enlazadas. Los árboles son estructuras no lineales de datos.

Las listas de datos pueden ser almacenadas en arreglos, pero las listas enlazadas proporcionan varias ventajas. Una lista enlazada es apropiada cuando no es predecible de inmediato el número de elementos de datos a representarse en la estructura. Las listas enlazadas son dinámicas, por lo que conforme sea necesario la longitud de una lista puede aumentar o disminuir. Por su parte, el tamaño de un arreglo no puede ser modificado, porque la memoria del arreglo es asignada en tiempo de compilación. Los arreglos pueden llenarse. Las listas enlazadas sólo se llenan cuando el sistema no tiene suficiente memoria para satisfacer las solicitudes de asignación dinámica de almacenamiento.

Sugerencia de rendimiento 12.1

Podría declararse un arreglo que contenga más elementos que el número esperado de elementos de datos, pero esto puede desperdiciar memoria. En estas situaciones las listas enlazadas pueden obtener una mejor utilización de la memoria.

Las listas enlazadas pueden mantenerse en orden, insertando cada elemento nuevo en el punto apropiado dentro de la lista.

Sugerencia de rendimiento 12.2

Puede resultar muy tardado insertar y eliminar en un arreglo ya ordenado —deberán ser desplazados en forma apropiada. Todos los elementos que sigan al elemento insertado o borrado.

Sugerencia de rendimiento 12.3

Los elementos de un arreglo se almacenan en forma contigua en memoria. Esto permite acceso inmediato a cualquier arreglo del elemento, porque la dirección de cualquier elemento puede ser calculada directamente, basada en su posición en relación con el principio del arreglo. Las listas enlazadas no proporcionan un acceso inmediato como éste a sus elementos.

Normalmente, los nodos de las listas enlazadas no están almacenados en memoria en forma contigua. Sin embargo, lógicamente, los nodos de una lista enlazada aparecen como contiguos. En la figura 12.2 se ilustra una lista enlazada con varios nodos.

Sugerencia de rendimiento 12.4

Tratándose de estructuras de datos que crecen o se reducen en tiempo de ejecución, es posible ahorrar memoria utilizando asignación dinámica de memoria (en vez de los arreglos). Recuerde, sin embargo, que los apuntadores toman espacio, y que la asignación dinámica de memoria incurre en sobrecarga por las llamadas de función.

El programa de la figura 12.3 (cuya salida se muestra en la figura 12.4) manipula una lista de caracteres. El programa da dos opciones: 1) insertar un carácter en la lista en orden alfabético (función **insert**) y 2) borra un carácter de la lista (función **delete**). Este es un programa grande y complejo. Sigue un análisis detallado respecto al programa. El ejercicio 12.20 solicita al estudiante que ponga en operación una función recursiva, que imprima una lista al revés. En el ejercicio 12.21 se le pide al estudiante que ponga en marcha una función recursiva, que busque en una lista enlazada un elemento particular de datos.

Las dos funciones primarias de las listas enlazadas son **insert** y **delete**. La función **isEmpty** se conoce como una *función predicada* —no altera en forma alguna la lista; más bien determina si la lista está vacía (es decir, si el apuntador al primer nodo de la lista es **NULL**). Si la lista está vacía, se regresa **1**; de lo contrario se regresa **0**. La función **printList** imprime la lista.

En la lista los caracteres se insertan en orden alfabético. La función **insert** recibe la *dirección* de la lista y el carácter a insertarse. Es necesaria la dirección de la lista cuando se va a insertar un valor en el inicio de la lista. Proporcionar la dirección de la lista permite que se pueda modificar la lista (el apuntador al primer nodo de la lista) vía una llamada por referencia. Dado que la lista propiamente dicha es un apuntador (a su primer elemento), pasar la dirección de la lista crea un *apuntador a un apuntador* (es decir, una *doble indirección*). Este es un concepto complejo y requiere de cuidadosa programación. Los pasos para la inserción de un carácter en la lista son como sigue (vea la figura 12.5):

1) Crear un nodo llamando **malloc**, asignando a **newPtr** la dirección de la memoria asignada, asignando el carácter a insertarse a **newPtr->data**, y asignando **NULL** a **newPtr->nextPtr**.
2) Inicialice **previousPtr** a **NULL**, y **currentPtr** a ***sPtr** (el apuntador al inicio de la lista). Los apuntadores **previousPtr** y **currentPtr** se utilizan para almacenar las posiciones del nodo anterior al punto de inserción y del nodo posterior al punto de inserción.

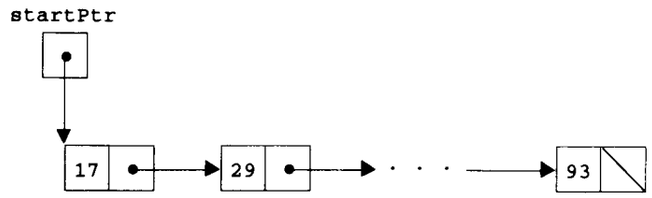


Fig. 12.2 Representación gráfica de una lista enlazada.


```
/* Operating and maintaining a list */
#include <stdio.h>
#include <stdlib.h>

struct listNode {  /* self-referential structure */
   char data;
   struct listNode *nextPtr;
};

typedef struct listNode LISTNODE;
typedef LISTNODE *LISTNODEPTR;

void insert(LISTNODEPTR *, char);
char delete(LISTNODEPTR *, char);
int isEmpty(LISTNODEPTR);
void printList(LISTNODEPTR);
void instructions(void);

main()
{
   LISTNODEPTR startPtr = NULL;
   int choice;
   char item;

   instructions();  /* display the menu */
   printf("? ");
   scanf("%d", &choice);

   while (choice != 3) {

      switch (choice) {
         case 1:
            printf("Enter a character: ");
            scanf("\n%c", &item);
            insert(&startPtr, item);
            printList(startPtr);
            break;
         case 2:
            if (!isEmpty(startPtr)) {
               printf("Enter character to be deleted: ");
               scanf("\n%c", &item);

               if (delete(&startPtr, item)) {
                  printf("%c deleted.\n", item);
                  printList(startPtr);
               }
               else
                  printf("%c not found.\n\n", item);
            }
            else
               printf("List is empty.\n\n");

            break;
```

Fig. 12.3 Cómo insertar y borrar nodos en una lista (parte 1 de 3).

```
            default:
               printf("Invalid choice.\n\n");
               instructions();
               break;
         }

         printf("? ");
         scanf("%d", &choice);
      }

      printf("End of run.\n");
      return 0;
   }

   /* Print the instructions */
   void instructions(void)
   {
      printf("Enter your choice:\n"
             "   1 to insert an element into the list.\n"
             "   2 to delete an element from the list.\n"
             "   3 to end.\n");
   }

   /* Insert a new value into the list in sorted order */
   void insert(LISTNODEPTR *sPtr, char value)
   {
      LISTNODEPTR newPtr, previousPtr, currentPtr;

      newPtr = malloc(sizeof(LISTNODE));

      if (newPtr != NULL) {    /* is space available */
         newPtr->data = value;
         newPtr->nextPtr = NULL;

         previousPtr = NULL;
         currentPtr = *sPtr;

         while (currentPtr != NULL && value > currentPtr->data) {
            previousPtr = currentPtr;          /* walk to ...   */
            currentPtr = currentPtr->nextPtr;  /* ... next node */
         }

         if (previousPtr == NULL) {
            newPtr->nextPtr = *sPtr;
            *sPtr = newPtr;
         }
         else {
            previousPtr->nextPtr = newPtr;
            newPtr->nextPtr = currentPtr;
         }
      }
      else
         printf("%c not inserted. No memory available.\n", value);
   }
```

Fig. 12.3 Cómo insertar y borrar nodos en una lista (parte 2 de 3).


```
   /* Delete a list element */
   char delete(LISTNODEPTR *sPtr, char value)
   {
      LISTNODEPTR previousPtr, currentPtr, tempPtr;

      if (value == (*sPtr)->data) {
         tempPtr = *sPtr;
         *sPtr = (*sPtr)->nextPtr;  /* de-thread the node */
         free(tempPtr);             /* free the de-threaded node */
         return value;
      }
      else {
         previousPtr = *sPtr;
         currentPtr = (*sPtr)->nextPtr;

         while (currentPtr != NULL && currentPtr->data != value) {
            previousPtr = currentPtr;          /* walk to ...   */
            currentPtr = currentPtr->nextPtr;  /* ... next node */
         }

         if (currentPtr != NULL) {
            tempPtr = currentPtr;
            previousPtr->nextPtr = currentPtr->nextPtr;
            free(tempPtr);
            return value;
         }
      }

      return '\0';
   }

   /* Return 1 if the list is empty, 0 otherwise */
   int isEmpty(LISTNODEPTR sPtr)
   {
      return sPtr == NULL;
   }

   /* Print the list */
   void printList(LISTNODEPTR currentPtr)
   {
      if (currentPtr == NULL)
         printf("List is empty.\n\n");
      else {
         printf("The list is:\n");

         while (currentPtr != NULL) {
            printf("%c --> ", currentPtr->data);
            currentPtr = currentPtr->nextPtr;
         }

         printf("NULL\n\n");
      }
   }
```

Fig. 12.3 Cómo insertar y borrar nodos en una lista (parte 3 de 3)

```
Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 1
Enter a character: B
The list is:
B -> NULL

? 1
Enter a character: A
The list is:
A -> B -> NULL

? 1
Enter a character: C
The list is:
A -> B -> C -> NULL

? 2
Enter character to be deleted: D
D not found.

? 2
Enter character to be deleted: B
B deleted.
Tne list is:
A -> C -> NULL

? 2
Enter character to be deleted: C
C deleted.
The list is:
A -> NULL

? 2
Enter character to be deleted: A
A deleted.
List is empty.

? 4
Invalid choice.

Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 3
End of run.
```

Fig. 12.4 Salida de muestra del programa de la figura 12.3.

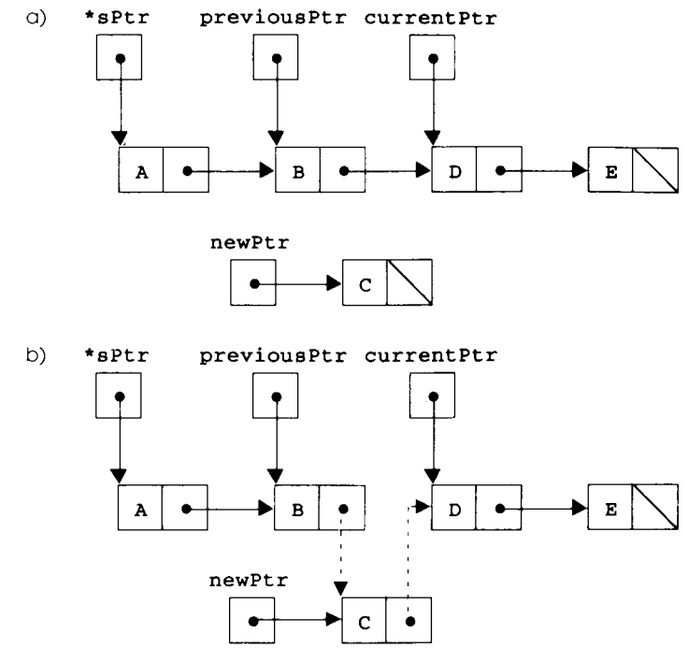


Fig. 12.5 Cómo insertar un nodo en orden dentro de una lista.

3) En tanto **currentPtr** no sea **NULL** y el valor a insertarse sea mayor que **currentPtr->data**, asigne **currentPtr** a **previousPtr** y avance **currentPtr** al siguiente nodo en la lista. Esto posiciona en la lista el punto de inserción del valor.
4) Si **previousPtr** es **NULL**, se inserta el nuevo nodo como primer nodo de la lista. Asigne ***sPtr** a **newPtr->nextPtr** (el nuevo enlace de nodo apunta al anterior primer nodo), y asigne **newPtr** a ***sPtr** (***sPtr** apunta al nuevo nodo). Si **previousPtr** no es **NULL**, el nuevo nodo se inserta en su lugar. Asigne **newPtr** a **previousPtr->nextPtr** (el nodo anterior apunta al nuevo nodo), y asigne **currentPtr** a **newPtr->nextPtr** (el enlace de nuevo nodo apunta al nodo actual).

Práctica sana de programación 12.4

Asigne **NULL** *al miembro de enlace de un nuevo nodo. Los apuntadores deben ser inicializados antes de ser utilizados.*

En la figura 12.5 se ilustra la inserción de un nodo conteniendo el carácter **'C'** en una lista ordenada. La parte a) de la figura muestra la lista y el nuevo nodo antes de la inserción. La parte b) de la figura muestra el resultado de la inserción del nuevo nodo. Los apuntadores reasignados están representados por flechas y líneas punteadas.

La función **delete** recibe la dirección del apuntador hacia el principio de la lista y un carácter a borrarse. Los pasos para borrar un carácter de la lista son como siguen:

1) Si el carácter a borrarse coincide con el primer carácter del primer nodo de la lista, asigna `*sPtr` a `tempPtr` (`tempPtr` será utilizado para liberar, usando `free`, la memoria no necesaria), asigna `(*sPtr)->nextPtr` a `*sPtr` (`*sPtr` ahora apunta al segundo nodo de la lista), `free` libera la memoria apuntada por `tempPtr`, y regresa el carácter que fue borrado.
2) De no ser así, inicializa `previousPtr` con `*sPtr` e inicializa `currentPtr` con `(*sPtr)->nextPtr`.
3) En tanto `currentPtr` no sea `NULL` y el valor a borrarse no sea igual `currentPtr->data`, asigna `currentPtr` a `previousPtr`, y asigna `currentPtr->nextPtr` a `currentPtr`. Esto localizará el carácter a borrarse, si está contenido dentro de la lista.
4) Si `currentPtr` no es `NULL`, asigna `currentPtr` a `tempPtr`, asigna `currentPtr->nextPtr` a `previousPtr->nextPtr`, libera el nodo al cual apunta `tempPtr`, y regresa el carácter que fue borrado de la lista. Si `currentPtr` es `NULL`, regresa el carácter `NULL ('\0')`, para significar que el carácter a borrarse no fue encontrado dentro de la lista.

En la figura 12.6 se ilustra el borrado de un nodo de una lista enlazada. La parte a) de la figura muestra la lista enlazada, antes de la operación de inserción anterior. La parte b) muestra la reasignación del elemento de enlace de `previousPtr` y la asignación de `currentPtr` a `tempPtr`. El apuntador `tempPtr` se utiliza para liberar la memoria asignada para almacenar `'C'`.

La función `printList` recibe un apuntador al inicio de la lista como un argumento, y se refiere al apuntador como `currentPtr`. La función primero determina si la lista está vacía. Si es así, `printList` imprime `"The list is empty"` y termina. De lo contrario, imprime

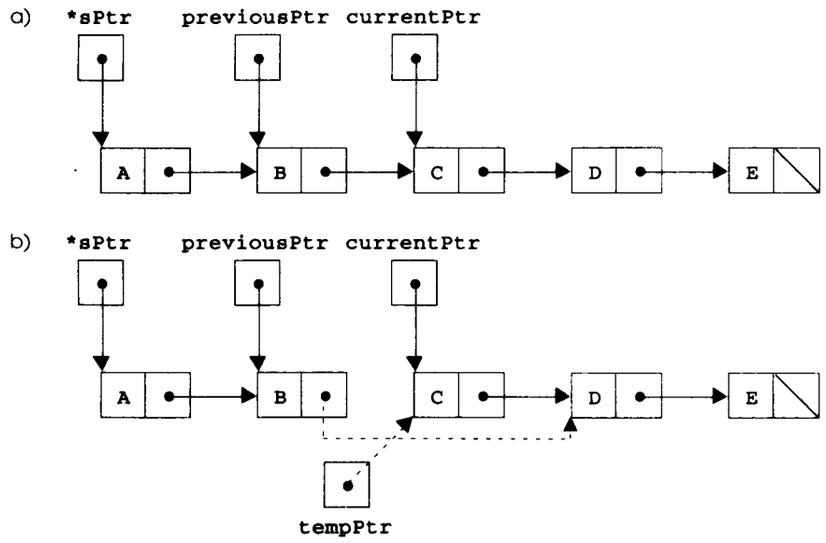


Fig. 12.6 Cómo borrar un nodo de una lista.

los datos en la lista. En tanto `currentPtr` no sea `NULL`, `currentPtr->data` será impreso por la función, y `currentPtr->nextPtr` será asignado a `currentPtr`. Note que si en el último nodo el enlace de la lista no es `NULL`, el algoritmo de impresión tratará de imprimir más allá del final de la lista, y ocurrirá un error. El algoritmo de impresión es idéntico para listas enlazadas, pilas y colas de espera.

12.5 Pilas

Una *pila* es una versión restringida de una lista enlazada. A una pila se le pueden añadir y retirar nuevos nodos únicamente de su parte superior. Por esta razón, se conoce una pila como una estructura de datos como *últimas entradas, primeras salidas* (*LIFO por last-in, first-out*). Se referencía una pila mediante un apuntador al elemento superior de la misma. El miembro de enlace en el último nodo de la pila se define a `NULL`, para indicar que se trata de la parte inferior de la pila misma.

En la figura 12.7 se ilustra una pila con varios nodos. Note que las pilas y las listas enlazadas se representan en forma idéntica. La diferencia entre las pilas y las listas enlazadas es que en una lista enlazada las inserciones y borrados pueden ocurrir en cualquier parte, pero en una pila únicamente en su parte superior.

Error común de programación 12.6

No definir a `NULL` el enlace en el nodo inferior de una pila.

Las funciones primarias utilizadas para manipular una pila son ***push*** y ***pop***. La función `push` crea un nuevo nodo y lo coloca en la parte superior de la pila. La función `pop` elimina un nodo de la parte superior de la pila, liberando la memoria que fue asignada al nodo retirado, y regresando el valor retirado.

El programa de la figura 12.8 (cuya salida se muestra en la salida 12.9) representa una pila de enteros simple. El programa presenta tres opciones: 1) incluir (push) un valor en la pila (función `push`), 2) retirar (pop) un valor de la pila (función `pop`) y 3) terminar el programa.

La función `push` coloca un nuevo nodo en la parte superior de la pila. La función está formada de tres pasos:

1) Crear un nuevo nodo llamando a `malloc`, asignando la posición de la memoria asignada a `newPtr`, asignando el valor a colocarse en la pila a `newPtr->data`, y asignar `NULL` a `newPtr->nextPtr`.
2) Asignar `*topPtr` (el apuntador a la parte superior de la pila) a `newPtr->nextPtr` —el miembro de enlace de `newPtr` ahora apunta al nodo superior anterior.
3) Asigna `newPtr` a `*topPtr` —`*topPtr` apunta ahora a la nueva parte superior de la pila.

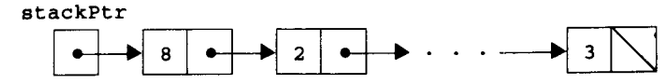


Fig. 12.7 Representación gráfica de una pila.

```
/* dynamic stack program */
#include <stdio.h>
#include <stdlib.h>

struct stackNode {  /* self-referential structure */
   int data;
   struct stackNode *nextPtr;
};

typedef struct stackNode STACKNODE;
typedef STACKNODE *STACKNODEPTR;

void push(STACKNODEPTR *, int);
int pop(STACKNODEPTR *);
int isEmpty(STACKNODEPTR);
void printStack(STACKNODEPTR);
void instructions(void);

main()
{
   STACKNODEPTR stackPtr = NULL;  /* points to the stack top */
   int choice, value;

   instructions();
   printf("? ");
   scanf("%d", &choice);

   while (choice != 3) {

      switch (choice) {
         case 1:      /* push value onto stack */
            printf("Enter an integer: ");
            scanf("%d", &value);
            push(&stackPtr, value);
            printStack(stackPtr);
            break;
         case 2:      /* pop value off stack */
            if (!isEmpty(stackPtr))
               printf("The popped value is %d.\n",
                      pop(&stackPtr));

            printStack(stackPtr);
            break;
         default:
            printf("Invalid choice.\n\n");
            instructions();
            break;
      }

      printf("? ");
      scanf("%d", &choice);
   }

   printf("End of run.\n");
   return 0;
}
```

Fig. 12.8 Un programa de pilas simple (parte 1 de 3).


```
/* Print the instructions */
void instructions(void)
{
   printf("Enter choice:\n"
          "1 to push a value on the stack\n"
          "2 to pop a value off the stack\n"
          "3 to end program\n");
}

/* Insert a node at the stack top */
void push(STACKNODEPTR *topPtr, int info)
{
   STACKNODEPTR newPtr;

   newPtr = malloc(sizeof(STACKNODE));
   if (newPtr != NULL) {
      newPtr->data = info;
      newPtr->nextPtr = *topPtr;
      *topPtr = newPtr;
   }
   else
      printf("%d not inserted. No memory available.\n", info);
}

/* Remove a node from the stack top */
int pop(STACKNODEPTR *topPtr)
{
   STACKNODEPTR tempPtr;
   int popValue;

   tempPtr = *topPtr;
   popValue = (*topPtr)->data;
   *topPtr = (*topPtr)->nextPtr;
   free(tempPtr);
   return popValue;
}

/* Print the stack */
void printStack(STACKNODEPTR currentPtr)
{
   if (currentPtr == NULL)
      printf("The stack is empty.\n\n");
   else {
      printf("The stack is:\n");

      while (currentPtr != NULL) {
         printf("%d --> ", currentPtr->data);
         currentPtr = currentPtr->nextPtr;
      }

      printf("NULL\n\n");
   }
}
```

Fig. 12.8 Un programa de pilas simple (parte 2 de 3).

```
/* Is the stack empty? */
int isEmpty(STACKNODEPTR topPtr)
{
   return topPtr == NULL;
}
```

Fig. 12.8 Un programa de pilas simple (parte 3 de 3).

```
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1
Enter an integer: 5
The stack is:
5 --> NULL

? 1
Enter an integer: 6
The stack is:
6 --> 5 --> NULL

? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL

? 2
The popped value is 4.
The stack is:
6 --> 5 --> NULL

? 2
The popped value is 6.
The stack is:
5 --> NULL

? 2
The popped value is 5.
The stack is empty.

? 2
The stack is empty.

? 4
Invalid choice.

Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 3
End of run.
```

Fig. 12.9 Salida de muestra correspondiente al programa de la figura 12.8.

Las manipulaciones que involucran a ***topPtr** modifican el valor de **stackPtr** en **main**. La figura 12.10 ilustra la función **push**. La parte a) de la figura muestra la pila y el nuevo nodo, antes de la operación **push**. En la parte b) las flechas con líneas punteadas ilustran los pasos 2 y 3 de la operación **push**, que le permiten al nodo que contiene **12** convertirse en la nueva parte superior de la pila. La función **pop** retira un nodo de la parte superior de la pila. Note que **main** determina, antes de llamar a **pop**, si la pila está vacía. La operación **pop** consiste de cinco pasos.

1) Asigna ***topPtr** a **tempPtr** (**tempPtr** se utilizará para liberar memoria no necesaria).
2) Asigna **(*topPtr)->data** a **popValue** (guarda el valor almacenado en el nodo superior).
3) Asigna **(*topPtr)->nextPtr** a ***topPtr** (asigna ***topPtr** la dirección del nuevo nodo superior).
4) Libera la memoria a la cual apunta **tempPtr**.
5) Regresa **popValue** al llamador (**main**, en el programa de la figura 12.8).

La figura 12.11 ilustra el uso de la función **pop**. La parte a) muestra la pila antes de la operación anterior **push**. La parte b) muestra **tempPtr** apuntando al primer nodo de la pila y **topPtr** apuntando al segundo nodo de la misma. La función **free** es utilizada para liberar la memoria a la cual apunta **tempPtr**.

Las pilas tienen muchas aplicaciones interesantes. Por ejemplo, siempre que se hace una llamada de función, la función llamada debe saber cómo regresar a su llamador, por lo que la dirección de regreso es introducida en una pila. Si ocurre una serie de llamadas de función, los valores de regreso sucesivos son introducidos en la pila, en orden de últimas entradas, primeras salidas, de forma tal que cada función pueda regresar a su llamador. Las pilas aceptan llamadas de función recursivas, de la misma forma que llamadas normales no recursivas.

Las pilas contienen el espacio creado para variables automáticas en cada invocación de una función. Cuando la función regresa a su llamador, el espacio para las variables automáticas de dicha función es retirado (popped off) de la pila, y dichas variables dejan de ser conocidas para el programa.

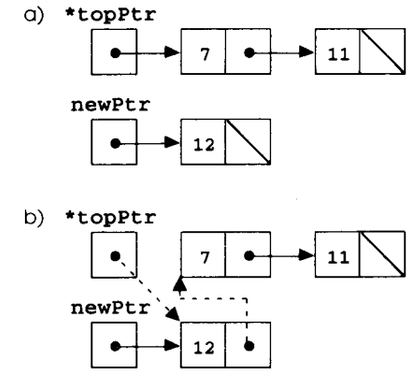


Fig. 12.10 La operación **push**.

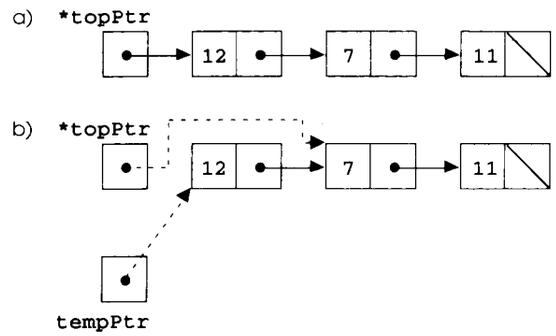


Fig. 12.11 La operación **pop**.

Las pilas son utilizadas por los compiladores en el proceso de evaluar expresiones y de generar código de lenguaje máquina. En los ejercicios se exploran varias aplicaciones de las pilas.

12.6 Colas de espera

Otra estructura de datos común es la *cola de espera*. Una cola de espera es similar a una línea de pagos en un supermercado —la primera persona en la línea es atendida primero, y los otros clientes entran en la línea únicamente por la parte final y esperan para ser atendidos. Los nodos de la cola son eliminados sólo de la parte delantera o *cabeza* de la cola, y son incluidos o insertos únicamente en la *parte trasera* de la cola. Por esta razón, una cola se conoce como una estructura de datos de *primeras entradas, primeras salidas (FIFO por first-in, first-out)*. Las operaciones de insertar y de retirar se conocen como *enqueue* y *dequeue*.

Las colas tienen muchas aplicaciones en sistemas de cómputo. Muchas computadoras tienen únicamente un solo procesador, de tal forma que sólo un usuario puede ser servido a la vez. Las entradas de los demás usuarios son colocados en una cola. Cada entrada avanza en forma gradual hacia el frente de la cola, conforme los usuarios reciben servicio. La entrada que aparece en la parte delantera de la cola es la siguiente a recibir servicio.

Las colas también se utilizan para apoyar colas de impresión. Un entorno de multiusuario pudiera tener una sola impresora. Muchos usuarios podrían estar generando salidas para impresión. Si la impresora está ocupada, aún así se pueden generar otras salidas. Estas quedan en "espera" en el disco, donde esperan en una cola hasta que la impresora quede disponible.

En las redes de computadoras los paquetes de información también esperan en colas. Cada vez que un paquete llega a un nodo de red, debe ser encaminado al siguiente nodo de red, siguiendo una trayectoria hacia el destino final del paquete. El nodo de encaminamiento envía un paquete a la vez, por lo que los paquetes adicionales quedan en cola, hasta que el encaminador pueda enviarlos. En la figura. 12.12 se ilustra una cola con varios nodos. Note los apuntadores a la cabeza y a la parte trasera de la cola.

Error común de programación 12.7

No definir a **NULL** *el enlace en el último nodo de una cola.*

El programa de la figura 12.13 (cuya salida aparece en la figura 12.14) ejecuta manipulaciones de colas. El programa presenta varias opciones: insertar un nodo en la cola (función **enqueue**), eliminar o retirar un nodo de la cola (función **dequeue**) y terminar el programa.

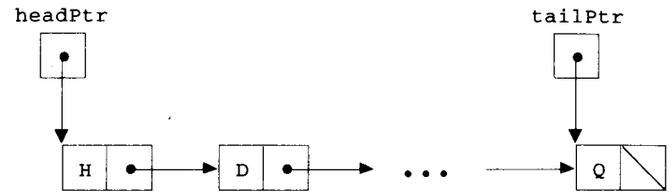


Fig. 12.12 Representación gráfica de una cola.

```
/* Operating and maintaining a queue */

#include <stdio.h>
#include <stdlib.h>

struct queueNode {  /* self-referential structure */
   char data;
   struct queueNode *nextPtr;
};

typedef struct queueNode QUEUENODE;
typedef QUEUENODE *QUEUENODEPTR;

/* function prototypes */
void printQueue(QUEUENODEPTR);
int isEmpty(QUEUENODEPTR);
char dequeue(QUEUENODEPTR *, QUEUENODEPTR *);
void enqueue(QUEUENODEPTR *, QUEUENODEPTR *, char);
void instructions(void);

main()
{
   QUEUENODEPTR headPtr = NULL, tailPtr = NULL;
   int choice;
   char item;

   instructions();
   printf("? ");
   scanf("%d", &choice);

   while (choice != 3) {

      switch(choice) {

         case 1:
            printf("Enter a character: ");
            scanf("\n%c", &item);
            enqueue(&headPtr, &tailPtr, item);
            printQueue(headPtr);
            break;
```

Fig. 12.13 Procesamiento de una cola (parte 1 de 3).

```
            case 2:
               if (!isEmpty(headPtr)) {
                  item = dequeue(&headPtr, &tailPtr);
                  printf("%c has been dequeued.\n", item);
               }

               printQueue(headPtr);
               break;

            default:
               printf("Invalid choice.\n\n");
               instructions();
               break;
         }

         printf("? ");
         scanf("%d", &choice);
      }

      printf("End of run.\n");
      return 0;
   }

   void instructions(void)
   {
      printf ("Enter your choice:"
              "   1 to add an item to the queue"
              "   2 to remove an item from the queue"
              "   3 to end");
   }

   void enqueue(QUEUENODEPTR *headPtr, QUEUENODEPTR *tailPtr,
                char value)
   {
      QUEUENODEPTR newPtr;

      newPtr = malloc(sizeof(QUEUENODE));

      if (newPtr != NULL) {
         newPtr->data = value;
         newPtr->nextPtr = NULL;

         if (isEmpty(*headPtr))
            *headPtr = newPtr;
         else
            (*tailPtr)->nextPtr = newPtr;

         *tailPtr = newPtr;
      }
      else
         printf("%c not inserted. No memory available.\n", value);
   }
```

Fig. 12.13 Procesamiento de una cola (parte 2 de 3).


```
char dequeue(QUEUENODEPTR *headPtr, QUEUENODEPTR *tailPtr)
{
   char value;
   QUEUENODEPTR tempPtr;

   value = (*headPtr)->data;
   tempPtr = *headPtr;
   *headPtr = (*headPtr)->nextPtr;

   if (*headPtr == NULL)
      *tailPtr = NULL;

   free(tempPtr);
   return value;
}

int isEmpty(QUEUENODEPTR headPtr)
{
   return headPtr == NULL;
}

void printQueue(QUEUENODEPTR currentPtr)
{
   if (currentPtr == NULL)
      printf("Queue is empty.\n\n");
   else {
      printf("The queue is:\n");

      while (currentPtr != NULL) {
         printf("%c --> ", currentPtr->data);
         currentPtr = currentPtr->nextPtr;
      }

      printf("NULL\n\n");
   }
}
```

Fig. 12.13 Procesamiento de una cola (parte 3 de 3).

La función **enqueue** recibe desde **main** tres argumentos: la dirección del apuntador a la cabeza de la cola, la dirección del apuntador a la parte trasera de la cola, y el valor a ser inserto en la cola. La función está formada de tres pasos:

1) Para crear un nuevo nodo: llama **malloc**, asigna a **newPtr** la posición asignada de memoria, asigna el valor que se va a insertar en la cola a **newPtr->data** y asigna **NULL** a **newPtr->nextPtr**.
2) Si la cola está vacía, asigna **newPtr** a ***headPtr**; de no ser así, asigna el apuntador **newPtr** a **(*tailPtr)->nextPtr**.
3) Asigna **newPtr** a ***tailPtr**.

La figura 12.15 ilustra una operación **enqueue**. La parte a) de la figura muestra la cola y el nuevo nodo antes de la operación. Las flechas con líneas punteadas de la parte b) ilustran los pasos 2 y 3 de la función **enqueue** que permiten añadir un nuevo nodo al final de una cola que no esté vacía.

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 1
Enter a character: A
The queue is:
A --> NULL

? 1
Enter a character: B
The queue is:
A --> B --> NULL

? 1
Enter a character: C
The queue is:
A --> B --> C --> NULL

? 2
A has been dequeued.
The queue is:
B --> C --> NULL

? 2
B has been dequeued.
The queue is:
C --> NULL

? 2
C has been dequeued.
Queue is empty.

? 2
Queue is empty.

? 4
Invalid choice.

Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 3
End of run.
```

Fig. 12.14 Salida de muestra del programa 12.13.

La función **dequeue** recibe como argumentos la dirección del apuntador a la cabeza de la cola y la dirección del apuntador a la parte trasera de la cola, y retira el primer nodo de la cola. La operación **dequeue** consiste de seis pasos:

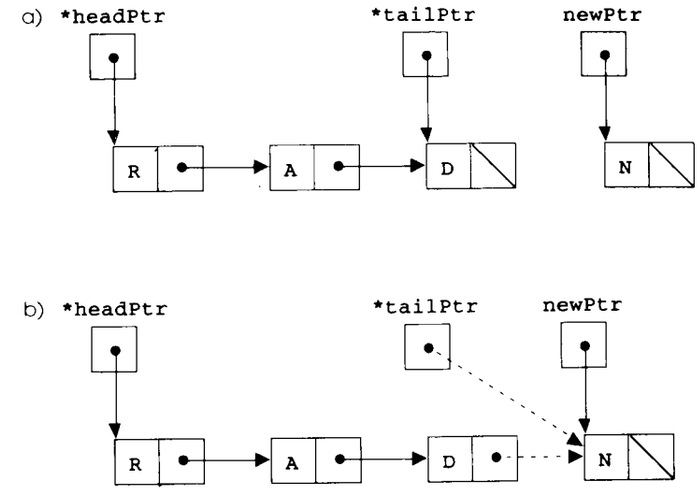


Fig. 12.15 Representación gráfica de la operación **enqueue**.

1) Asigna **(*headPtr)->data** a **value** (guarda los datos)
2) Asigna ***headPtr** a **tempPtr** (**tempPtr** es utilizado para liberar, mediante **free**, la memoria no necesaria).
3) Asigna **(*headPtr)->nextPtr** a ***headPtr** (***headPtr** apunta ahora al primer nodo en la cola).
4) Si ***headPtr** es **NULL**, asigna **NULL** a ***tailPtr**
5) Libera la memoria a la cual apunta **tempPtr**.
6) Regresa **value** al llamador (la función de **dequeue** es llamada desde **main** en el programa de la figura 12.13).

La figura 12.16 ilustra la función **dequeue**. La parte a) muestra la cola antes de la operación **enqueue** anterior. La parte b) muestra **tempPtr** apuntando al nodo a retirarse (dequeue), y **headPtr** apuntando al nuevo primer nodo de la cola. La función **free** es utilizada para recuperar la memoria a la cual **tempPtr** apunta.

12.7 Arboles

Las listas enlazadas, las pilas y las colas son *estructuras lineales de datos*. Un árbol es una estructura no lineal y de dos dimensiones de datos, con propiedades especiales. Los nodos de los árboles contienen dos o más enlaces. Esta sección analiza los *árboles binarios* (figura 12.17) —árboles cuyos nodos todos ellos contienen dos enlaces (ninguno, uno, o ambos de los cuales pudieran ser **NULL**). El *nodo raíz* es el primer nodo de un árbol. Cada enlace en el nodo raíz se refiere a un *hijo*. El *hijo izquierdo* es el primer nodo en el *subárbol izquierdo* y el *hijo derecho* es el primer nodo en el *subárbol izquierdo*. Los hijos de un nodo se conocen como *descendientes*. Un nodo sin hijos se conoce como *nodo de hoja*. Los científicos de la computación normalmente dibujan los árboles partiendo del nodo raíz hacia abajo —en forma exactamente opuesta a los árboles en la naturaleza.

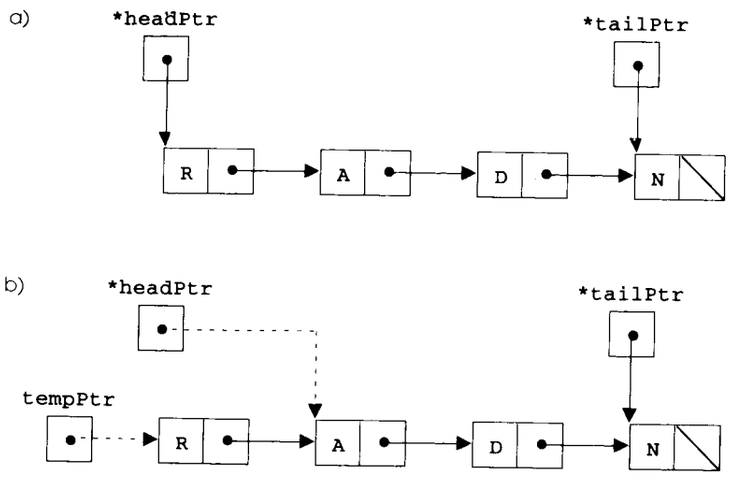


Fig. 12.16 Representación gráfica de la operación **dequeue**.

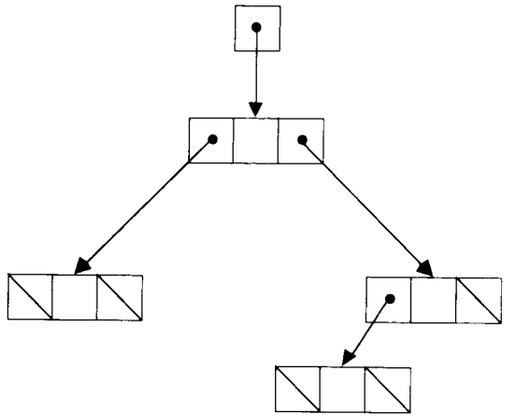


Fig. 12.17 Representación gráfica de un árbol binario.

En esta sección, se creará un árbol binario especial conocido como un *árbol de búsqueda binario*. Un árbol de búsqueda binario (que no tiene valores duplicados de nodos) tienen la característica que los valores en cualquier subárbol izquierdo son menores que el valor en sus nodos padre, y los valores en cualquier subárbol derecho son mayores que el valor en sus nodos padre. En la figura 12.18 se ilustra un árbol de búsqueda binario con 12 valores. Note que la forma del árbol de búsqueda binario que corresponde a un conjunto de datos puede variar, dependiendo del orden en el cual los valores están insertos dentro del árbol.

Error común de programación 12.8

No definir a `NULL` *los enlaces en los nodos de hoja de un árbol.*

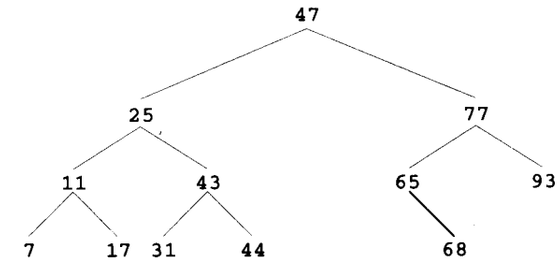


Fig. 12.18 Un árbol de búsqueda binario.

El programa de la figura 12.19 (cuya salida se muestra en la figura 12.20) crea un árbol de búsqueda binario y lo recorre de tres formas —*enorden, preorden* y *postorden*. El programa genera 10 números aleatorios e inserta cada uno de ellos en el árbol, a excepción de los valores duplicados, que son descartados.

Las funciones utilizadas en la figura 12.19, para crear un árbol de búsqueda binario y recorrer el árbol, son recursivas. La función **`insertNode`** recibe como argumentos la dirección del árbol y un entero para almacenarse en el árbol. *En un árbol de búsqueda binario un nodo puede ser únicamente inserto como nodo de hoja.* Los pasos para insertar un nodo en un árbol de búsqueda binario, son como sigue:

1) Si **`*treePtr`** es **`NULL`**, crea un nuevo nodo. Llame **`malloc`**, asigna la memoria asignada a **`*treePtr`**, asigna a **`(*treePtr)->data`** el entero a almacenarse, asigna a **`(*treePtr)->leftPtr`** y **`(*treePtr)->rightPtr`** el valor **`NULL`** y devuelve o regresa el control al llamador (ya sea a **`main`** o a una llamada anterior a **`insertNode`**).
2) Si el valor de **`*treePtr`** no es **`NULL`**, y el valor a insertarse es menor que **`(*treePtr)->data`**, se llama a la función **`insertNode`** con la dirección de **`(*treePtr)->leftPtr`**. De no ser así, se llama a la función **`insertNode`** con la dirección de **`(*treePtr)->rightPtr`**. Se continúan los pasos recursivos hasta que se encuentre un apuntador **`NULL`**, entonces se ejecutará el paso 1) para insertar el nuevo nodo.

Las funciones **`inOrder`**, **`preOrder`** y **`postOrder`** cada una de ellas recibe un árbol (es decir, el apuntador al nodo raíz del árbol) y recorren el árbol.

Los pasos para un recorrido **`inorder`** son:

1) Recorrer el subárbol izquierdo **`inOrder`**.
2) Procesar el valor en el nodo.
3) Recorrer el subárbol derecho **`inOrder`**.

El valor en un nodo no es procesado en tanto no sean procesados los valores de su subárbol izquierdo. El recorrido **`inOrder`** del árbol en la figura 12.21 es

```
6 13 17 27 33 42 48
```

Note que el recorrido **`inOrder`** de un árbol de búsqueda binario imprime los valores de nodo en valor ascendente. El proceso de crear un árbol de búsqueda binario, de hecho ordena los datos —y por lo tanto este proceso se llama la *clasificación de árbol binario*.

```
/* Create a binary tree and traverse it
   preorder, inorder, and postorder */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct treeNode {
   struct treeNode *leftPtr;
   int data;
   struct treeNode *rightPtr;
};

typedef struct treeNode TREENODE;
typedef TREENODE *TREENODEPTR;

void insertNode(TREENODEPTR *, int);
void inOrder(TREENODEPTR);
void preOrder(TREENODEPTR);
void postOrder(TREENODEPTR);

main()
{
   int i, item;
   TREENODEPTR rootPtr = NULL;

   srand(time(NULL));

   /* attempt to insert 10 random values betwenn 0 and 14 in the tree */
   printf("The numbers being placed in the tree are:\n");

   for (i = 1; i <= 10; i++) {
      item = rand() % 15;
      printf("%3d", item);
      insertNode(&rootPtr, item);
   }

   /* traverse the tree preOrder */
   printf("\n\nThe preOrder traversal is:\n");
   preOrder(rootPtr);

   /* traverse the tree inOrder */
   printf("\n\nThe inOrder traversal is:\n");
   inOrder(rootPtr);

   /* traverse the tree postOrder */
   printf("\n\nThe postOrder traversal is:\n");
   postOrder(rootPtr);

   return 0;
}
```

Fig. 12.19 Cómo crear y recorrer un árbol binario (parte 1 de 2).


```
void insertNode(TREENODEPTR *treePtr, int value)
{
   if (*treePtr == NULL) {   /* *treePtr is NULL */
      *treePtr = malloc(sizeof(TREENODE));

      if (*treePtr != NULL) {
         (*treePtr)->data = value;
         (*treePtr)->leftPtr = NULL;
         (*treePtr)->rightPtr = NULL;
      }
      else
         printf("%d not inserted. No memory available.\n",
                value);

   }
   else
      if (value < (*treePtr)->data)
         insertNode(&((*treePtr)->leftPtr), value);
      else
         if (value > (*treePtr)->data)
            insertNode(&((*treePtr)->rightPtr), value);
         else
            printf("dup");
}

void inOrder(TREENODEPTR treePtr)
{
   if (treePtr != NULL) {
      inOrder(treePtr->leftPtr);
      printf("%3d", treePtr->data);
      inOrder(treePtr->rightPtr);
   }
}

void preOrder(TREENODEPTR treePtr)
{
   if (treePtr != NULL) {
      printf("%3d", treePtr->data);
      preOrder(treePtr->leftPtr);
      preOrder(treePtr->rightPtr);
   }
}

void postOrder(TREENODEPTR treePtr)
{
   if (treePtr != NULL) {
      postOrder(treePtr->leftPtr);
      postOrder(treePtr->rightPtr);
      printf("%3d", treePtr->data);
   }
}
```

Fig. 12.19 Cómo crear y recorrer un árbol binario (parte 2 de 2).

```
The numbers being placed in the tree are:
  7  8  0  6 14  1  0dup 13  0dup  7dup

The preOrder traversal is:
  7  0  6  1  8 14 13

The inOrder traversal is:
  0  1  6  7  8 13 14

The postOrder traversal is:
  1  6  0 13 14  8  7
```

Fig. 12.20 Salida de muestra correspondiente al programa de la figura 12.19.

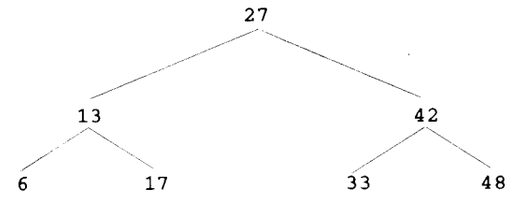


Fig. 12.21 Un árbol de búsqueda binario.

Los pasos para un recorrido **preOrder** son:

1) Procesar el valor en el nodo.
2) Recorrer el subárbol izquierdo **preOrder**.
3) Recorrer el subárbol derecho **preOrder.**

El valor en cada nodo es procesado conforme se pasa por cada nodo. Después de que se procese el valor en un nodo dado, son procesados los valores del subárbol izquierdo, y a continuación los valores en el subárbol derecho. El recorrido **preOrder** del árbol en la figura 12.21 es:

```
27 13 6 17 42 33 48
```

Los pasos para un recorrido **postOrder** son:

1) Recorrer el subárbol izquierdo **postOrder**.
2) Recorrer el subárbol derecho **postOrder**.
3) Procesar el valor en el nodo.

El valor en cada nodo no se imprime hasta que sean impresos los valores de sus hijos. El recorrido **postOrder** del árbol de la figura 12.21 es:

```
6 17 13 33 48 42 27
```


El árbol de búsqueda binario facilita la eliminación de duplicados. Conforme se crea el árbol, cualquier intento para insertar un valor duplicado será detectado porque en cada una de las comparaciones un duplicado seguirá las mismas decisiones "ir a la izquierda" o "ir a la derecha" que utlizó el valor original. Entonces, el duplicado eventualmente será comparado con un nodo que contenga el mismo valor. Llegado a este punto el valor duplicado pudiera simplemente ser descartado.

También es rápido buscar en un árbol binario un valor que coincida con un valor clave. Si el árbol es denso, entonces cada nivel contendrá aproximadamente dos veces tantos elementos como el nivel anterior. Por lo tanto un árbol de búsqueda binario con n elementos tendría un máximo de $\log_2 n$ (logaritmo de base 2 de n niveles) y, por lo tanto, tendrían que efectuarse un máximo $\log_2 n$ de comparaciones, ya sea para encontrar una coincidencia, o para determinar que no existe ninguna. Esto significa, por ejemplo, que al buscar en un árbol de búsqueda binario de 1000 elementos (densamente empacados), no es necesario llevar a cabo más de 10 comparaciones, porque $2^{10} > 1000$. Al buscar en un árbol de búsqueda binario de 1,000,000 (densamente empacados), no es necesario llevar a cabo más de 20 comparaciones, porque $2^{20} > 1{,}000{,}000$.

En los ejercicios se presentan algoritmos para otras operaciones de árboles de búsqueda binarios, como es borrar un elemento de un árbol binario, imprimir un árbol binario en un formato de árbol de dos dimensiones, y ejecutar un recorrido en orden de niveles de un árbol binario. El recorrido de orden de niveles de un árbol binario pasa por los nodos de un árbol, renglón por renglón, empezando por el nivel del nodo raíz. En cada uno de los niveles de árbol, se pasa por los nodos de izquierda a derecha. Otros ejercicios de árboles binarios incluyen permitir que un árbol de búsqueda binario contenga valores duplicados, insertar valores de cadenas en un árbol binario, y determinar cuántos niveles están incluidos en un árbol binario.

Resumen

- Las estructuras autorreferenciadas contienen miembros conocidos como enlaces, que apuntan a estructuras del mismo tipo de estructura.
- Las estructuras autorreferenciadas permiten que se enlacen muchas estructuras juntas en pilas, colas, listas y árboles.
- La asignación dinámica de memoria reserva un bloque de bytes en la memoria para almacenar un objeto de datos durante la ejecución del programa.
- La función **malloc** toma como argumento un número de bytes a asignarse, y regresa un apuntador **void** a la memoria asignada. La función **malloc** se utiliza normalmente junto con el operador **sizeof**. El operador **sizeof** determina el tamaño en bytes de la estructura para la cual se está asignando memoria.
- La función **free** cancela la asignación de memoria.
- Una lista enlazada es una colección de datos almacenados en un grupo de estructuras autorreferenciadas conectadas.
- Una lista enlazada es una estructura dinámica de datos —la longitud de la lista puede aumentarse o reducirse conforme sea necesario.
- Las listas enlazadas pueden continuar creciendo en tanto exista memoria disponible.
- Las listas enlazadas proporcionan un mecanismo para insertar y borrar datos en forma simple mediante la reasignación de apuntadores.

- Las pilas y las colas son versiones especializadas de una lista enlazada.
- Se añaden nuevos nodos a una pila y son retirados nodos de una pila únicamente de su parte superior. Por esta razón, se conoce una pila como una estructura de datos de últimas entradas, primeras salidas (LIFO).
- El miembro de enlace en el último nodo de la pila se define a **NULL** para indicar la parte inferior de la pila.
- Las dos operaciones primarias utilizadas para manipular una pila son **push** y **pop**. La operación **push** crea un nuevo nodo y lo coloca en parte superior de la pila. La operación **pop** retira un nodo de la parte superior de la pila, libera la memoria que estaba asignada al nodo retirado, y regresa el valor retirado.
- En una estructura de datos de cola, los nodos son retirados de la cabeza y añadidos a la parte trasera. Por esta razón, una cola se conoce como una estructura de datos de primeras entradas, primeras salidas (FIFO). Las operaciones de añadir y de retirar se conocen como **enqueue** y **dequeue**.
- Los árboles son estructuras de datos más complejas que las listas enlazadas, las colas y las pilas. Los árboles son estructuras de datos de dos dimensiones, que requieren de dos o más enlaces por nodo.
- Los árboles binarios contienen dos enlaces por nodo.
- El nodo raíz es el primer nodo en el árbol.
- Cada uno de los apuntadores en el nodo raíz se refiere a un hijo. El hijo izquierdo es el primer nodo en el subárbol izquierdo, y el hijo derecho es el primer nodo en el subárbol derecho. Los hijos de un nodo se conocen como descendientes. Si un nodo no tiene ningún descendiente, se le conoce como un nodo de hoja.
- Un árbol de búsqueda binario tiene la característica que tiene el valor en el hijo izquierdo del nodo menor que el valor del nodo padre, y el valor en el hijo derecho de un nodo es mayor o igual que el valor del nodo padre. Si se puede determinar que no existen valores de datos duplicados, simplemente el valor en el hijo derecho es mayor que el valor en el nodo padre.
- Un recorrido enorden de un árbol binario recorre enorden el subárbol izquierdo, procesa el valor en el nodo, y recorre enorden el subárbol derecho. El valor en un nodo no será procesado hasta que hayan sido procesados los valores en su subárbol izquierdo.
- Un recorrido en preorden procesa el valor en el nodo, recorre en preorden el subárbol izquierdo, y recorre en preorden el subárbol derecho. El valor en cada nodo se procesa conforme se encuentra cada nodo.
- Un recorrido en postorden recorre en postorden el subárbol izquierdo, recorre en postorden el subárbol derecho, y procesa el valor en el nodo. El valor en cada nodo no es procesado hasta que sean procesados los valores en ambos de sus subárboles.

Terminología

árbol de búsqueda binaria
árbol binario
clasificación de árbol binario
nodo hijo
descendientes
borrar un nodo
dequeue
doble indirección
estructuras dinámicas de datos
asignación dinámica de memoria

enqueue
FIFO (primeras entradas, primeras salidas)
free
cabeza de una cola
recorrido enorden
inserción de un nodo
nodo de hoja
hijo izquierdo
subárbol izquierdo
LIFO (últimas entradas, primeras salidas)
estructura lineal de datos
lista enlazada
malloc (asignar memoria)
nodo
estructura de datos no lineal
apuntador **NULL**
nodo padre
apuntador a un apuntador
pop
recorrido postorden
función predicada
recorrido preorden
push
cola
hijo derecho
subárbol derecho
nodo raíz
estructura autorreferenciada
descendencias
sizeof
pila
subárbol
parte trasera de una cola
parte superior
recorrido
árbol
pasar por un nodo

Errores comunes de programación

12.1 No establecer a **NULL** el enlace en el último nodo de una lista.
12.2 Suponer que el tamaño de una estructura es simplemente la suma de los tamaños de sus miembros.
12.3 No regresar memoria dinámicamente asignada cuando ésta ya no es necesaria, puede hacer que el sistema se quede sin memoria prematuramente. Esto se conoce a veces como "fuga de memoria".
12.4 Utilizando **malloc**, liberar memoria no dinámicamente asignada.
12.5 Referirse a memoria que ya ha sido liberada.
12.6 No definir a **NULL** el enlace en el nodo inferior de una pila.
12.7 No definir a **NULL** el enlace en el último nodo de una cola.
12.8 No definir a **NULL** los enlaces en los nodos de hoja de un árbol.

Prácticas sanas de programación

12.1 Para determinar el tamaño de una estructura utilice el operador **sizeof**.
12.2 Al utilizar **malloc**, compruebe si es **NULL** el valor de regreso de apuntador. Si la memoria solicitada no ha sido asignada imprima un mensaje de error.
12.3 Cuando ya no se requiera memoria que fue dinámicamente asignada, utilice **free**, para regresar esta memoria inmediatamente al sistema.
12.4 Asigne **NULL** al miembro de enlace de un nuevo nodo. Los apuntadores deben ser inicializados antes de ser utilizados.

Sugerencias de rendimiento

12.1 Podría declararse un arreglo que contenga más elementos que el número esperado de elementos de datos, pero esto puede desperdiciar memoria. En estas situaciones las listas enlazadas pueden obtener una mejor utilización de la memoria.
12.2 Puede resultar muy tardado insertar y eliminar en un arreglo ya ordenado —deberán ser desplazados en forma apropiada. Todos los elementos que sigan al elemento insertado o borrado.

12.3 Los elementos de un arreglo se almacenan en forma contigua en memoria. Esto permite acceso inmediato a cualquier arreglo del elemento, porque la dirección de cualquier elemento puede ser calculada directamente, basada en su posición en relación con el principio del arreglo. Las listas enlazadas no proporcionan un acceso inmediato como éste a sus elementos.

12.4 Tratándose de estructuras de datos que crecen o se reducen en tiempo de ejecución, es posible ahorrar memoria utilizando asignación dinámica de memoria (en vez de los arreglos). Recuerde, sin embargo, que los apuntadores toman espacio, y que la asignación dinámica de memoria incurre en sobrecarga por las llamadas de función.

Sugerencia de portabilidad

12.1 El tamaño de una estructura no es necesariamente la suma de los tamaños de sus miembros. Esto es debido a varios requisitos de alineación de límites, que son dependientes de la máquina (vea el capítulo 10).

Ejercicios de autoevaluación

12.1 Llene cada uno de los siguientes espacios vacíos:

a) Una estructura auto_____ se utiliza para formar estructuras dinámicas de datos.
b) La función _____ se utiliza para asignar dinámicamente la memoria.
c) Una _____ es una versión especializada de una lista enlazada, en la cual los nodos pueden ser insertos y borrados únicamente a partir del principio de la lista.
d) Las funciones que no modifican una lista enlazada, pero que simplemente analizan la lista se conocen como _____.
e) Una cola de espera se conoce como una estructura de datos _____ porque los primeros nodos insertos son los primeros nodos retirados.
f) El apuntador al siguiente nodo en una lista enlazada se conoce como un _____.
g) La función _____ se utiliza para recuperar memoria dinámicamente asignada.
h) Una _____ es una versión especializada de una lista enlazada en la cual los nodos pueden ser insertos únicamente al principio de la lista y retirados o borrados únicamente de la parte final de la lista.
i) Una _____ es una estructura de datos de dos dimensiones no lineal, que contiene nodos con dos o más enlaces.
j) Una pila se conoce como una estructura de datos _____ porque el último nodo insertado es el primer nodo retirado.
k) Los nodos de un árbol _____ contienen dos miembros enlazados.
l) El primer nodo de un árbol es el nodo _____.
m) Cada enlace en un nodo de árbol apunta a un _____ o a un _____ de dicho nodo.
n) Un nodo de árbol que no tiene hijos se conoce como un nodo _____.
o) Los tres algoritmos de recorrido para un árbol binario son _____, _____ y _____.

12.2 ¿Cuáles son las diferencias entre una lista enlazada y una pila?

12.3 ¿Cuáles son las diferencias entre una pila y una cola de espera?

12.4 Escriba un enunciado, o un conjunto de enunciados para llevar a cabo cada uno de los siguientes. Suponga que todas las manipulaciones ocurren en `main` (por lo tanto, no son necesarias direcciones a variables de apuntador) y suponga las definiciones siguientes:

```
struct gradeNode {
   char lastName[20];
   float grade;
   struct gradeNode *nextPtr;
};
```



```
typedef struct gradeNode GRADENODE;
typedef GRADENODE *GRADENODEPTR;
```

a) Crea un apuntador al principio de la lista llamado `startPtr`. La lista está vacía.
b) Crea un nuevo nodo del tipo `GRADENODE` al cual apunta el apuntador `newPtr` del tipo `GRADENODEPTR`. Asigne la cadena `"Jones"` al miembro `LastName` y el valor `91.5` al miembro `grade` (utilice `strcpy`). Incluya todas las declaraciones y enunciados necesarios.
c) Suponga que la lista a la cual apunta `startPtr` está formada actualmente de dos nodos —uno que contiene `"Jones"` y uno que contiene `"Smith"`. Los nodos están en orden alfabético. Proporcione los enunciados necesarios para insertar otros nodos que contengan los siguientes datos para `lastName` y `grade`:

```
"Adams"      85.0
"Thompson"   73.5
"Pritchard"  66.5
```

Utilice los apuntadores `previousPtr`, `currentPtr` y `newPtr` para llevar a cabo las inserciones. Declare hacia donde apuntan `previousPtr`, y `currentPtr` antes de cada inserción. Suponga que `newPtr` siempre apunta al nuevo nodo, y que el nuevo nodo ya ha sido asignado con los datos.
d) Escriba un ciclo `while` que imprima los datos de cada nodo de la lista. Utilice el apuntador `currentPtr` para pasar a lo largo de la misma.
e) Escriba un ciclo `while` que borre todos los nodos en la lista y que libere la memoria asociada con cada uno de los nodos. Utilice el apuntador `currentPtr` y el apuntador `tempPtr` para caminar a lo largo de la lista y liberar memoria respectivamente.

12.5 Proporcione manualmente los recorridos enorden, preorden y postorden del árbol de búsqueda binario de la Figura 12.22.

Respuestas a los ejercicios de autoevaluación

12.1 a) referenciados. b) `malloc`. c) pila. d) predicados. e) FIFO. f) enlace. g) `free`. h)cola de espera. i) árbol. j) LIFO. k) binario. l) raíz. m) hijo o subárbol. n) hoja. o) enorden, preorden y postorden.

12.2 Es posible insertar un nodo en cualquier parte de una lista enlazada, y eliminar un nodo de cualquier parte de una lista enlazada. Sin embargo, los nodos en una pila pueden únicamente ser insertos en la parte superior de la misma y eliminados también de la parte superior de la misma.

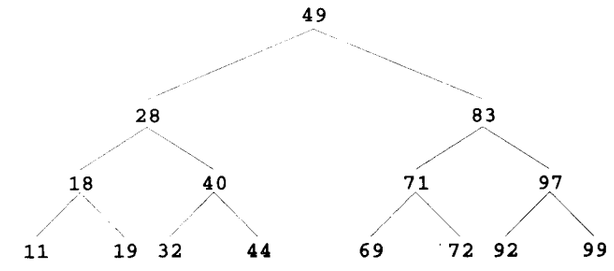


Fig. 12.22 Un árbol de búsqueda binario de 15 nodos.

12.3 Una cola tiene apuntadores tanto a su cabeza como a su parte trasera, de tal forma que los nodos pueden ser insertos en la parte trasera y borrados de la cabeza. Una pila tiene un solo apuntador a la parte superior de la misma donde se ejecuta tanto la inserción como el borrado de los nodos.

12.4 a)
```
GRADENODEPTR startPtr = NULL;
```
b)
```
GRADENODEPTR newPtr;
newPtr = malloc(sizeof(GRADENODE));
strcpy(newPtr->lastName, "Jones");
newPtr->grade = 91.5;
newPtr->nextPtr = NULL
```
c) Para insertar `"Adams"`;
`previousPtr` es `NULL`, `current Ptr` apunta al primer elemento en la lista.
```
newPtr->nextPtr = currentPtr;
startPtr = newPtr;
```
Para insertar `"Thompson"`:
`previousPtr` apunta al último elemento en la lista (que contiene `"Smith"`)
`current Ptr` es `NULL`
```
newPtr->nextPtr = currentPtr;
previousPtr->nextPtr = newPtr;
```
Para insertar `"Pritchard"`
`previousPtr` apunta al nodo que contiene `"Jones"`
`currentPtr` apunta al nodo que contiene `"Smith"`.
```
newPtr->nextPtr = currentPtr;
previousPtr->nextPtr = newPtr;
```
d)
```
current Ptr = startPtr;
while (current Ptr != NULL) {
   printf("Lastname = %s\nGrade = %6.2f\n",
          currentPtr->lastName, currentPtr->grade);
   currentPtr = current Ptr->nextPtr;
}
```
e)
```
currentPtr = startPtr;
while (current Ptr != NULL) {
   tempPtr = currentPtr;
   currentPtr = currentPtr->nextPtr;
   free(tempPtr);
}
startPtr = NULL;
```

12.5 El recorrido enorden es:

```
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99
```

El recorrido preorden es:

```
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99
```

El recorrido postorden es:

```
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49
```

Ejercicios

12..6 Escriba un programa que concatene dos listas enlazadas de caracteres. El programa deberá incluir la función **concatenate** que toma como argumentos apuntadores a ambas listas y que concatena la segunda lista a la primera.

12.7 Escriba un programa que combine dos listas ordenadas de enteros en una sola lista ordenada de enteros. La función **merge** deberá recibir apuntadores al primer nodo de cada una de las listas a combinarse, y deberá regresar un apuntador al primer nodo de la lista combinada.

12.8 Escriba un programa que inserte 25 enteros al azar desde 0 hasta 100 en orden en una lista enlazada. El programa deberá calcular la suma de los elementos, y el promedio en punto flotante de los mismos.

12.9 Escriba un programa que origine una lista enlazada de 10 caracteres, y a continuación origine una copia de la lista de orden inverso.

12.10 Escriba un programa que introduzca una línea de texto y utilice una pila para imprimir la línea invertida.

12.11 Escriba un programa que utilice una pila para determinar si una cadena es un palíndromo (es decir, si la cadena se deletrea en forma idéntica hacia adelante y hacia atrás). El programa deberá ignorar espacios y puntuaciones.

12.12 Las pilas son utilizadas por los compiladores para auxiliarse en el proceso de evaluar expresiones y para generar código en lenguaje máquina. En este ejercicio y en el siguiente, investigamos cómo los compiladores evalúan expresiones aritméticas formadas únicamente por constantes, operadores y paréntesis.

Los seres humanos normalmente escriben las expresiones como `3 + 4` y `7 /9`, en el cual el operador (aquí `+` o `/`) se escribe entre sus operandos —esto se conoce como *notación infija*. Las computadoras "prefieren" la *notación postfija*, en la cual el operador se escribe a la derecha de sus dos operandos. Las expresiones infijas anteriores aparecerían en notación postfija como `3 4 +` y `7 9 /`, respectivamente.

Para evaluar una expresión infija compleja, un compilador primero convertiría la expresión a notación postfija, y a continuación evaluaría la versión postfija de la expresión. Cada uno de estos algoritmos requiere únicamente de una pasada de izquierda a derecha en la expresión. Cada algoritmo utiliza una pila para apoyarse en esta operación, y en cada uno la pila se utiliza para fines distintos.

En este ejercicio, usted escribirá una versión en C del algoritmo de conversión de infijos a postfijos. En el siguiente ejercicio, escribirá una versión en C del algoritmo de evaluación de la expresión postfija.

Escriba un programa que convierta una expresión infija aritmética ordinaria (suponga que se ha introducido una expresión válida) con enteros de un solo dígito, como

```
(6 + 2) * 5 - 8 / 4
```

a una expresión postfija. La expresión postfija de la expresión infija anterior es

```
6 2 + 5 * 8 4 / -
```

El programa debería leer la expresión al arreglo de caracteres **infix**, y utilizar las versiones modificadas de las funciones de pila puestas en práctica en este capítulo para crear la expresión postfija en el arreglo de caracteres **postfix**. El algoritmo para crear una expresión postfija es como sigue:

1) Inserte (push) un paréntesis izquierdo `'('` en la parte superior de la pila.
2) Agregue un paréntesis derecho `')'` al final de infix.
3) En tanto la pila no esté vacía, lea infix de izquierda a derecha y haga lo siguiente:
   Si el carácter actual en **infix** es un dígito, cópielo al siguiente elemento de **postfix**.
   Si el carácter actual en **infix** es un paréntesis izquierdo, insértelo (push) sobre la pila.
   Si el carácter actual en **infix** es un operador,
      Retire (pop) los operadores (si es que hay alguno) en la parte superior de la pila en tanto tengan precedencia igual o mayor que el operador actual, e inserte los operadores retirados en **postfix**.
   Inserte (push) el carácter actual en **infix** sobre la pila.

Si el carácter actual en `infix` es un paréntesis derecho
Retire (pop) los operadores de la parte superior de la pila e insértelos en `postfix` hasta que en la parte superior de la pila quede un paréntesis izquierdo.
Retire (pop) y descarte el paréntesis izquierdo de la pila.

Las siguientes operaciones aritméticas se permiten en una expresión:

+	adición
-	substracción
*	multiplicación
/	división
^	exponenciación
%	módulo

La pila deberá ser mantenida utilizando las siguientes declaraciones:

```
struct stackNode {
   char data;
   struct stackNode *nextPtr;
};
typedef struct stackNode STACKNODE;
typedef STACKNODE *STACKNODEPTR;
```

El programa deberá estar formado por `main` y 8 otras funciones, con los siguientes encabezados de función:

`void convertToPostfix(char infix[], char postfix[])`
Convierta la expresión infija a notación postfija.
`int isOperator(char c)`
Determinar si `c` es un operador.
`int precedence(char operator1, char operator2)`
Determinar si la precedencia de `operador1` es menor que, igual a, o mayor que la precedencia de `operador2`. La función regresa -1, 0 y 1, respectivamente.
`void push(STACKNODEPTR *topPtr, char value)`
Insertar (push) un valor en la pila.
`char pop(STACKNODEPTR *topPtr)`
Retirar (pop) un valor de la pila.
`char stackTop(STACKNODEPTR topPtr)`
Regresar el valor superior de la pila sin retirar nada de la pila.
`int isEmpty(STACKNODEPTR topPtr)`
Determinar si la pila está vacía.
`void printStack(STACKNODEPTR topPtr)`
Imprimir la pila.

12.13 Escriba un programa que evalúe una expresión postfija (suponga que es válida) como

```
6 2 + 5 * 8 4 / -
```

El programa deberá leer una expresión postfija formada de dígitos y de operadores a un arreglo de caracteres. Utilizando versiones modificadas de las funciones de pila puestas anteriormente en práctica en este capítulo, el programa deberá rastrear la expresión y evaluarla. El algoritmo es como sigue:

1) Agregue el carácter `NULL` (`'\0'`) al final de la expresión postfija. Cuando se encuentre el carácter `NULL`, ya no es necesario procesamiento adicional.
2) En tanto no se encuentre `'\0'`, lea la expresión de izquierda a derecha.
   Si el carácter actual es un dígito,
   Inserte (push) su valor entero en la pila (el valor entero de un carácter de dígito es su valor en el conjunto de caracteres de la computadora, menos el valor de `'0'` en el conjunto de caracteres de la computadora).

De lo contrario, si el carácter actual es un operador,
Retire (pop) los dos elementos superiores de la pila a las variables `x` e `y`.
Calcule `y operator x`
Inserte (push) el resultado del cálculo en la pila.
3) Cuando en la expresión se encuentre el carácter `NULL`, retire (pop) el valor superior de la pila. Ese es el resultado de la expresión postfija.

Nota: en 2) arriba, si el operador es `'/'`, la parte superior de la pila es `2`, y el siguiente elemento en la pila es `8`, entonces retira (pop) `2` a `x`, retira (pop) `8` a `y`, evalúa `8/2` e inserta (push) el resultado con `4`, de regreso a la pila. Esta nota también se aplica al operador `'-'`. Las operaciones aritméticas permitidas en una expresión son:

+	adición
-	substracción
*	multiplicación
/	división
^	exponenciación
%	módulo

La pila deberá ser mantenida utilizando las declaraciones siguientes:

```
struct stackNode {
   int data;
   struct stackNode *nextPtr;
};
typedef struct stackNode STACKNODE;
typedef STACKNODE *STACKNODEPTR;
```

El programa deberá consistir de `main` y de 8 otras funciones, con los siguientes encabezados de función:

`int evaluatePostfixExpression(char *expr)`
Evalúa la expresión postfija.
`int calculate(int op1, int op2, char operator)`
Evalúa la expresión `op1 operator op2`.
`void push(STACKNODEPTR *topPtr, int value)`
Insertar (push) un valor en la pila.
`int pop(STACKNODEPTR *topPtr)`
Retirar (pop) un valor de la pila.
`int isEmpty(STACKNODEPTR topPtr)`
Determinar si la pila está vacía.
`void printStack(STACKNODEPTR topPtr)`
Imprimir la pila.

12.14 Modifique el programa evaluador postfijo del ejercicio 12.13, de tal forma que pueda procesar operandos de enteros mayores que 9.

12.15 *(Simulación de supermercado).* Escriba un programa que simule una línea de caja en el supermercado. La línea es una cola de espera. Los clientes llegan a intervalos enteros al azar, de 1 a 4 minutos. También, cada cliente es atendido en intervalos enteros al azar, de entre 1 y 4 minutos. Obviamente, las velocidades necesitan equilibrarse. Si la velocidad de llegada promedio es mayor que la velocidad promedio de servicio, la cola crecerá en forma indefinida. Inclusive tratándose de velocidades equilibradas, el azar todavía puede causar líneas largas. Ejecute la simulación de supermercado para un día de 12 horas (720 minutos) utilizando el algoritmo siguiente:

1) Escoja un entero al azar entre 1 y 4 para determinar el minuto en el cual el primer cliente llega.
2) En el momento de llegada del primer cliente:
   Determine el tiempo de servicio para ese cliente (entero al azar de 1 a 4);
   Empiece a darle servicio al cliente;

programe el tiempo de llegada del siguiente cliente (un entero al azar de 1 a 4, añadido al tiempo actual).
3) Para cada minuto del día:
Si llega el siguiente cliente,
Póngalo en la cola;
Programe el tiempo de llegada del siguiente cliente;
Si para el último cliente el servicio ya fue terminado;
Dígalo así
Retire de la cola para darle servicio al cliente siguiente
Determine el tiempo de terminación del servicio del cliente (entero al azar de 1 a 4, añadido al tiempo actual).

Ahora ejecute su simulación durante 720 minutos y conteste cada una de las siguientes preguntas:

a) ¿Cuál es en cualquier momento el número máximo de clientes en la cola?
b) ¿Cuál es la espera más larga experimentada por un cliente?
c) ¿Qué pasa si el intervalo de llegadas es modificado de 1 a 4 minutos, a 1 a 3 minutos?

12.16 Modifique el programa de la figura 12.19 para permitir que el árbol binario contenga valores duplicados.

12.17 Escriba un programa basado en el programa de la figura 12.19 que introduzca una línea de texto, divida léxicamente la oración en palabras por separado, inserte las palabras en un árbol de búsqueda binaria, e imprima los recorridos enorden, preorden y postorden del árbol.

Sugerencia: lea la línea de texto a un arreglo. Utilice `strtok` para dividir léxicamente el texto. Una vez encontrada una división, genere un nuevo nodo en el árbol, asigne el apuntador que regresa `strtok` al miembro `string` del nuevo nodo, e inserte el nodo en el árbol.

12.18 En este capítulo, vimos al crear un árbol de búsqueda binario, que la eliminación de duplicados es sencilla. Describa cómo ejecutaría usted la eliminación de duplicados utilizando únicamente un arreglo de un solo subíndice. Compare el rendimiento de la eliminación de duplicados basado en arreglos, con el rendimiento de la eliminación de duplicados basado en el árbol de búsqueda binario.

12.19 Escriba una función `depth` que recibe un árbol binario y que determine cuántos niveles tiene.

12.20 *(Impresión recursiva de una lista en orden inverso).* Escriba una función `printListBackwards` que en forma recursiva extraiga los elementos en una lista en orden inverso. Utilice su función en un programa de prueba que origine una lista ordenada de enteros e imprima la lista en orden inverso.

12.21 *(Búsqueda de una lista en forma recursiva).* Escriba una función `searchList` que busque en una lista enlazada en forma recursiva, buscando un valor específico. Si éste es hallado la función deberá regresar un apuntador al valor; de lo contrario, `NULL` deberá ser regresado. Utilice su función en un programa de prueba que origine una lista de enteros. El programa deberá solicitar al usuario un valor a localizar dentro de la lista.

12.22 *(Borrado de árbol binario).* En este ejercicio, analizamos el borrado de elementos de árboles de búsqueda binarios. El algoritmo de borrado no es tan sencillo como el algoritmo de inserción. Al borrar un elemento se presentan tres casos distintos —el elemento está contenido en un nodo de hoja (es decir, no tiene hijo), el elemento está contenido en un nodo que tiene un hijo, o el elemento está contenido en un nodo que tiene dos hijos.

Si el elemento a borrarse está contenido en un nodo de hoja, el nodo se borra y el apuntador en el nodo padre se define a `NULL`.

Si el elemento a borrarse está contenido en un nodo con un hijo, el apuntador en el nodo padre se define para apuntar al nodo hijo y el nodo conteniendo el elemento de dato se borra. Esto hace que el nodo hijo tome el lugar del nodo borrado dentro del árbol.

El último caso es el más difícil. Cuando es borrado un nodo que tiene dos hijos, otro nodo dentro del árbol debe tomar su lugar. Sin embargo, el apuntador en el nodo padre no puede simplemente asignarse para apuntar a uno de los hijos del nodo a borrarse. En la mayor parte de los casos, el árbol de búsqueda binario no cumpliría con la característica siguiente de los árboles de búsqueda binarios: *los valores en cualquier subárbol izquierdo son menores que el valor en el nodo padre, y los valores en cualquier subárbol derecho son mayores que el valor en el nodo padre.*

¿Qué nodo se utilizará como *nodo de remplazo* para poder mantener esta característica? Ya sea el nodo que contenga el valor más grande en el árbol menor que el valor del nodo a borrarse, o el nodo que contenga el valor más pequeño en el árbol mayor que el valor del nodo a borrarse. Consideremos el nodo con el valor menor. En un árbol de búsqueda binario, el valor menor que el valor del padre se localiza en el subárbol izquierdo del nodo padre y se garantiza que está contenido en el nodo más a la derecha del subárbol. Este nodo se localiza dirigiéndose hacia abajo por el subárbol izquierdo, hacia la derecha, hasta que resulte `NULL` el apuntador hacia el hijo derecho del nodo actual. Estamos ahora apuntando al nodo de remplazo, que es o un nodo de hoja o un nodo con un hijo a su izquierda. Si el nodo de remplazo es un nodo de hoja, los pasos para llevar a cabo el borrado son como sigue:

1) Almacene el apuntador al nodo a borrarse en una variable de apuntador temporal (este apuntador se utiliza para liberar memoria dinámicamente asignada).
2) Defina el apuntador en el padre del nodo a borrarse, para que apunte al nodo de remplazo.
3) Defina a `NULL` el apuntador en el padre del nodo de remplazo.
4) Defina el apuntador al subárbol derecho, en el nodo de remplazo, para que apunte al subárbol derecho del nodo a borrarse.
5) Borre el nodo al cual apunta la variable de apuntador temporal.

Los pasos de borrado para un nodo de remplazo con un hijo izquierdo son similares a los de un nodo de remplazo sin hijos, pero el algoritmo también debe mover el hijo a la posición del nodo de remplazo en el árbol. Si el nodo de remplazo es un nodo con un hijo izquierdo, los pasos a llevarse a cabo para el borrado son como sigue:

1) Almacene en una variable de apuntador temporal el apuntador al nodo a borrarse.
2) Defina el apuntador en el padre del nodo a borrarse, para apuntar al nodo de remplazo.
3) Defina el apuntador en el padre del nodo de remplazo, para apuntar al hijo izquierdo del nodo de remplazo.
4) Defina el apuntador al subárbol derecho en el nodo de remplazo, para apuntar al subárbol derecho del nodo a borrarse.
5) Borre el nodo hacia el cual apunta la variable de apuntador temporal.

Escriba la función `deleteNode`, que toma como sus argumentos un apuntador al nodo raíz del árbol y el valor a borrarse. La función deberá localizar el nodo que contenga el valor a borrarse en el árbol y utilizar los algoritmos analizados aquí para borrarlo. Si no se encuentra el valor en el árbol, la función deberá imprimir un mensaje, que indique si se borra o no el valor. Modifique el programa de la figura 12.19 para utilizar esta función. Después de borrar un elemento, llame a las funciones de recorrido `inOrder`, `preOrder` y `postOrder` para confirmar que la operación de borrado fue correctamente ejecutada.

12.23 *(Búsqueda de árbol binario).* Escriba la función `binaryTreeSearch`, que intenta localizar un valor especificado en un árbol de búsqueda binario. La función deberá tomar como argumentos un apuntador al nodo raíz del árbol binario y una clave de búsqueda a localizar. Si se encuentra el nodo que contenga la clave de búsqueda, la función deberá regresar un apuntador a dicho nodo; de lo contrario, la función deberá regresar un apuntador `NULL`.

12.24 *(Recorrido de árbol binario en orden de nivel).* El programa de la figura 12.19 ilustró tres métodos recursivos de recorrer un árbol binario —recorrido enorden, recorrido preorden y recorrido postorden. En este ejercicio se presenta el *recorrido en orden de nivel* de un árbol binario, en el cual los valores de nodo se imprimen nivel por nivel, iniciándose en el nivel del nodo raíz. En cada nivel los nodos se imprimen de

izquierda a derecha. El recorrido en orden de nivel no es un algoritmo recursivo. Utiliza la estructura de datos de cola de espera para controlar la salida de los nodos. El algoritmo es como sigue:

1) Inserte en la cola de espera el nodo raíz.
2) En tanto existan nodos en la cola,
   Obtenga el nodo siguiente en la cola
   Imprima el valor del nodo
   Si no es `NULL` el apuntador al hijo izquierdo del nodo
      Inserte en la cola el nodo del hijo izquierdo
   Si no es `NULL` el apuntador al hijo derecho del nodo
      Inserte en la cola el nodo del hijo derecho.

Escriba la función `levelOrder` para ejecutar un recorrido en orden de nivel de un árbol binario. La función deberá tomar como argumento un apuntador al nodo raíz del árbol binario. Modifique el programa de la figura 12.19 para utilizar esa función. Compare la salida correspondiente a esa función con las salidas de otros algoritmos de recorrido, para ver si funcionó correctamente. (Nota: también será necesario modificar e incorporar dentro de este programa las funciones de procesamiento de colas de la figura 12.13.)

12.25 *(Impresión de árboles).* Escriba una función recursiva `outputTree` para desplegar en pantalla un árbol binario. La función deberá de extraer el árbol, renglón por renglón, con la parte superior del árbol en la parte izquierda de la pantalla, y la parte inferior del árbol hacia la derecha de la pantalla. Cada renglón es extraído en forma vertical. Por ejemplo, el árbol binario ilustrado en la figura 12.22 es extraído como sigue:

```
            99
      97
            92
   83
            72
      71
            69
49
            44
      40
            32
   28
            19
      18
            11
```

Note que el nodo de hoja más a la derecha aparece en la parte superior de la salida, en la columna más a la derecha y el nodo de raíz aparece a la izquierda de la salida. Cada columna extraída se inicia cinco espacios a la derecha de la columna anterior. La función `outputTree` debe recibir como argumentos un apuntador al nodo raíz del árbol y un entero `totalSpaces`, representando el número de espacios que anteceden al valor a ser extraído (esta variable deberá iniciarse en cero, de tal forma que el nodo raíz sea extraído en la parte izquierda de la pantalla). La función utiliza para sacar el árbol un recorrido enorden modificado —se inicia en el nodo más a la derecha en el árbol y funciona de regreso hacia el izquierdo. El algoritmo es como sigue:

En tanto no sea `NULL` el apuntador al nodo actual
   Llame recursivamente `outputTree` con el subárbol derecho del nodo actual y `totalSpace + 5`
   Utilice una estructura `for` para contar de 1 hasta `totalSpace` y extraiga los espacios.

   Extraiga el valor del nodo actual
   Defina el apuntador al nodo actual para que apunte al subárbol izquierdo del nodo actual.
   Incremente `totalSpaces` en 5.

Sección especial: Cómo construir su propio compilador

En los ejercicios 7.18 y 7.19 presentamos el lenguaje de máquina Simpletron (LMS) y creamos el simulador de computadora Simpletron para ejecutar programas escritos en LMS. En esta sección, construiremos un compilador que convierte a LMS programas escritos en un lenguaje de programación de alto nivel. Esta sección se "encadena" con todo el proceso de programación. Escribiremos programas en este nuevo lenguaje de alto nivel, compilaremos los programas en el compilador que construiremos, y ejecutaremos los programas en el simulador que construimos en el ejercicio 7.19.

12.26 *(El Lenguaje Simple).* Antes de que empecemos a construir el compilador, analizamos un lenguaje de alto nivel simple, aunque poderoso, similar a las versiones primeras del popular lenguaje BASIC. Llamamos el lenguaje *Simple*. Cada *enunciado* Simple consiste de un *número de línea* y de una *instrucción* Simple. Los números de línea deben aparecer en orden ascendente. Cada instrucción inicia con uno de los siguientes *comandos* Simple: `rem`, `input`, `let`, `print`, `goto`, `if/goto` o `end` (vea la figura 12.23). Todos los comandos, a excepción de `end`, pueden ser utilizados en forma repetida. Simple sólo evalúa expresiones enteras utilizando los operadores +, -, *, y /. Estos operadores tienen la misma precedencia que en C. Pueden utilizarse paréntesis para cambiar el orden de evaluación de una expresión.

Nuestro compilador Simple reconoce únicamente letras minúsculas. Todos los caracteres en un archivo Simple deberán ser minúsculas (las letras mayúsculas resultarán en un error de sintaxis, a menos de que aparezcan en un enunciado `rem`, en cuyo caso serán ignoradas). Un *nombre variable* es una sola letra. Simple no permite nombres descriptivos de variables, por lo que las variables deberán ser explicadas en comentarios, para documentar su uso dentro del programa. Simple utiliza únicamente variables enteras. Simple no tiene declaraciones de variables —el mero mencionar de un nombre de variable en un programa hace que la variable sea declarada y automáticamente inicializada a cero. La sintaxis de Simple no permite

Comando	Enunciado de muestra	Descripción
`rem`	`50 rem this is a remark`	Cualquier texto que siga al comando `rem` es sólo para fines de documentación y será ignorado por el compilador
`input`	`30 input x`	Despliega un signo de interrogación, para solicitar al usuario que introduzca un entero. Lee dicho entero del teclado y almacena el entero en `x`.
`let`	`80 let u = 4 * (j - 56)`	Asigna `u` el valor de `4* (j - 56)`. Note que a la derecha del signo igual puede aparecer una expresión arbitrariamente compleja.
`print`	`10 print w`	Despliega el valor de `w`.
`goto`	`70 goto 45`	Transfiere el control del programa a la línea `45`.
`if/goto`	`35 if i == z goto 80`	Compara `i` con `z` buscando igualdad y transfiere el control del programa a la línea `80`, si la condición es verdadera; de lo contrario, continúa la ejecución en el enunciado siguiente.
`end`	`99 end`	Termina la ejecución del programa.

Fig. 12.23 Comandos Simple.

la manipulación de cadenas (lectura de una cadena, escritura de una cadena, comparación de cadenas, etcétera).

Si en un programa Simple se encuentra una cadena (después de un comando distinto que `rem`), el compilador generará un error de sintaxis. Nuestro compilador supondrá que los programas Simple se escriben en forma correcta. En el ejercicio 12.19 se le solicita al estudiante que modifique el compilador para que lleve a cabo verificación de errores de sintaxis.

Simple utiliza durante la ejecución del programa el enunciado condicional `if/goto` y el enunciado incondicional `goto` para alterar el flujo de control. Si la condición en el enunciado `if/goto` es verdadera, el control se transfiere a una línea específica dentro del programa. Los siguientes operadores de igualdad relacionales son válidos en un enunciado `if/goto`: `<`, `>`, `<=`, `>=`, `==`, o `!=`. La precedencia de estos operadores es la misma que en C.

Consideremos ahora varios programas Simple que demuestran las características de Simple. El primer programa (la figura 12.24) lee dos enteros del teclado, almacena los valores en las variables `a` y `b` y calcula e imprime su suma (almacenada en la variable `c`).

El programa de la figura 12.25 determina e imprime cuál es el mayor de dos enteros. Los enteros son introducidos desde el teclado y almacenados en `s` y en `t`. El enunciado `if/goto` prueba la condición `s=> t`. Si la condición es verdadera, el control se transfiere a la línea `90` y se extrae `s`; de lo contrario, se extrae `t` y el control es transferido al enunciado `end` en la línea `99`, donde el programa termina.

```
10 rem   determine and print the sum of two integers
15 rem
20 rem   input the two integers
30 input a
40 input b
45 rem
50 rem   add integers and store result in c
60 let c = a + b
65 rem
70 rem   print the result
80 print c
90 rem   terminate program execution
99 end
```

Fig. 12.24 Cómo determinar la suma de dos enteros.

```
10 rem   determine the larger of two integers
20 input s
30 input t
32 rem
35 rem   test if s >= t
40 if s >= t goto 90
45 rem
50 rem   t is greater than s, so print t
60 print t
70 goto 99
75 rem
80 rem   s is greater than or equal to t, so print s
90 print s
99 end
```

Fig. 12.25 Cómo encontrar el mayor de dos enteros.

Simple no posee una estructura de repetición (como `for`, `while` o `do/while` de C). Sin embargo, Simple puede simular cada una de las estructuras de repetición de C, mediante el uso de los enunciados `if/goto` y `goto`. La figura 12.26 utiliza un ciclo controlado por centinela para calcular los cuadrados de varios enteros. Cada entero es introducido desde el teclado y almacenado en la variable `j`. Si el valor introducido es el centinela `-9999`, el control se transfiere a la línea `99`, donde el programa termina. De lo contrario, `k` es asignado el valor del cuadrado de `j`, `k` es extraído a la pantalla y el control se pasa a la línea `20`, donde se introduce el siguiente entero.

Utilizando los programas de muestra de la figura 12.24, figura 12.25 y figura 12.26 como guía, escriba un programa Simple para llevar a cabo cada uno de los siguientes:

a) Introducir tres enteros, determinar su promedio e imprimir el resultado.
b) Utilizar un ciclo controlado por centinela para introducir 10 enteros y calcular e imprimir su suma.
c) Utilizar un ciclo controlado por contador para introducir 7 enteros, algunos positivos y algunos negativos, y calcular e imprimir su promedio.
d) Introducir una serie de enteros y determinar e imprimir cuál es el mayor. El primer entero introducido indicará cuántos números deberán de ser procesados.
e) Introducir 10 enteros e imprimir el más pequeño.
f) Calcular e imprimir la suma de enteros pares, desde 2 hasta 30.
g) Calcular e imprimir el producto de los enteros impares desde 1 hasta 9.

12.27 *(Cómo construir un compilador; prerrequisito: haber terminado los ejercicios 7.18, 7.19. 12.12, 12.13 y 12.26).* Ahora que se ha presentado el lenguaje Simple (ejercicio 12.26), analizamos cómo construir nuestro compilador Simple. Primero, consideraremos el proceso mediante el cual un programa Simple se convierte a LMS y es ejecutado por el simulador Simpletron (vea la figura 12.27). Un archivo que contiene un programa Simple es leído por el compilador y convertido a código LMS. El código LMS es extraído a un archivo en disco, en el cual las instrucciones LMS aparecen, una por línea. A contiuación el archivo LMS es cargado en el simulador Simpletron, y los resultados se envían a un archivo en disco y a pantalla. Note que el programa Simpletron desarrollado en el ejercicio 7.19, tomó su entrada desde el teclado. Deberá ser modificado para poder leerse desde un archivo, de tal forma que pueda ejecutar los programas producidos por nuestro compilador.

Para convertirlo a LMS, el compilador ejecuta dos *pasadas* del programa Simple. La primera pasada construye una *tabla simbólica*, en la cual se almacenan todos *los números de línea, nombres de variable* y *constantes* del programa Simple, con su tipo y posición correspondiente en el código final LMS (la tabla simbólica se analiza en detalle más adelante). Esta primera pasada también produce las instrucciones correspondientes LMS para cada enunciado Simple. Como veremos, si el programa Simple contiene

```
10 rem   calculate the squares of several integers
20 input j
23 rem
25 rem   test for sentinel value
30 if j == -9999 goto 99
33 rem
35 rem   calculate square of j and assign result to k
40 let k = j * j
50 print k
53 rem
55 rem   loop to get next j
60 goto 20
99 end
```

Fig. 12.26 Cómo calcular los cuadrados de varios enteros.

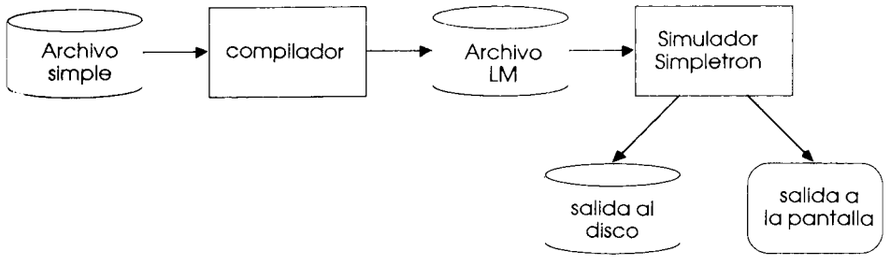


Fig. 12.27 Cómo escribir, compilar y ejecutar un programa de lenguaje Simple.

enunciados en el programa que transfieren el control a una línea más adelante, la primera pasada resulta en un programa LMS que contiene algunas instrucciones incompletas. La segunda pasada del compilador localiza y completa las instrucciones incompletas, y extrae el programa LMS a un archivo.

`Primera pasada`

El compilador empieza leyendo a la memoria un enunciado del programa Simple. La línea deberá ser dividida en sus *componentes léxicos* individuales (es decir, las "piezas" de un enunciado) para proceso y compilación (para facilitar esta tarea se puede utilizar la función estándar de biblioteca **`strtok`**). Recuerde que cada enunciado se inicia con un número de línea seguido por un comando. Conforme el compilador divide un enunciado en *componentes léxicos*, si el componente es un número de línea, una variable o una constante, es colocado en la tabla simbólica. Un número de línea se coloca en la tabla simbólica sólo si se trata del primer componente de un enunciado. El **`symbolTable`** es un arreglo de estructuras **`tableEntry`** que representan cada uno de los símbolos en el programa. No existen restricciones en el número de símbolos que pueden aparecer en el programa. Por lo tanto, el **`symbolTable`** para un programa en particular pudiera resultar estenso. Por ahora, haga **`symbolTable`** un arreglo de 100 elementos. Una vez que el programa esté trabajando podrá aumentar o disminuir su tamaño.

La definición de estructura **`tableEntry`** es como sigue:

```
struct tableEntry {
   int symbol;
   char type;          /* 'C', 'L', or 'V' */
   int location; /* 00 to 99 */
}
```

Cada estructura **`tableEntry`** contiene tres miembros. El miembro **`symbol`** es un entero que contiene la representación ASCII de una variable (recuerde que los nombres de variable son un carácter solo), un número de línea o una constante. El miembro **`type`** es uno de los siguientes caracteres, indicando el tipo del símbolo: **`'C'`** para constante, **`'L'`** para número de línea, o **`'V'`** para variable. El miembro **`location`** contiene la posición en memoria Simpletron (**`00`** a **`99`**) al cual se refiere el símbolo. La memoria Simpletron es un arreglo de 100 enteros, en el cual se almacenan las instrucciones LMS y los datos. Para un número de línea, la posición es el elemento en el arreglo de memoria Simpletron en el cual las instrucciones LMS empiezan para el enunciado Simple. Para una variable o una constante, la posición es el elemento en el arreglo de memoria Simpletron en el cual está almacenada la variable o la constante. Las variables y las constantes están asignadas hacia atrás a partir del final de la memoria Simpletron. La primera variable o constante se almacena en la posición en **`99`**, la siguiente en la posición en **`98`**, etcétera.

La tabla simbólica juega una parte integral en la conversión de programa Simple a LMS. Aprendimos en el capítulo 7 que una instrucción LMS es un entero de cuatro dígitos, formados de dos partes —el *código de operación* y el *operando*. El código de operación está determinado en Simple mediante comandos. Por ejemplo, el comando sencillo **`input`** corresponde al código de operación LMS **`10`** (leer), y el comando Simple **`print`** corresponde al código de operación LMS **`11`** (escribir). El operando es una posición de

memoria, que contiene los datos sobre los cuales el código de operación ejecutará su tarea (por ejemplo, el código de operación **`10`** lee un valor del teclado y lo almacena en la posición de memoria especificado por el operando). El compilador busca en **`symbolTable`** para determinar la posición en la memoria Simpletron de cada símbolo, a fin de que la posición correspondiente pueda ser utilizada para completar las instrucciones LMS.

La compilación de cada enunciado Simple se basa en su comando. Por ejemplo, una vez inserto en la tabla simbólica el número de línea de un enunciado **`rem`**, el resto del enunciado es ignorado por el compilador, dado que el comentario es únicamente para fines de documentación. Los enunciados **`input`**, **`print`**, **`goto`** y **`end`** corresponden a las instrucciones *read, write, branch* (a una posición específica) y *halt* de LMS. Los enunciados que contengan estos comandos Simple se convierten directamente a LMS (note que un enunciado **`goto`** pudiera contener una referencia sin resolver, si la línea especificada se refiere a un enunciado posterior del archivo de programa Simple; esto a veces se conoce como una referencia hacia adelante).

Cuando se compila un enunciado **`goto`** con una referencia sin resolver, la instrucción LMS deberá ser marcada con una *bandera* para indicar que la instrucción será terminada en la segunda pasada del compilador. Las banderas se almacenan en un arreglo de 100 elementos **`flags`** del tipo **`int`**, en el cual cada elemento se inicializa a **`-1`**. Si la posición de memoria a la cual se refiere el número de línea en el programa Simple aún no es conocido (es decir, aún no aparece en la tabla simbólica), el número de línea se almacena en el arreglo **`flags`** en aquel elemento que tenga el mismo subíndice que la instrucción incompleta. El operando de la instrucción incompleta se establece temporalmente a **`00`**. Por ejemplo, una instrucción incondicional de bifurcación (haciendo una referencia hacia adelante) se deja como **`+4000`** hasta la siguiente pasada del compilador. La segunda pasada del compilador será descrita en breve.

La compilación de los enunciados **`if/goto`** y **`let`** es más complicada que la de otros enunciados —son los únicos enunciados que producen más de una instrucción LMS. Para un enunciado **`if/goto`**, el compilador produce código para probar la condición y si es necesario, para bifurcarse a otra línea. El resultado de la bifurcación podría ser una referencia sin resolver. Cada uno de los operadores relacionales y de igualdad pueden ser simulados utilizando las instrucciones *branch zero* y *branch negative* de LMS (o posiblemente una combinación de ambas).

En el caso de un enunciado **`let`**, el compilador produce un código para evaluar una expresión arbitrariamente compleja, compuesta de variables y/o constantes enteras. Las expresiones deberán de separar cada operando y operador con espacios. Los ejercicios 12.12 y 12.13 presentaron el algoritmo de conversión infijo a postfijo y el algoritmo de evaluación postfijo, utilizado por compiladores para evaluar expresiones. Antes de seguir adelante con su compilador, deberá de haber completado cada uno de esos ejercicios. Cuando un compilador se encuentra con una expresión, convierte la expresión de notación infija a notación postfija, y a continuación evalúa la expresión postfija.

¿Cómo es que el compilador produce el lenguaje máquina para valuar una expresión que contenga variables? El algoritmo de evaluación postfijo contiene un "gancho" que permite que nuestro compilador genere instrucciones LMS, en vez de que proceda a evaluar la expresión. Para habilitar este "gancho" dentro del compilador, deberá ser modificado el algoritmo de evaluación postfijo, para que cada símbolo que encuentre (y posiblemente inserte) sea buscado en la tabla simbólica, determine la posición de memoria correspondiente a dicho símbolo, e *inserte (push) la posición de memoria sobre la pila en lugar del símbolo.* Cuando en la expresión postfija se encuentra un operador, son extraídas (pop) las dos posiciones de memoria de la parte superior de la pila, y se produce lenguaje máquina para llevar a cabo la operación, utilizando como operandos las posiciones de memoria. El resultado de cada una de estas subexpresiones se almacena en una posición de memoria temporal y se inserta (push) de vuelta a la pila, de tal forma que se pueda continuar con la evaluación de la expresión postfija. Cuando quede terminada la evaluación postfija, la posición de memoria que contiene el resultado es la única posición remanente sobre la pila. Esta es extraída (popped) y se generan instrucciones LMS para asignar el resultado a la variable en la parte izquierda del enunciado **`let`**.

Segunda pasada

La segunda pasada del compilador lleva a cabo dos tareas; resolver cualquier referencia pendiente de resolver y extraer el código LMS a un archivo. La resolución de las referencias se ejecuta como sigue:

1) Busca en el arreglo `flags` alguna referencia sin resolver (es decir, un elemento que tenga un valor distinto a `-1`).
2) Localiza la estructura en el arreglo `symbolTable`, que contiene el símbolo almacenado en el arreglo `flags` (asegúrese que el tipo del símbolo es `'L'`, correspondiente a número de línea).
3) Inserte la posición de memoria del miembro de estructura `location` dentro de la instrucción con la referencia pendiente de resolver (recuerde que una instrucción que contenga una referencia pendiente de resolver tiene un operando `00`).
4) Repita los pasos 1, 2 y 3, hasta que se llegue al final del arreglo `flags`.

Una vez terminado el proceso de resolución, todo el arreglo que contiene el código LMS es extraído a un archivo de disco, con una instrucción LMS por línea. Este archivo puede ser leído por el Simpletron para su ejecución (después de que el simulador haya sido modificado para que pueda leer su entrada a partir de un archivo).

Un ejemplo completo

El ejemplo siguiente ilustra una conversión completa de un programa Simple a LMS, como sería ejecutado por el compilador Simple. Considere un programa Simple, que introduce un entero y suma los valores desde 1 hasta dicho entero. El programa y las instrucciones LMS producidas por la primera pasada se ilustran en la figura 12.28. La tabla simbólica construida en la primera pasada se muestra en la figura 12.29.

La mayor parte de los enunciados Simple se convierten directamente a instrucciones sencillas LMS. Las excepciones a este programa son los comentarios, el enunciado `if/goto` de la línea `20`, y los enunciados `let`. Los comentarios no se traducen a lenguaje máquina. Sin embargo, el número de línea correspondiente a un comentario es colocado en la tabla simbólica, por si en un enunciado `goto` o en un enunciado `if/goto` el número de línea aparece referenciado. La línea `20` del programa especifica que si la condición `y == x` es verdadera, el control del programa se transfiere a la línea `60`. Dado que la línea `60` aparece más adelante en el programa, en la primera pasada del compilador no ha colocado aún `60` en la tabla simbólica (los números de línea se colocan en la tabla simbólica únicamente cuando aparecen como primera división léxica de un enunciado). Por lo tanto, en este momento no es posible determinar el operando de la instrucción *branch zero* LMS en la posición `03` en el arreglo de instrucciones LMS. El compilador colocará `60` en la posición `03` del arreglo `flags`. para indicar que esta instrucción se completará en la segunda pasada.

En el arreglo LMS debemos mantener registro de la posición de la siguiente instrucción, porque no existe una correspondencia uno a uno, entre los enunciados Simple y las instrucciones LMS. Por ejemplo, el enunciado `if/goto` de la línea `20` se compila en tres instrucciones LMS. Cada vez que se produce una instrucción, dentro del arreglo LMS debemos incrementar el *contador de instrucciones* a la siguiente posición. Note que el tamaño de la memoria Simpletron podría presentar un problema para los programas Simple que contengan muchos enunciados, variables y constantes. Es concebible que el compilador se quede sin memoria disponible. Para comprobar si éste es el caso, su programa deberá contener un *contador de datos,* a fin de llevar registro de la posición dentro del arreglo LMS en la cual se almacenará la siguiente variable o constante. Si el valor del contador de instrucciones es mayor que el valor que el del contador de datos, el arreglo LMS está lleno. En este caso, deberá darse por terminado el proceso de compilación y el compilador deberá imprimir un mensaje de error, indicando que durante la compilación se le terminó la memoria.

Revisión paso a paso del proceso de compilación

Recorramos el proceso de compilación del programa Simple de la figura 12.28. El compilador lee a la memoria la primera línea del programa

```
5 rem sum 1 to x
```


Programa simple	Localización e instrucción LMS		Descripción
`5 rem sum 1 to x`		*ninguna*	`rem` es ignorado
`10 input x`	`00`	`+1099`	leer `x` a la posición `99`
`15 rem check y == x`		*ninguna*	`rem` es ignorado
`20 if y == x goto 60`	`01`	`+2098`	cargar `y` (`98`) a acumulador
	`02`	`+3199`	substraer `x` (`99`) del acumulador
	`03`	`+4200`	*bifurcar cero a posición sin resolver*
`25 rem increment y`		*ninguna*	`rem` es ignorado
`30 let y = y + 1`	`04`	`+2098`	cargar `y` al acumulador
	`05`	`+3097`	añadir `1` (`97`) al acumulador
	`06`	`+2196`	almacenar en posición temporal `96`
	`07`	`+2096`	cargar de la posición temporal `96`
	`08`	`+2198`	almacenar acumulador en `y`
`35 rem add y to total`		*ninguna*	`rem` es ignorado
`40 let t = t + y`	`09`	`+2095`	cargar `t` (`95`) al acumulador
	`10`	`+3098`	añadir `y` al acumulador
	`11`	`+2194`	almacenar en posición temporal `94`.
	`12`	`+2094`	cargar de la posición temporal `94`
	`13`	`+2195`	almacenar acumulador en `t`
`45 rem loop y`		*ninguna*	`rem` es ignorado
`50 goto 20`	`14`	`+4001`	bifurcarse a la posición `01`
`55 rem output result`		*ninguna*	`rem` es ignorado
`60 print t`	`15`	`+1195`	extraer `t` a pantalla
`99 end`	`16`	`+4300`	terminar la ejecución.

Fig. 12.28 Instrucciones LMS, producidas después de la primera pasada del compilador.

La primera división léxica del enunciado (el número de línea) se determina utilizando `strtok` (vea el capítulo 8 en el que se analizan las funciones de manipulación de las cadenas en C). El componente léxico regresado por `strtok` se convierte a un entero, mediante el uso de `atoi`, de tal forma que el símbolo `5` puede ser localizado en la tabla simbólica. Si no se encuentra el símbolo, se insertará en la tabla simbólica. Dado que estamos al principio del programa y ésta es la primera línea, en la tabla aún no existen símbolos. Por lo tanto, en la tabla simbólica se inserta `5` como del tipo `L` (número de línea) y se le asigna la primera posición en el arreglo LMS (`00`): aunque esta línea se trata de un comentario, aún así en la tabla simbólica se le asigna un espacio para el número de línea (por si es referenciado por un `goto` o por un `if/goto`). Como en relación con un enunciado `rem` no se genera ninguna instrucción LMS, no se incrementa el contador de instrucciones.

El enunciado

```
10 input x
```

es analizado léxicamente después. El número de línea `10` se coloca en la tabla simbólica como del tipo `L` y se le asigna la primera posición en el arreglo LMS (`00` debido a que un comentario fue el que inició el

Símbolo	Tipo	Posición
5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04
30	L	04
1	C	97
35	L	09
40	L	09
't'	V	95
45	L	14
50	L	14
55	L	15
60	L	15
99	L	16

Fig. 12.29 Tabla simbólica correspondiente al programa 12.28.

programa, por lo que el contador de instrucciones es todavía `00`). El comando `input` indica que la siguiente división léxica es una variable (únicamente una variable puede aparecer en un enunciado `input`). Dado que `input` corresponde directamente a un código de operación LMS, el compilador simplemente tiene que determinar la posición de `x` en el arreglo LMS. El símbolo `x` no se encuentra en la tabla simbólica.

Por lo tanto, en la tabla simbólica se le inserta como la representación ASCII de `x`, se le da el tipo `V`, y en el arreglo LMS se le asigna a la posición `99` (el almacenamiento de datos se inicia en `99` y se asigna hacia atrás). El código LMS puede ser generado a partir de este enunciado. El código de operación `10` (el código de operación de lectura LMS) se multiplica por 100, y se le añade la posición de `x` (como quedó determinada en la tabla simbólica), para completar la instrucción. La instrucción es entonces almacenada en el arreglo LMS en la posición `00`. El contador de instrucciones se aumenta en 1, porque se produjo una sola instrucción LMS.

El enunciado

```
15 rem check y == x
```

se divide léxicamente después. El número de línea `15` es buscado en la tabla simbólica (y no se encuentra). El número de línea se inserta como del tipo `L` y se le asigna la siguiente posición en el arreglo, `01` (recuerde que los enunciados `rem` no producen código, por lo que no se aumenta el contador de instrucciones).

El enunciado

```
20 if y == x goto 60
```

se divide a continuación. El número de línea `20` se inserta en la tabla simbólica, se le da el tipo `L` con la siguiente posición `01` en el arreglo LMS. El comando `if` indica que se tendrá que hacer una evaluación de condición. La variable `y` no se encuentra en la tabla simbólica, por lo que se inserta y dándosele el tipo `V` y la posición LMS `98`. A continuación, se generan instrucciones LMS para evaluar la condición. Dado que no existe un equivalente directo en LMS para `if/goto`, debe de ser simulado, ejecutando un cálculo utilizando `x` y `y` y bifurcación, basándose en el resultado. Si `y` es igual a `x`, el resultado de restar `x` de `y` debe ser cero, por lo que para simular el enunciado `if/goto` la instrucción *branch zero* puede ser utilizada

con el resultado del cálculo. El primer paso requiere que se cargue `y` (de la posición `98` LMS) al acumulador. Esto produce la instrucción `01 +2098`. A continuación se resta `x` del acumulador. Esto produce la instrucción `02 +3199`. El valor que aparece en el acumulador puede ser cero, positivo o negativo. Dado que el operador es `==`, deseamos hacer una *bifurcación cero*. Primero, se busca la tabla simbólica por la posición de bifurcación (en este caso `60`), misma que no se encuentra. Por lo tanto, `60` se coloca en el arreglo `flags` en la posición `03`, y se genera la instrucción `03 +4200` (no podemos añadir la posición de bifurcación, porque todavía no hemos asignado en el arreglo LMS una posición a la línea `60`). El contador de instrucciones se incrementa a `04`.

El compilador sigue con el enunciado

```
25 rem increment y
```

El número de línea 25 se inserta en la tabla simbólica como de tipo `L` y se le asigna la posición LMS `04`. El contador de instrucciones no se incrementa.

Cuando se divide léxicamente el enunciado

```
30 let y = y + 1
```

el número de línea `30` se inserta en la tabla simbólica como del tipo `L` y se le asigna la posición LMS `04`. El comando `let` indica que la línea es un enunciado de asignación. Primero, todos los símbolos en la línea se insertan en la tabla simbólica (si todavía no están ahí). El entero `1` se añade a la tabla simbólica como del tipo `C` y se le asigna la posición LMS `97`. A continuación, el lado derecho de la asignación se convierte de notación infija a postfija. A continuación se evalúa la expresión postfija (`y 1 +`). El símbolo `y` se localiza en la tabla simbólica y se inserta en la pila su posición correspondiente en memoria. El símbolo `1` también es localizado en la tabla simbólica y se inserta dentro de la pila su posición correspondiente en memoria. Cuando se llega al operador `+`, el evaluador de postfijo extrae (pop) la pila hacia el operando derecho del operador y vuelve a extraer la pila nuevamente en el operando izquierdo del operador, y a continuación produce las instrucciones LMS

```
04 +2089 (cargar y)
05 +3097 (añadir 1)
```

El resultado de la expresión se almacena en una posición temporal en memoria (`96`) mediante la instrucción

```
06 +2196 (almacenar temporal)
```

y la posición temporal se inserta en la pila. Ahora que ha sido evaluada la expresión, el resultado debe ser almacenado en `y` (es decir la variable del lado izquierdo del signo igual). Por lo tanto, la posición temporal se carga al acumulador y el acumulador se almacena en `y`, mediante las instrucciones

```
07 +2096 (cargar temporal)
08 +2198 (almacenar y)
```

El lector notará de inmediato que las instrucciones LMS son aparentemente redundantes. Analizaremos este tema en breve.

Cuando se divide léxicamente el enunciado

```
35 rem    add y to total
```

el número de línea 35 se inserta en la tabla simbólica como del tipo `L` y se le asigna la posición `09`.

El enunciado

```
40 let t = t + y
```

es similar a la línea 30. La variable se inserta en la tabla simbólica como en el tipo `V` y se le asigna la posición LMS `95`. Las instrucciones siguen la misma lógica y formato que en la línea `30`, y son generadas las instrucciones `09 + 2095, 10 +3098, 11 +2194, 12 +2094,` y `13 +2195`. Note que el resultado

de `t + y` se asigna a la posición temporal `94` antes de asignarse a `t (95)`. Otra vez, el lector notará que las instrucciones en las posiciones de memoria `11` y `12` aparecen como redundantes. Otra vez, en breve analizaremos lo anterior.

El enunciado

```
45 rem    loop y
```

es un comentario, por lo que la línea `45` se añade a la tabla simbólica como del tipo `L` y se le asigna la posición LMS `14`.

El enunciado

```
50 goto 20
```

transfiere el control a la línea 20. El número de línea `50` se inserta en la tabla simbólica como del tipo `L` y se le asigna a la posición LMS `14`. En LMS el equivalente de `goto` es la instrucción *unconditional branch* (`40`) que transfiere control a una posición LMS específica. El compilador busca la línea 20 en la tabla simbólica y encuentra que corresponde a la posición LMS 01. El código de operación (`40`) se multiplica por 100 y la posición `01` se añade a esta cifra, para producir la instrucción `14 +4001`.

El enunciado

```
55 rem    output result
```

es un comentario, por lo que la línea `55` se inserta a la tabla simbólica como del tipo `L` y se le asigna la posición LMS `15`.

El enunciado

```
60 print t
```

es un enunciado de salida. El número de línea `60` se inserta en la tabla simbólica como del tipo `L` y se le asigna la posición LMS `15`. El equivalente de `print` en código de operación LMS es `11` (*write*). La posición de `t` se determina a partir de la tabla simbólica y se añade al resultado de multiplicar el código de operación por 100.

El enunciado

```
99 end
```

es la línea final del programa. El número de línea `99` se almacena en la tabla simbólica como del tipo `L` y se le asigna la posición LMS `16`. El comando `end` produce la instrucción LMS `+4300` (en LMS `43` es *halt*) que se escribe como instrucción final en el arreglo de memoria LMS.

Esto completa la primera pasada del compilador. Ahora veamos la segunda pasada. Valores distintos a `-1` son buscados dentro del arreglo `flags`. La posición `03` contiene `60`, por lo que el compilador sabe que la instrucción `03` está incompleta. El compilador completa la instrucción buscando en la tabla simbólica el `60`, determinando su posición y añadiendo la posición a la instrucción incompleta. En este caso, la búsqueda determina que la línea `60` corresponde a la posición LMS `15`, por lo que en remplazo de `03` `+4200` se produce la instrucción completa `03 +4215`. Ahora el programa Simple ha quedado ya compilado con éxito.

Para construir el compilador tendrá que llevar a cabo cada una de las siguientes tareas.

a) Modificar el programa simulador Simpletron, que usted escribió en el ejercicio 7.19, para que tome su entrada a partir de un archivo especificado por el usuario (vea el capítulo 11). También, el simulador deberá extraer sus resultados a un archivo de disco, con el mismo formato que la salida a pantalla.

b) Modifique el algoritmo de evaluación infijo a postfijo del ejercicio 12.12, para que pueda procesar operandos enteros multidigitales, y operandos de nombres de variables de una sola letra. *Sugerencia:* puede ser utilizada la función estándar de biblioteca `strtok` para localizar dentro de una expresión cada constante y variable, y las constantes pueden ser convertidas de

cadenas a enteros utilizando la función estándar de biblioteca `atoi`. (Nota: deberá ser modificada la representación de datos de la expresión postfija, para que acepte nombres de variables y constantes enteras).

c) Modifique el algoritmo de evaluación postfijo para que pueda procesar operandos enteros de más de un dígito y operandos de nombre de variable. También, el algoritmo deberá ahora poner en funcionamiento el "gancho" discutido anteriormente, de tal forma que sean producidas las instrucciones LMS en vez de que la expresión se evalúe directamente. Sugerencia: la función estándar de biblioteca `strtok` puede ser utilizada para localizar dentro de una expresión cada una de las constantes y variables, y las constantes pueden ser convertidas de cadenas a enteros utilizando la función estándar de biblioteca `atoi`. (Nota: deberá ser modificada la representación de datos de una expresión postfija, para que acepte nombres de variables y constantes enteras.)

d) Construya el compilador. Incorpore las partes (b) y (c) para evaluar las expresiones en los enunciados `let`. Su programa deberá contener una función que ejecute la primera pasada del compilador, y una función que ejecute la segunda pasada del compilador. Para llevar a cabo sus tareas ambas funciones pueden llamar a otras funciones.

12.28 *(Cómo optimizar el compilador Simple).* Cuando se compila un programa y se convierte a LMS, se generan un conjunto de instrucciones. A menudo ciertas combinaciones de instrucciones se repiten, normalmente en grupos de tres, llamadas *producciones*. Una producción está formada normalmente de tres instrucciones, como *load, add,* y *store*. Por ejemplo, en la figura 12.30 se ilustran cinco de las instrucciones LMS, que fueron producidas en la compilación del programa de la figura 12.28. Las primeras tres instrucciones son la producción que añade `1` a `y`. Note que las instrucciones `06` y `07` almacenan el valor en el acumulador en la posición temporal `96`. Y a continuación vuelven a cargar el valor en el acumulador, de tal forma que la instrucción `08` puede almacenar el valor en la posición `98`. A menudo una producción es seguida por una instrucción de carga para la misma posición que fue almacenada inmediatamente antes. Este código puede ser *optimizado* eliminando la instrucción de almacenar y las instrucciones de cargar subsecuentes, que operan en la misma posición de memoria. Esta optimización permitiría que el programa Simpletron se ejecutara más aprisa, porque en esta versión existirían menos instrucciones. En la figura 12.31 se ilustra el LMS optimizado, correspondiente al programa de la figura 12.28. Note que en el código optimizado hay cuatro instrucciones menos —un ahorro de espacio en memoria del 25%.

Modifique el compilador para que pueda dar la opción de optimizar el código que produce en lenguaje de máquina Simpletron. Compare manualmente el código no optimizado con el código optimizado, y calcule el porcentaje de reducción.

12.29 *(Modificaciones al compilador Simple).* Lleve a cabo las siguientes modificaciones al compilador Simple. Algunas de estas modificaciones también pudieran requerir modificaciones al programa simulador Simpletron escrito en el ejercicio 7.19.

a) Haga posible que en los enunciados `let` se pueda utilizar el operador de módulo (%). El lenguaje de máquina Simpletron deberá ser modificado para incluir la instrucción de módulo.

b) Haga posible la exponenciación en un enunciado `let` utilizando el `^` como operador de exponenciación. El lenguaje de máquina Simpletron debe ser modificado para incluir la instrucción de exponenciación.

```
04 +2098    (load)
05 +3097    (add)
06 +2196    (store)
07 +2096    (load)
08 +2198    (store)
```

Fig. 12.30 Código sin optimizar del programa de la figura 12.28.

Programa simple	Localización e instrucción LMS		Descripción
`5 rem sum 1 to x`		*ninguna*	`rem` es ignorado
`10 input x`	`00`	`+1099`	leer `x` a la posición `99`
`15 rem check y == x`		*ninguna*	`rem` es ignorado
`20 if y == x goto 60`	`01`	`+2098`	cargar `y` (`98`) a acumulador
	`02`	`+3199`	substraer `x` (`99`) del acumulador
	`03`	`+4211`	bifurcar cero a posición sin resolver
`25 rem increment y`		*ninguna*	`rem` es ignorado
`30 let y = y + 1`	`04`	`+2098`	cargar `y` al acumulador
	`05`	`+3097`	añadir `1` (`97`) al acumulador
	`06`	`+2198`	almacenar acumulador en `y` (`98`)
`35 rem add y to total`		*ninguna*	`rem` es ignorado
`40 let t = t + y`	`07`	`+2096`	cargar `t` de posición (`96`)
	`08`	`+3098`	añadir `y` (`98`) a acumulador
	`09`	`+2196`	almacenar acumulador en `t` (`96`)
`45 rem loop y`		*ninguna*	`rem` es ignordo
`50 goto 20`	`10`	`+4001`	bifurcarse a la posición `01`
`55 rem output result`		*ninguna*	`rem` es ignorado
`60 print t`	`11`	`+1195`	extraer `t` (`96`) a pantalla
`99 end`	`12`	`+4300`	terminar la ejecución.

Fig. 12.31 Código optimizado para el programa de la figura 12.28.

c) Haga posible que en los enunciados Simple el compilador reconozca letras mayúsculas y minúsculas (por ejemplo `'A'` sea equivalente a `'a'`). No son requeridas modificaciones al simulador Simpletron.

d) Haga posible que los enunciados `input` puedan leer valores para variables múltiples, como `input x, y`. No se requieren modificaciones al simulador Simpletron.

e) Haga posible que el compilador extraiga varios valores en un enunciado único `print`, como `print a, b, c`. No se requieren modificaciones al simulador Simpletron.

f) Añada capacidades de verificación de sintaxis al compilador, de tal forma que cuando en un programa Simple se encuentren errores de sintaxis se emitan mensajes de error. No se requieren modificaciones al simulador Simpletron.

g) Haga posible el uso de arreglos de enteros. No se requieren modificaciones al simulador Simpletron.

h) Permitir subrutinas especificadas por los comandos Simple `gosub` y `return`. El comando `gosub` pasa el control del programa a una subrutina y el comando `return` pasa el control de regreso al enunciado existente después del `gosub`. Esto es similar a una llamada de función en C. La misma subrutina puede ser llamada a partir de muchos `gosub` distribuidos a lo largo de un programa. No se requiere de modificaciones al simulador Simpletron.

i) Haga posible estructuras de repetición de la forma


```
for x = 2 to 10 step 2
    enunciados Simple
next
```

Este enunciado `for` cicla desde `2` hasta `10` con un incremento de `2`. La línea `next` marca el fin del cuerpo de la línea `for`. No se requieren de modificaciones al simulador Simpletron.

j) Haga posible estructuras de repetición de la forma

```
for x = 2 to 10
    enunciados Simple
next
```

Este enunciado `for` cicla desde `2` hasta `10` con un incremento preestablecido de `1`. No se requieren de modificaciones al simulador Simpletron.

k) Haga posible que el compilador procese entradas y salidas de cadenas. Esto requiere la modificación del simulador Simpletron para procesar y almacenar valores de cadenas. *Sugerencia:* cada palabra Simpletron puede ser dividida en dos grupos, cada uno de los grupos conteniendo un entero de dos dígitos. Cada entero de dos dígitos representa el equivalente decimal ASCII de un carácter. Añada una instrucción en lenguaje máquina que imprimirá una cadena empezando en una posición de memoria Simpletron. La primera mitad de la palabra en dicha posición es una cuenta del número de caracteres dentro de la cadena (es decir la longitud de la cadena). Cada media palabra sucesiva contiene un carácter ASCII expresado como dos dígitos decimales. La instrucción en lenguaje máquina verifica la longitud e imprime la cadena traduciendo cada número de dos dígitos en su carácter equivalente.

l) Haga posible que además de enteros el compilador procese valores de punto flotante. También deberá ser modificado el simulador Simpletron para procesar valores de punto flotante.

12.30 *(Un intérprete Simple).* Un intérprete es un programa que lee un enunciado de un programa en un lenguaje de alto nivel, determina la operación a ejecutarse por dicho enunciado, y de inmediato procede a ejecutar la operación. El programa no se convierte primero a lenguaje máquina. Los intérpretes ejecutan lentamente, porque cada enunciado que se encuentra en el programa deberá primero ser descifrado. Si los enunciados están incluidos en un ciclo, cada vez que se encuentren dentro del ciclo, los enunciados serán descifrados. Las versiones primeras del lenguaje de programación BASIC eran puestas en operación por medio de intérpretes.

Escriba un intérprete para el lenguaje Simple analizado en el ejercicio 12.26. El programa deberá utilizar el convertidor de infijo a postfijo desarrollado en el ejercicio 12.12 y el evaluador postfijo desarrollado en el ejercicio 12.13 para evaluar las expresiones en un enunciado `let`. En este programa deberán ser impuestas las mismas restricciones puestas en práctica en el lenguaje Simple del ejercicio 12.26. Pruebe el intérprete con los programas Simple escritos en el ejercicio 12.26. Compare los resultados de ejecutar estos programas en el intérprete con los resultados de compilar programas Simple y ejecutarlos en el simulador Simpletron que se construyó en el ejercicio 7.19.

13

El preprocesador

Objetivos

- Ser capaz de utilizar **#include** para el desarrollo de programas extensos.
- Ser capaz de utilizar **#define** para crear macros y macros con argumentos.
- Comprender la compilación condicional.
- Ser capaz de mostrar mensajes de error durante la compilación condicional.
- Ser capaz de usar afirmaciones para probar si los valores de expresiones son correctos.

Sostén lo bueno; defínelo bien.
Alfred, Lord Tennyson

Le he encontrado un argumento; pero no estoy obligado de encontrarle una comprensión.
Samuel Johnson

Un buen símbolo es el mèjor argumento, y es un misionero para persuadir a miles.
Ralph Waldo Emerson

Las condiciones son básicamente sanas.
Herbert Hoover (Diciembre 1929)

Al partidario, cuando está inmerso en una discusión, no le importan los derechos de la pregunta, sino que solo está ansioso de convencer a sus oyentes de sus propias afirmaciones.
Platón

Sinopsis

Resumen • Terminología • Errores comunes de programación • Prácticas sanas de programación • Sugerencias de rendimiento • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios.

13.1 Introducción

Este capítulo presenta el *preprocesador C*. El preprocesamiento ocurre antes de que se compile un programa. Algunas acciones posibles son: inclusión de otros archivos en el archivo bajo compilación, definición de *constantes simbólicas* y de *macros*, *compilación condicional* de código de programa, y *ejecución condicional de directivas* de preprocesador. Todas las directivas de preprocesador empiezan con el signo **#**, y antes de una directiva de preprocesador en una línea sólo pueden aparecer espacios en blanco.

13.2 La directiva de preprocesador # include

La *directiva de preprocesador* `#include` ha sido utilizada a todo lo largo de este texto. La directiva **#include** hace que en el lugar de la directiva se incluya una copia del archivo especificado. Las dos formas de la directiva **#include** son:

```
#include <filename>
#include "filename"
```

La diferencia entre estas formas estriba en la localización donde el preprocesador buscará el archivo a incluirse. Si el nombre del archivo está encerrado entre comillas, el preprocesador buscará el archivo a incluirse en el mismo directorio que el del archivo que se está compilando. Este método se utiliza normalmente para incluir archivos de cabecera definidos por el programador. Si el nombre del archivo esta encerrado en corchetes angulares (< y >) —que se utilizan para los *archivos de cabecera estándar de biblioteca*— la búsqueda se llevará a cabo de forma independiente a la puesta en práctica, por lo regular a través de directorios predesignados.

La directiva **#include** se utiliza por lo general para incluir archivos de cabecera de biblioteca estándar, como **stdio.h** y **stdlib.h** (vea la figura 5.6). La directriz **#include** también se utiliza con programas formados de varios archivos fuente, que deben ser compilados juntos. A menudo es creado e incluido en el archivo un *archivo de cabecera* conteniendo declaraciones comunes a los varios archivos de un programa. Ejemplos de declaraciones como éstas son declaraciones de estructuras y de uniones, en numeraciones y prototipos de función.

En UNIX, los archivos de programa se compilan utilizando el *comando* CC. Por ejemplo, para compilar y enlazar **main.c** con **square.c** escriba el comando.

```
cc main.c square.c
```

en el indicador de UNIX. Esto produce el archivo ejecutable **a.out**. Para mayor información sobre la compilación, el enlace y la ejecución de programas, vea los manuales de referencia correspondientes a su compilador.

13.3 La directiva de preprocesador # define: constantes simbólicas

La *directiva* **#define** crea *constantes simbólicas* —constantes representadas como símbolos— y *macros*— operaciones definidas como símbolos. El formato de la directiva **#define** es

```
#define identifier replacement-text
```

Cuando en un archivo aparece esta línea, todas las subsecuentes apariciones de *identifier* serán de forma automática remplazadas por *replacement-text*, antes de la compilación del programa. Por ejemplo,

```
#define PI 3.14159
```

remplaza todas las ocurrencias subsiguientes de la constante simbólica **PI** por la constante numérica **3.14159**. Las constantes simbólicas permiten al programador ponerle un nombre a una constante y utilizarlo a todo lo largo del programa. Si durante el programa la constante requiere de modificación, puede ser modificada una vez en la directiva **#define** y cuando sea el programa recompilado, todas las ocurrencias de la constante dentro del programa de manera automática serán modificadas. Nota: *todo lo que exista a la derecha del nombre de la constante simbólica, remplaza a la constante simbólica.* Por ejemplo, **#define PI = 3.14159** hace que el preprocesador remplace todas las ocurrencias de **PI** con **= 3.14159**. Esto es causa de muchos errores sutiles de lógica y de sintaxis. Es también un error redefinir una constante simbólica con un nuevo valor.

Práctica sana de programación 13.1

Utilizar nombres significativos para las constantes simbólicas, ayuda a que los programas resulten más autodocumentados.

13.4 La directiva de preprocesador #define: macros

Una *macro* es una operación definida en una directiva de preprocesador **#define**. Como en el caso de las constantes simbólicas, antes de compilarse el programa, el *macro identifier* se remplaza en el mismo por el *replacement-text*. Las macros pueden ser definidas con o sin *argumentos*. Una macro sin argumentos se procesa como si fuera una constante simbólica. En una macro con argumentos, los argumentos se sustituyen en el texto de remplazo, y a continuación la macro se *expande*, es decir, en el programa el texto de remplazo remplaza al identificador y a la lista de argumentos.

Veamos la siguiente definición de macro con un argumento, correspondiente al área de un círculo:

```
#define CIRCLE_AREA(x) PI * (x) * (x)
```

Siempre que en el archivo aparezca **CIRCLE_AREA(x)**, el valor de **x** será sustituido por **x** en el texto de remplazo, la constante simbólica **PI** será remplazada por su valor (previamente definido), y en el programa la macro se expandirá. Por ejemplo, el enunciado

```
area = CIRCLE_AREA(4);
```

se expande a

```
area = 3.14159 * (4) * (4);
```

Dado que la expresión está sólo formada de constantes, al momento de compilarse se evalúa el valor de la expresión y se asigna a la variable **area**. Cuando el argumento del macro es una expresión, los paréntesis alrededor de cada **x** en el texto de remplazo obligan al orden apropiado de evaluación. Por ejemplo, el enunciado

```
area = CIRCLE_AREA(c + 2);
```

se expande a

```
area = 3.14159 * (c + 2) * (c + 2);
```

que se evalúa de forma correcta, porque los paréntesis obligan al orden correcto de evaluación. Si se omiten los paréntesis, la expansión del macro es

```
area = 3.14159 * c + 2 * c + 2;
```

lo que se evalúa de forma incorrecta como

```
area = (3.14159 * c) + (2 * c) + 2;
```

en razón de las reglas de precedencia de operadores.

Error común de programación 13.1

Olvidar encerrar los argumentos de macro en paréntesis en el texto de remplazo.

La macro **CIRCLE_AREA** podría ser definida como una función. La función **circleArea**

```
double circleArea(double x)
{
   return 3.14159 * x * x;
}
```

lleva a cabo el mismo cálculo que la macro **CIRCLE_AREA**, pero con la función **circleArea** se asocia la sobrecarga de una llamada de función. Las ventajas de la macro **CIRCLE_AREA** es que las macros insertan código en el programa directamente evitando sobrecarga de función y conservándose el programa legible, porque el cálculo **CIRCLE_AREA** es definido por separado y nombrado de forma significativa. Una desventaja estriba en que su argumento debe evaluarse dos veces.

Sugerencia de rendimiento 13.1

A veces las macros pueden ser utilizadas para sustituir una llamada de función por código en línea, previo al tiempo de ejecución. Esto elimina la sobrecarga de una llamada de función.

La siguiente es una definición de macro con dos argumentos, correspondiente al área de un rectángulo:

```
#define RECTANGLE_AREA(x, y) (x) * (y)
```

Siempre que en el programa aparezca **RECTANGLE_AREA(x, y)**, los valores de **x** y **y** serán sustituidos por el texto de remplazo de la macro, y la macro será expandida en lugar del nombre de la macro. Por ejemplo, el enunciado

```
rectArea = RECTANGLE_AREA(a + 4, b + 7);
```

se expande a

```
rectArea = (a + 4) * (b + 7);
```

El valor de la expresión es evaluado y asignado a la variable **rectArea.**

El texto de remplazo para una macro o una constante simbólica es por lo regular cualquier texto sobre la línea, después del identificador en la directriz **#define**. Si el texto de remplazo de una macro o de una constante simbólica es más largo que el resto de la línea, deberá colocarse una diagonal invertida (\) al final de la misma, indicando que el texto de remplazo continúa sobre la siguiente línea.

Las constantes simbólicas y las macros pueden ser descartadas utilizando la *directiva de preprocesador* **#undef**. La directiva **#undef** "elimina" la definición de una constante simbólica o de un nombre de macro. El *alcance* de una constante simbólica o de una macro cubre desde su definición hasta que mediante **#undef** queda eliminada su definición, o si no, hasta el final del archivo. Una vez eliminada la definición, puede volverse a definir un nombre utilizando **#define**.

Las funciones en la biblioteca estándar son definidas algunas veces como macros basadas en otras funciones de biblioteca. Una macro, comúnmente definida en el archivo de cabecera **stdio.h**, es

```
#define getchar() getc(stdin)
```

La definición de macro de **getchar** utiliza la función **getc** para obtener un carácter a partir del flujo de entrada estándar. A menudo la función **putchar** en el encabezado **stdio.h**, y las funciones de manejo de caracteres del encabezado **ctype.h**, también son puestas en operación como macros. Note que las expresiones con efectos colaterales (es decir, donde los valores de las variables se modifican) no deberían ser pasadas a una macro porque los argumentos de macro pudieran ser evaluados más de una vez.

13.5 Compilación condicional

La *compilación condicional* le permite al programador controlar la ejecución de las directivas de preprocesador, y la compilación del código del programa. Cada una de las directivas de preprocesador evalúa una expresión entera constante. Las expresiones de cambio de tipo, las expresiones **sizeof**, y las constantes de enumeración no pueden ser evaluadas en directivas de preprocesador.

El constructor de preprocesador condicional es muy parecido a la estructura de selección **if**. Veamos el siguiente código de preprocesador:

```
#if !defined(NULL)
   #define NULL 0
#endif
```

Estas directivas determinan si NULL ha sido definido. La expresión `defined(NULL)` se evalúa a `1`, si NULL está definido; y a `0` de lo contrario. Si el resultado es `0`, `!defined(NULL)` se evalúa a `1`, y NULL queda definido. De lo contrario, la directiva `#define` es pasada por alto. Cada constructor `#if` termina con `#endif`. Las directivas `#ifdef` y `#ifndef` son abreviaturas correspondientes a `#ifdefined(`*nombre*`)` y `#if !defined(`*nombre*`)`. Un constructor de preprocesador condicional de varias partes puede ser probado utilizando las directivas `#elif` (el equivalente de `else if` en una estructura `if`) y `#else` (el equivalente de `else` en una estructura `if`).

Durante el desarrollo del programa, a menudo los programadores encuentran útil "anotar" grandes porciones de código para evitar que sean compilados. Si el código contiene comentarios, para esta tarea no es posible utilizar `/*` y `*/`. En vez de ello, el programador puede recurrir al siguiente constructor de preprocesador

```
#if 0
   code prevented from compiling
#endif
```

Para permitir que el código sea compilado, en el constructor anterior el `0` es remplazado por `1`.

La compilación condicional se utiliza por lo común como ayuda de depuración. Muchas instalaciones en C proporcionan *depuradores*. Sin embargo, con frecuencia los depuradores son difíciles de utilizar y de comprender, por lo que son rara vez utilizados por los estudiantes de un primer curso de programación. En vez de ello, se usan enunciados `printf` para imprimir valores de variables y para confirmar el flujo de control. Estos enunciados `printf` pueden ser encerrados en directivas de preprocesador condicionales, de tal forma que los enunciados sólo sean compilados mientras no se haya terminado el proceso de depuración. Por ejemplo

```
#ifdel DEBUG
   printf("Variable x = %d\n", x);
#endif
```

hace que se compile en el programa un enunciado `printf` si la constante simbólica DEBUG ha sido definida (`#define DEBUG`) antes de la directiva `#ifdef DEBUG`. Cuando se haya terminado la depuración, se elimina del archivo fuente la directiva `#define`, y durante la compilación los enunciados `printf`, insertos para fines de depuración, serán ignorados. En programas más extensos pudiera ser deseable definir varias constantes simbólicas distintas, que controlen la compilación condicional en secciones separadas del archivo fuente.

Error común de programación 13.2

Insertar enunciados `printf` compilados condicionalmente para fines de depuración, en posiciones donde en ese momento C espera un enunciado. En este caso, el enunciado compilado condicional debería haberse encerrado en un enunciado compuesto. Entonces, cuando el programa se compile con enunciados de depuración, el flujo del control del programa no será alterado.

13.6 Las directivas de preprocesador #error y #pragma

La *directiva* `#error`

```
#error tokens
```

imprime un mensaje, que depende de la puesta en práctica, incluyendo las *divisiones o componentes léxicas* especificadas en la directriz. Las divisiones léxicas son secuencias de caracteres separadas por espacios. Por ejemplo,


```
#error 1 - Out of range error
```

contiene 6 divisiones léxicas. En Borland C++ para PC, por ejemplo, cuando se procesa la directiva `#error`, se muestran las componentes o divisiones léxicas en la directiva como un mensaje de error, el preproceso se detiene, y el programa no se compila.

La *directiva* `#pragma`

```
#pragma tokens
```

genera una acción definida por la puesta en práctica. Un pragma no reconocido por la puesta en práctica será ignorado. Borland C++, por ejemplo, reconoce varios pragmas que le permiten al programador aprovechar completamente la puesta en marcha de Borland C++. Para mayor información sobre `#error` y `pragma`, vea la documentación en su instalación de C.

13.7 Los operadores # y

Los operadores de preprocesador # y ## están disponibles sólo en ANSI C. El operador # hace que una división léxica de texto de remplazo se convierta en una cadena encerrada entre comillas. Considere la siguiente definición de macro:

```
#define HELLO(x) printf("Hello, " #x "\n");
```

Cuando `HELLO(John)` aparece en un archivo de programa, se expande a

```
printf("Hello, " "John" "\n");
```

La cadena `"John"` remplaza `#x` en el texto de remplazo. Las cadenas separadas por un espacio en blanco son concatenadas durante el preprocesamiento, por lo que el enunciado arriba citado es equivalente a

```
printf("Hello, John\n");
```

Note que el operador # deberá ser utilizado en una macro con argumentos, porque el operando de # se refiere a un argumento de la macro.

El operador ## concatena dos componentes léxicos. Considere la siguiente definición de macro:

```
#define TOKENCONCAT(x, y) x ## y
```

Cuando en el programa aparece TOKENCONCAT, sus argumentos son concatenados y utilizados para remplazar la macro. Por ejemplo, en el programa `TOKENCONCAT(O, K)` es remplazado por `OK`. El operador ## debe tener dos operandos.

13.8 Números de línea

La *directiva de preprocesador* `#line` hace que las líneas de código fuente subsecuentes se renumeren, iniciándose con el valor entero constante especificado. La directiva

```
#line 100
```

inicia la numeración de líneas a partir de `100`, empezando con la siguiente línea de código fuente. En la directiva `#line` puede incluirse un nombre de archivo. La directiva

```
#line 100 "file1.c"
```

indica que las líneas están numeradas a partir de **100** empezando desde la siguiente línea de código fuente, y el nombre del archivo para fines de cualquier mensaje de compilador es **"file1.c"**. Esta directiva normalmente se utiliza para ayudar a preparar los mensajes producidos debido a errores de sintaxis y a advertencias más significativas del compilador. En el archivo fuente no aparecen los números de línea.

13.9 Constantes simbólicas predefinidas

Existen cinco *constantes simbólicas predefinidas* (figura 13.1). Los identificadores correspondientes a cada una de las constantes simbólicas predefinidas empiezan y terminan con *dos* subrayados. Estos identificadores y el identificador **defined** (utilizado en la sección 13.5) no pueden ser utilizados en las directivas **#define** o **#undef**.

13.10 Asertos

La *macro* `assert` definida en el archivo de cabecera **assert.h**, prueba el valor de una expresión. Si el valor de la expresión es **0** (falso), entonces **assert** imprime un mensaje de error, y llama a la función *abort* (de la biblioteca general de utilerías **stdlib.h**) para terminar la ejecución del programa. Esta es una herramienta de depuración útil, para probar si una variable tiene el valor correcto. Por ejemplo, suponga que en un programa la variable **x** jamás debería ser mayor de **10**. Se podría utilizar una afirmación para probar el valor de **x** y si el valor de **x** es incorrecto imprimir un mensaje de error. El enunciado sería

```
assert(x <= 10);
```

Si cuando se encuentre este enunciado en un programa **x** es mayor que **10**, se imprimirá un mensaje de error, conteniendo el número de línea y el nombre del archivo, y el programa terminará. Entonces el programador podrá concentrarse en esta área del código y encontrar el error. Si queda definida la constante simbólica **NDEBUG**, asertos subsecuentes serán ignoradas. Entonces, una vez que ya no sean requeridas los asertos, la línea

Constante simbólica	Explicación
__LINE__	Número de línea de la línea actual del código fuente (una constante entera).
__FILE__	El nombre supuesto del archivo fuente (una cadena).
__DATE__	La fecha de compilación del archivo fuente (una cadena de la forma **"Mmm dd yyyy** como **"Jan 19 1991"**).
__TIME__	La hora en que fue compilado el archivo fuente (una literal de cadena de la forma **"hh:mm:ss"**).
__STDC__	La constante entera 1. Se utiliza para indicar que ésta puesta en marcha cumple con ANSI.

Fig. 13.1 Las constantes simbólicas predefinidas.


```
#define NDEBUG
```

es inserta en el archivo del programa, en vez de borrar de manera manual cada una de los asertos.

Resumen

- Todas las directivas de preprocesador empiezan con **#**.
- Sólo caracteres de espacio en blanco pueden aparecer en una línea antes de una directiva de preprocesador.
- La directiva **#include** incluye una copia del archivo especificado. Si el nombre del archivo está encerrado entre comillas, el preprocesador empieza buscando el archivo a incluirse en el mismo directorio que el archivo que se está compilando. Si el nombre del archivo está encerrado en corchetes angulares (< y >), la búsqueda se ejecuta de una forma definida de acuerdo a la puesta en marcha.
- La directiva de preprocesador **#define** se utiliza para crear constantes simbólicas y macros.
- Una constante simbólica es el nombre para una constante.
- Una macro es una operación definida en una directiva de preprocesador **#define**. Las macros pueden ser definidas con o sin argumentos.
- El texto de remplazo para una macro o una constante simbólica es cualquier texto que quede en la línea después del identificador en la directiva **#define**. Si el texto de remplazo para una macro o una constante simbólica es más largo que el resto de la línea, se coloca una diagonal invertida (\) al final de la línea, lo que indica que el texto de remplazo continúa sobre la línea siguiente.
- Las constantes simbólicas y las macros pueden ser descartadas utilizando la directiva de preprocesador **#undef**. La directiva **#undef** "elimina la definición" de la constante simbólica o del nombre de la macro.
- El alcance de una constante simbólica o de una macro es desde su definición hasta que queda eliminada su definición mediante **#undef**, o si no hasta el final del archivo.
- La compilación condicional le permite al programador controlar la ejecución de las directivas de preprocesador y la compilación del código de programa.
- Las directivas de preprocesador condicionales evalúan expresiones enteras constantes. Las expresiones de cambio de tipo, las expresiones **sizeof** y las constantes de enumeración no pueden ser evaluadas en directivas de preprocesador.
- Cada constructor **#if** termina con **#endif**.
- Las directivas **#ifdef** y **#ifndef** han sido diseñadas como abreviaturas de **#if defined**(*nombre*) y **#if !defined**(*nombre*).
- Un constructor de preprocesador condicional de varias partes puede ser probado utilizando **#elif** (el equivalente de **else if** en una estructura **if**) y **#else** (el equivalente de **else** en una estructura **if**) directivas.
- La directiva **#error** imprime un mensaje, que depende de la puesta en práctica, y que incluye las componentes léxicas especificadas en la directriz.
- La directiva **#pragma** genera una acción definida por la puesta en práctica. Si el pragma no es reconocido por la puesta en marcha, será ignorado.

- El operador # hace que un componente léxico de texto de remplazo, se convierta a una cadena encerrada entre comillas. El operador # debe ser utilizado en una macro con argumentos, porque el operando # debe ser un argumento de la macro.
- El operador ## concatena dos componentes léxicos. El operador ## debe de tener dos operandos.
- La directiva de preprocesador **#line** hace que las líneas de código fuente subsecuentes vuelvan a ser numeradas, iniciándose a partir del valor entero constante especificado.
- Existen cinco constantes simbólicas predefinidas. La constante **__LINE__** es el número de línea de la línea de código fuente actual (un entero). La constante **__FILE__** es el nombre supuesto del archivo (una cadena). La constante **__DATE__** es la fecha de compilación del archivo fuente (una cadena). La constante **__TIME__** es la hora de compilación del archivo fuente (una cadena). La constante **__STDC__** es **1**; su propósito es indicar que la instalación cumple con ANSI. Note que cada una de las constantes simbólicas predefinidas empieza y termina con dos subrayados.
- La macro **assert** definida en el archivo de cabecera **assert.h**, —prueba el valor de una expresión. Si el valor de la expresión es **0** (falsa), entonces **assert** imprime un mensaje de error y llama a la función **abort** para terminar la ejecución del programa.

Terminología

#define
#elif
#else
#endif
#error
#if
#ifdef
#ifndef
#include <filename>
#include "filename"
#line
#pragma
#undef
\ (diagonal invertida) carácter de continuación
__DATE__
__FILE__
__LINE__
__STDC__
__TIME__
a. out en UNIX
abort
argumento
assert
assert.h
preprocesador C
comando **cc** en UNIX
operador **##** de concatenación de preprocesador
compilación condicional
directrices de preprocesador de ejecución condicional
operador # de preprocesador de conversión a cadena
depurador
expandir una macro
archivo de cabecera
macro
macro con argumentos
constantes simbólicas predefinidas
directiva de preprocesador
texto de remplazo
alcance de una constante simbólica o de una macro
archivos de cabecera de la biblioteca estándar
stdio.h
stdlib.h
constante simbólica

Errores comunes de programación

13.1 Olvidar encerrar los argumentos de macro en paréntesis en el texto de remplazo.

13.2 Insertar enunciados **printf** compilados condicional para fines de depuración, en posiciones donde en ese momento C espera un enunciado. En este caso, el enunciado compilado condicional debería haberse encerrado en un enunciado compuesto. Entonces, cuando el programa se compile con enunciados de depuración, el flujo del control del programa no será alterado.

Práctica sana de programación

13.1 Utilizar nombres significativos para las constantes simbólicas, ayuda a que los programas resulten más autodocumentados.

Sugerencia de rendimiento

13.1 A veces las macros pueden ser utilizadas para substituir una llamada de función por código en línea, previo al tiempo de ejecución. Esto elimina la sobrecarga de una llamada de función.

Ejercicios de autoevaluación

13.1 Llene los espacios en blanco con cada uno de los siguientes:

a) Cada directiva de preprocesador deberá empezar con _____.
b) El constructor de compilación condicional puede ser extendido para probar casos múltiples utilizando las directivas _____ y _____.
c) La directiva _____ genera macros y constantes simbólicas.
d) Sólo caracteres _____ pueden aparecer sobre una línea antes de una directiva de preprocesador.
e) La directiva _____ descarta constantes simbólicas y nombres de macros.
f) Las directivas _____ y _____ son una notación abreviada en lugar de **#if defined (*nombre*)** y **#if !defined (*nombre*)**.
g) _____ le permite al programador controlar la ejecución de las directivas de preprocesador, y la compilación del código del programa.
h) La macro _____ imprime un mensaje y termina la ejecución de un programa si el valor de la expresión que la macro evalúa es 0.
i) La directiva _____ inserta un archivo en otro archivo.
j) El operador _____ concatena sus dos argumentos.
k) El operador _____ convierte su operando a una cadena.
l) El carácter _____ indica que el texto de remplazo de una constante simbólica o de una macro continúa en la línea siguiente.
m) La directiva _____ hace que las líneas de código fuente se enumeren a partir del valor indicado, empezando con la siguiente línea de código fuente.

13.2 Escriba un programa que imprima los valores de las constantes simbólicas predefinidas, listadas en la figura 13.1.

13.3 Escriba una directiva de preprocesador para que lleve a cabo cada una de las siguientes:

a) Definir la constante simbólica **YES** para que tenga el valor **1**.
b) Definir la constante simbólica **NO** para que tenga el valor **0**.
c) Incluir el archivo de cabecera **common.h**. El archivo de cabecera se encuentra en el mismo directorio que el archivo bajo compilación.
d) Renumerar las líneas restantes del archivo, empezando con el número de línea **3000**.
e) Si está definida la constante simbólica **TRUE**, elimine su definición, y vuélvala a definir como **1**. No utilice **#ifdef**.
f) Si la constante simbólica **TRUE** está definida, elimine su definición, y vuélvala a definir como **1**. Utilice la directiva de preprocesador **#ifdef**.

g) Si la constante simbólica **TRUE** no es igual a 0, defina la constante simbólica **FALSE** como 0. De lo contrario defina **FALSE** como 1.
h) Defina la macro **SQUARE_VOLUME**, que calcula el volumen de un cuadrado. La macro toma un argumento.

Respuesta a los ejercicios de autoevaluación

13.1 a) #. b) **#elif**, **#else**. c) **#define**. d) espacio en blanco. e) **#undef**. f) **#ifdef**, **#ifndef**. g) compilación condicional. h) **assert**. i) **#include**. j) ##. k) #. l) \. m) **#line**.

13.2
```
/* Print the values of the predefined macros */
#include <stdio.h>
main()
{
   printf("__LINE__ = %d\n", __LINE__);
   printf("__FILE__ = %s\n", __FILE__);
   printf("__DATE__ = %s\n", __DATE__);
   printf("__TIME__ = %s\n", __TIME__);
   printf("__STDC__ = %d\n", __STDC__);
}
```

```
__LINE__ = 5
__FILE__ = macros.c
__DATE__ = Sep 08 1993
__TIME__ = 10:23:47
__STDC__ = 1
```

13.3 a)
```
#define YES 1
```
b)
```
#define NO 0
```
c)
```
#include "common.h"
```
d)
```
#line 3000
```
e)
```
#if defined(TRUE)
   #undef TRUE
   #define TRUE 1
#endif
```
f)
```
#ifdef TRUE
   #undef TRUE
   #define TRUE 1
#endif
```
g)
```
#if TRUE
   #define FALSE 0
#else
    #define FALSE 1
#endif
```
h)
```
#define SQUARE_VOLUME(x)  (x) * (x) * (x)
```

Ejercicios

13.4 Escriba un programa que defina una macro con un argumento para calcular el volumen de una esfera. El programa deberá calcular el volumen para esferas de radios de 1 a 10, e imprimir los resultados en formato tabular. La fórmula del volumen de una esfera es:

$$(4/3) * \pi * r^3$$

donde π es **13.14159**

13.5 Escriba un programa que produzca la siguiente salida:

```
The sum of x and y is 13
```

El programa deberá definir la macro **SUM** con dos argumentos, **x** y **y**, y utilizar **SUM** para producir la salida.

13.6 Escriba un programa que utilice la macro **MINIMUM2** para determinar el más pequeño de dos valores numéricos. Introduzca los valores desde el teclado.

13.7 Escriba un programa que utilice la macro **MINIMUM3** para determinar el más pequeño de tres valores numéricos. La macro **MINIMUM3** deberá utilizar la macro **MINIMUM2** definida en el ejercicio 13.6 para determinar el valor más pequeño. Introduzca los valores desde el teclado.

13.8 Escriba un programa que utilice la macro **PRINT** para imprimir un valor de cadena.

13.9 Escriba un programa que utilice la macro **PRINTARRAY** para imprimir un arreglo de enteros. La macro deberá recibir como argumentos el arreglo y el número de elementos en el arreglo.

13.10 Escriba un programa que utilice la macro **SUMARRAY** para sumar los valores de un arreglo numérico. La macro deberá recibir como argumentos el arreglo y el número de elementos en el arreglo.

14

Temas avanzados

Objetivos

- Ser capaz de redirigir la entrada de teclado para que provenga de un archivo.
- Ser capaz de redirigir la salida a pantalla a un archivo.
- Ser capaz de escribir funciones que utilicen listas de argumentos de longitud variable.
- Ser capaz de procesar argumentos en la línea de comandos.
- Ser capaz de asignar tipos específicos a constantes numéricas.
- Ser capaz de utilizar archivos temporales.
- Ser capaz de procesar eventos inesperados dentro de un programa.
- Ser capaz de asignar memoria dinámicamente para los arreglos.
- Ser capaz de cambiar el tamaño de la memoria anteriormente dinámica asignada.

Utilizaremos una señal que ya he probado y comprobado llega lejos y es fácil de gritar. ¡Waa-hoo!
Zane Grey

Uselo, gástelo;
aprovéchelo o descártelo.
Anónimo

Verdaderamente es un problema de tres pipas.
Sir Arthur Conan Doyle

Sinopsis

Resumen • Terminología • Errores comunes de programación • Sugerencias de portabilidad • Sugerencias de rendimiento • Observaciones de ingeniería de software • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios.

14.1 Introducción

En esta capítulo se presentan varios temas avanzados, por lo regular no se tratan en cursos introductorios. Muchas de las capacidades que aquí se analizan son específicas a sistemas operativos particulares, en especial a UNIX y/o a DOS.

14.2 Cómo redirigir entradas/salidas en sistemas UNIX y DOS

Normalmente la entrada a un programa es a partir del teclado (entrada estándar), y la salida de un programa se muestra en la pantalla (salida estándar). En la mayor parte de los sistemas de computación y en particular en los sistemas UNIX y DOS es posible *redirigir* las entradas, para que en vez del teclado provengan de un archivo, y redirigir las salidas, para que en vez de en la pantalla sean colocadas en un archivo. Ambas formas de redirección pueden ser llevadas a cabo sin tener que utilizar las capacidades de procesamiento de archivos de la biblioteca estándar.

A partir de la línea de comandos UNIX existen varias maneras para redirigir la entrada y la salida. Considere al archivo ejecutable **sum.** que introduce enteros uno por uno, y lleva un total acumulativo de los valores hasta que se define el indicador de fin de archivo, y a continuación imprime el resultado. Por lo regular el usuario introduce enteros a partir del teclado y escribe la combinación de teclas de fin de archivo, para indicar que no se introducirán más valores. Con la redirección de la entrada, la entrada puede estar almacenada en un archivo. Por ejemplo, si los datos están almacenados en el archivo **input**, la línea de comando


```
$ sum < input
```

hace que se ejecute el programa **sum**; el *símbolo de redirección de entrada (<)* indica que los datos en el archivo **input** deben ser utilizados como entrada para el programa. En un sistema DOS la redirección de la entrada se efectúa en forma idéntica.

Note que en UNIX la indicación de la línea de comandos es el signo **$** (algunos sistemas UNIX utilizan la indicación %). Los estudiantes encuentran a menudo difícil comprender que la redirección es una función del sistema operativo, y no otra característica de C.

El segundo método para redirigir la entrada es *el entubamiento*. Una *tubería (|)* hace que la salida de un programa sea redirigida como entrada de otro programa. Suponga que el programa **random** tiene como salida una serie de enteros al azar; la salida de **random** puede ser "entubada" en forma directa al programa **sum**, utilizando la línea de comandos UNIX

```
$ random | sum
```

Esto hace que se calcule la suma de los enteros producidos por **random**. El entubamiento puede ser ejecutado en UNIX y en DOS.

La salida de programa puede ser redirigida a un archivo, mediante el uso del símbolo de *redirección de salida (>)* (tanto en UNIX como en DOS se utiliza el mismo símbolo). Por ejemplo, para redirigir la salida del programa **random** al archivo **out**, utilice

```
$ random > out
```

Por último, la salida del programa puede ser agregada al final de un archivo existente utilizando el *símbolo de agregar salida (>>)* (tanto en UNIX como en DOS se utiliza el mismo símbolo). Por ejemplo, para agregar la salida del programa **random** al archivo **out**, creado en la línea de comando anterior, utilice la línea de comando

```
$ random >> out
```

14.3 Listas de argumentos de longitud variable

Es posible crear funciones que reciban un número no especificado de argumentos. La mayor parte de los programas en el texto han utilizado la función **printf** estándar de biblioteca la cual, como usted sabe, toma un número variable de argumentos. Como mínimo, **printf** debe recibir una cadena como primer argumento, pero **printf** puede recibir cualquier número de argumentos adicionales. El prototipo de función correspondiente a **printf** es

```
int printf(const char *format, ...);
```

En el prototipo de función los puntos suspensivos (**...**) indican que la función recibe un número variable de argumentos de cualquier tipo. Note que los puntos suspensivos deberán siempre estar colocados al final de la lista de parámetros.

Los macros y definiciones del *encabezado de argumentos variables* **stdarg.h** (figura 14.1) proveen las capacidades necesarias para construir funciones con listas de argumentos de longitud variable. El programa de la figura 14.2 demuestra una función **average**, que recibe un número variable de argumentos. El primer argumento de **average** es siempre el número de valores a promediarse.

UNIVERSIDAD DE LA REPUBLICA
FACULTAD DE INGENIERIA
DEPARTAMENTO

Identificador	Explicación
va_list	Un tipo utilizable para conservar información necesaria por las macros **va_start**, **va_arg**, y **va_end**. Para tener acceso a los argumentos en una lista de argumentos de longitud variable, debe declararse un objeto del tipo **va_list**.
va_start	Una macro que se invoca antes de tener acceso a los argumentos de una lista de argumentos de longitud variable. La macro inicializa el objeto declarado mediante **va_list**, para que sea utilizada con las macros **va_arg**, y **va_end**.
va_arg	Una macro que se expande a una expresión del valor y del tipo del siguiente argumento, existente en la lista de argumentos de longitud variable. Cada invocación de **va_arg** modifica el objeto declarado con **va_list**, de tal forma que el objeto señale al siguiente argumento dentro de la lista.
va_end	Una macro que facilita el regreso normal de una función cuya lista de argumentos de longitud variable ha sido referenciada por la macro **va_start**.

Fig. 14.1 El tipo y las macros definidas en el encabezado **stdarg.h**.

La función **average** utiliza todas las definiciones y macros del encabezado **stdarg.h**. El objeto **ap**, del tipo **va_list**, es utilizado por las macros **va_start**, **va_arg** y **va_end** para procesar la lista de argumentos de longitud variable de la función **average**. La función empieza invocando la macro **va_start** para inicializar el objeto **ap** para su uso en **va_arg** y **va_end**. La macro recibe dos argumentos —el objeto **ap** y el identificador al argumento más a la derecha dentro de la lista de argumentos, antes de los puntos suspensivos— en este caso **i** (aquí **va_start** utilizará **i** para determinar dónde empieza la lista de argumentos de longitud variable). A continuación la función **average** añade en forma repetida los argumentos en la lista de argumentos de longitud variable de la variable **total**. El valor a añadirse a **total** se toma de la lista de argumentos invocando la macro **va_arg**. La macro **va_arg** recibe dos argumentos: el objeto **ap**, y el tipo de valor esperado en la lista de argumentos, en este caso **double**. La macro regresa el valor del argumento. La función **average** invoca la macro **va_end**, con el objeto **ap** como argumento, para facilitar un regreso normal hacia **main** a partir de **average**. Por último, se calcula el promedio y se regresa a **main**.

Error común de programación 14.1

Colocar puntos suspensivos en la mitad de una lista de parámetros de función. Los puntos suspensivos sólo pueden ser colocados al final de la lista de parámetros.

El lector puede cuestionar cómo pueden saber las funciones **printf** y **scanf** qué tipo utilizar en cada macro **va_arg**. La respuesta es que **printf** y **scanf** rastrean los especificadores de conversión de formato existentes en la cadena de control de formato, a fin de determinar el tipo del siguiente argumento a procesar.


```
/* Using variable-length argument lists */

#include <stdio.h>
#include <stdarg.h>

double average(int, ...);

main()
{
   double w = 37.5, x = 22.5, y = 1.7, z = 10.2;

   printf("%s%.1f\n%s%.1f\n%s%.1f\n%s%.1f\n\n",
          "w = ", w, "x = ", x, "y = ", y, "z = ", z);
   printf("%s%.3f\n%s%.3f\n%s%.3f\n",
          "The average of w and x is ",
          average(2, w, x),
          "The average of w, x, and y is ",
          average(3, w, x, y),
          "The average of w, x, y, and z is ",
          average(4, w, x, y, z));

   return 0;
}

double average(int i, ...)
{
   double total = 0;
   int j;
   va_list ap;

   va_start(ap, i);

   for (j = 1; j <= i; j++)
      total += va_arg(ap, double);

   va_end(ap);
   return total / i;
}
```

```
w = 37.5
x = 22.5
y = 1.7
z = 10.2

The average of w and x is 30.000
The average of w, x, and y is 20.567
The average of w, x, y, and z is 17.975
```

Fig. 14.2 Cómo utilizar listas de argumentos de longitud variable.

14.4 Cómo utilizar argumentos en la línea de comandos

En muchos sistemas y en particular en DOS y en UNIX incluyendo los parámetros **int argc** y **char *argv[]** en la lista de parámetros de **main**, es posible pasar argumentos a **main** a partir de la línea de comandos. El parámetro **argc** recibe el número de argumentos de la línea de comando. El parámetro **argv** es un arreglo de cadenas, en el cual se almacenan los argumentos de la línea de comando reales. Usos comunes de los argumentos en la línea de comandos incluyen la impresión de los mismos, pasar opciones a un programa, y pasar nombres de archivo a un programa.

El programa de la figura 14.3 copia un archivo a otro archivo, carácter por carácter. El archivo ejecutable para el programa se llama **copy**. En un sistema UNIX una línea de comando típica para el programa **copy** es

```
$ copy input output
```

Esta línea de comando indica que el archivo **input** será copiado al archivo **output**. Cuando se ejecuta el programa, si **argc** no es **3** (**copy** cuenta como uno de los argumentos), el programa imprime un mensaje de error y termina. De lo contrario, **argv** contiene las cadenas **"copy"**, **"input"**, y **"output"**. Los argumentos segundos y terceros de la línea de comandos se utilizan como nombres de archivo por el programa. Los archivos se abren utilizando la función **fopen**. Si ambos archivos se abren con éxito, los caracteres se leen del archivo **input** y se escriben al archivo **output** hasta que el indicador de fin de archivo del archivo **input** queda definido. Entonces termina el programa. El resultado es una copia exacta del archivo **input**. Note que no todos los sistemas de cómputo aceptan argumentos en la línea de comandos tan fácil como UNIX y DOS. Por ejemplo, los sistemas Macintosh y VMS, requieren de ajustes especiales para el proceso de argumentos en la línea de comandos. Vea los manuales de su sistema para más información relativa a argumentos en la línea de comandos.

14.5 Notas sobre la compilación de programas con varios archivos fuente

Como se indicó en el texto, es posible construir programas que estén formados por múltiples archivos fuente (vea el capítulo 16, "Clases"). Existen varias consideraciones a tomar en cuenta al crear programas en múltiples archivos. Por ejemplo, la definición de una función deberá estar por completo contenida en un archivo —no puede estar contenida en dos o más archivos.

En el capítulo 5, presentamos los conceptos de clase de almacenamiento y alcance. Aprendimos que las variables declaradas por afuera de cualquier definición de función son por omisión de clase de almacenamiento estática, y se conocen como variables globales. Una vez declaradas, las variables globales son accesibles a cualquier función definida en el mismo archivo. Las variables globales también son accesibles a funciones en otros archivos, pero sin embargo, estas variables globales deberán ser declaradas en cada uno de los archivos en los cuales serán utilizadas. Por ejemplo, si en un archivo definimos la variable entera global **flag**, y en un segundo archivo nos referimos a ella, el segundo archivo deberá contener la declaración.

```
extern int flag;
```

antes del uso de la variable en dicho archivo. En la declaración anterior, el especificador de clase de almacenamiento **extern** le indica al compilador que la variable **flag** está definida más adelante en el mismo archivo o en uno distinto. El compilador informa al enlazador que en dicho archivo aparecen referencias a la variable **flag** sin resolver (el compilador no sabe donde **flag**


```
/* Using command-line arguments */
#include <stdio.h>

main(int argc, char *argv[])
{
   FILE *inFilePtr, *outFilePtr;
   int c;

   if (argc != 3)
      printf("Usage: copy infile outfile\n");
   else
      if ((inFilePtr = fopen(argv[1], "r")) != NULL)

         if ((outFilePtr = fopen(argv[2], "w")) != NULL)

            while ((c = fgetc(inFilePtr)) != EOF)
               fputc(c, outFilePtr);

         else
            printf("File \"%s\" could not be opened\n", argv[2]);

      else
         printf("File \"%s\" could not be opened\n", argv[1]);

   return 0;
}
```

Fig. 14.3 Cómo utilizar argumentos en la línea de comandos.

queda definida, por lo que le deja al enlazador intentar encontrar a **flag**). Si el enlazador no puede localizar la definición de **flag**, se genera un error de enlace, y no se produce un archivo ejecutable. Si el enlazador localiza una definición global apropiada, éste resuelve las referencias indicando donde está localizada **flag**.

Sugerencia de rendimiento 14.1

Las variables globales aumentan el rendimiento debido a que son de manera directa accesibles por cualquier función —se elimina la sobrecarga de pasar datos a función.

Observación de ingeniería de software 14.1

Las globales variables deberían de evitarse, a menos de que el rendimiento de la aplicación sea crítico, porque violan el principio del mínimo privilegio.

Igual que las declaraciones **extern** pueden ser utilizadas para declarar variables globales en otros archivos de programa, los prototipos de función pueden extender el alcance de una función más allá del archivo en el cual ha sido definida (en un prototipo de función no es requerido el especificador **extern**). Esto se lleva a cabo incluyendo el prototipo de función en cada uno de los archivos en los cuales dicha función fue invocada, y compilando ambos archivos juntos (vea la sección 13.2). Los prototipos de función le indican al compilador que la función especificada se define más adelante en el mismo archivo o en un archivo distinto. De nuevo, el compilador no intenta resolver referencias a dicha función esa tarea se le deja al enlazador. Si el enlazador no puede localizar una definición de función apropiada, se genera un error.

Como ejemplo del uso de prototipos de función para extender el alcance de una función, considere cualquier programa que contenga la directiva de preprocesador **#include <stdio.h>**. Esta directiva incluye en un archivo los prototipos de función para funciones tales como **printf** y **scanf**. Otras funciones en el archivo pueden utilizar **printf** y **scanf** para llevar a cabo sus tareas. Las funciones **printf** y **scanf** se definen para nosotros en forma independiente. No necesitamos saber dónde están definidas. Estamos sólo volviendo a utilizar dicho código en nuestros programas. El enlazador resuelve nuestras referencias a dichas funciones en forma automática. Este proceso nos permite utilizar las funciones estándar de biblioteca.

Observación de ingeniería de software 14.2

La creación de programas en múltiples archivos fuente facilita la reutilización del software y buena ingeniería de software. Las funciones pueden ser comunes para muchas aplicaciones. En casos como éstos, estas funciones deberán ser almacenadas en sus propios archivos fuente, y cada archivo fuente deberá tener su correspondiente archivo de cabecera, que contenga los prototipos de función. Esto permite a los programadores de distintas aplicaciones para reutilizar el mismo código, mediante la inclusión del archivo de cabecera apropiado, y compilar su aplicación con el archivo fuente correspondiente.

Sugerencia de portabilidad 14.1

Algunos sistemas no aceptan nombres de variables globales o nombres de funciones con más de 6 caracteres. Esto deberá ser tomado en cuenta al escribir programas que se planea serán transportados a varias plataformas.

Es posible restringir el alcance de una variable global o de una función al archivo en el cual está definido. El especificador de clase de almacenamiento **static**, al ser aplicado a una variable global o a una función, impide que sea utilizada por cualquier función que no quede definida dentro del mismo archivo. Esto se conoce como *enlace interno*. Las variables globales y las funciones que en sus definiciones no estén precedidas por **static** tienen *enlace externo* —se puede tener acceso a ellas desde otros archivos, si dichos archivos contienen las declaraciones apropiadas y/o los prototipos de función.

La declaración de variable global

```
static float pi = 3.14159;
```

crea la variable **pi** del tipo **float**, la inicializa a **3.14159**, e indica que **pi** es conocida sólo por las funciones dentro del archivo en la cual está definida.

El especificador **static** se utiliza por lo general con funciones de utilería, que sólo son llamadas por funciones de un archivo en particular. Si fuera de un archivo en particular una función no es requerida, deberá cumplirse mediante el uso de **static** con el principio del mínimo privilegio. Si en un archivo una función se define antes de su uso, **static** deberá ser aplicado a la definición de función. De lo contrario, **static** deberá aplicarse a el prototipo de función.

Al construir programas extensos en múltiples archivos fuente, la compilación del programa se convierte en tediosa si se hacen pequeñas modificaciones a un archivo y todo el programa debe ser vuelto a compilar. Muchos sistemas tienen utilerías especiales, que recopilan sólo el archivo de programa modificado. En sistemas UNIX esta utilería se denomina ***make***. La utilería **make** lee un archivo llamado ***makefile***, que contiene instrucciones para compilar y enlazar el programa. Los sistemas como Borland C++ y Microsoft C/C++ 7.0 para PC también ofrecen utilerías **make**. Para mayor información sobre las utilerías **make**, vea el manual correspondiente a su sistema particular.

14.6 Terminación de programa mediante Exit y Atexit

La biblioteca general de utilerías (**stdlib.h**) contiene métodos de terminar la ejecución de programa distintos al regreso convencional a partir de la función **main**. La función ***exit*** obliga a que se termine un programa, como si se hubiera ejecutado en forma normal. Esta función se utiliza a menudo para terminar un programa cuando se detecta un error en la entrada o si un archivo a procesarse por el programa no puede ser abierto. La función ***atexit*** *registra* en el programa, una función, que a la terminación exitosa del programa será llamada, es decir, ya sea cuando termina el programa llegando al final de **main**, o cuando se invoca a **exit**.

La función **atexit** toma como argumento un apuntador a una función (es decir, el nombre de la función). Las funciones llamadas a la terminación del programa no pueden tener argumentos, y no pueden regresar un valor. Se pueden registrar hasta 32 funciones para ejecución a la terminación del programa.

La función **exit** toma un argumento. El argumento por lo regular es o la constante simbólica ***EXIT_SUCCESS*** o la constante simbólica ***EXIT_FAILURE***. Si **exit** es llamada mediante ***EXIT_SUCCESS***, el valor definido por la puesta en marcha para una terminación exitosa es regresado al entorno llamador. Si **exit** es llamado mediante **EXIT_FAILURE**, el valor definido por la puesta en marcha correspondiente a una terminación no exitosa es regresado. Cuando se invoca la función **exit**, cualesquiera funciones registradas ya con **atexit** son invocadas en orden inverso de su registro, son vaciados y cerrados todos los flujos asociados con el programa, y el control regresa al entorno huésped. El programa de la figura 14.4 prueba las funciones **exit** y **atexit**. El programa solicita al usuario que determine si el programa debe terminarse con **exit** o llegando al final de **main**. Note que en cada caso la función **print** se ejecuta a la terminación del programa.

14.7 El calificador de tipo volátil

En los capítulos 6 y 7, presentamos el calificador de tipo **const**. ANSI C también dispone del calificador de tipo **volatile**. El estándar ANSI (An90) indica que cuando se utiliza **volatile** para calificar un tipo, la naturaleza del acceso a un objeto de dicho tipo dependerá de la puesta en marcha. Kernighan y Ritchie (Ke88) indican que el calificador **volatile** se utiliza para suprimir varios tipos de optimizaciones.

14.8 Sufijos para constantes enteras y de punto flotante

C contiene sufijos enteros y de punto flotante para especificar los tipos de las constantes enteras y de punto flotante. Los sufijos enteros son: **u** o bien **U** para un entero **unsigned**, **l** o **L** para un entero **long** y **ul** o **UL** para un entero **unsigned long**. Las siguientes constantes son del tipo **unsigned, long** y **unsigned long** respectivamente:

```
174u
8358L
28373ul
```

Si una constante entera no tiene sufijo, su tipo queda determinado con el primer tipo capaz de almacenar un valor de dicho tamaño (primero **int**, después **long int**, y por último **unsigned long int**).

Los sufijos de punto flotante son: **f** o **F** para un **float**, y **l** o **L** para un **long double**. Las siguientes constantes son del tipo **long double** y **float**, respectivamente:

```
/* Using the exit and atexit functions */

#include <stdio.h>
#include <stdlib.h>

void print(void);

main()
{
   int answer;

   atexit(print);       /* register function print */
   printf("Enter 1 to terminate program with function exit\n"
          "Enter 2 to terminate program normally\n");
   scanf("%d", &answer);

   if (answer == 1) {
      printf("\nTerminating program with function exit\n");
      exit(EXIT_SUCCESS);
   }

   printf("\nTerminating program by reaching the end of main\n");
   return 0;
}

void print(void)
{
   printf("Executing function print at program termination\n"
          "Program terminated\n");
}
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
: 1

Terminating program with function exit
Executing function print at program termination
Program terminated
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
: 2

Terminating program by reaching the end of main
Executing function print at program termination
Program terminated
```

Fig. 14.4 Cómo utilizar las funciones **exit** y **atexit**.


```
3.14159L
1.28f
```

Una constante de punto flotante sin sufijo automáticamente es del tipo **double**.

14.9 Más sobre archivos

En el capítulo 11, se presentaron las capacidades para procesar archivos de texto mediante acceso secuencial y acceso directo. C también proporciona capacidades para el proceso de archivos binarios, pero algunos sistemas de cómputo no aceptan archivos binarios. Si no son aceptados los archivos binarios, y un archivo es abierto en modo de archivo binario (figura 14.5), el archivo será procesado como un archivo de texto. Los archivos binarios deberán ser utilizados en lugar de los archivos de texto, sólo en aquellas situaciones en que las condiciones de velocidad, almacenamiento y/o compatibilidad demanden archivos binarios. De lo contrario, siempre los archivos de texto deberán preferirse, debido a su inherente portabilidad, y debido a la posibilidad de utilizar otras herramientas estándar para examinar y manipular los datos del archivo.

Sugerencia de rendimiento 14.2

Considere utilizar archivos binarios en lugar de archivos de texto en aquellas aplicaciones que exijan alto rendimiento

Modo	Descripción
rb	Abre un archivo binario para lectura
wb	Crea un archivo binario para escritura. Si el archivo ya existe, descartará el contenido actual.
ab	Agrega; abre o crea un archivo binario para escritura al final del archivo.
rb+	Abre un archivo binario para actualizar (lectura y escritura).
wb+	Crea un archivo binario para actualizar. Si el archivo ya existe, descartará el contenido actual.
ab+	Agrega; abre o crea un archivo binario para actualizar; toda la escritura se efectuará al final del archivo.

Fig. 14.5 Modos de archivo binario abierto.

Sugerencia de portabilidad 14.2

Cuando esté escribiendo programas portátiles utilice archivos de texto.

Además de las funciones de procesamiento de archivos analizadas en el capítulo 11, la biblioteca estándar también tiene la función **tmpfile**, que abre un archivo temporal en modo **"wb+"**. Aunque este es un modo de archivo binario, algunos sistemas procesan los archivos temporales como archivos de texto. Un archivo temporal existe sólo hasta que es cerrado con **fclose**, o bien hasta que el programa termina.

El programa de la figura 14.6 sustituye los tabuladores existentes en un archivo por espacios. El programa solicita al usuario que escriba el nombre del archivo a modificarse. Si tanto el archivo

escrito por el usuario como el archivo temporal son abiertos con éxito, el programa lee caracteres del archivo a modificarse, y los escribe en el archivo temporal. Si el carácter leído es un tabulador (**'\t'**), será remplazado por un espacio y será escrito al archivo temporal. Cuando se llegue al fin del archivo bajo modificación, utilizando **rewind** los apuntadores de archivo para cada archivo serán reposicionados al principio de cada uno. A continuación, el archivo temporal se copia al archivo original, carácter por carácter. Con el fin de confirmar que los caracteres escritos son correctos, el programa imprime el archivo original conforme copia caracteres al archivo temporal, así como imprime el nuevo archivo conforme copia caracteres del archivo temporal al archivo original.

```
/* Using temporary files */
#include <stdio.h>

main()
{
   FILE *filePtr, *tempFilePtr;
   int c;
   char fileName[30];

   printf("This program changes tabs to spaces.\n"
          "Enter a file to be modified: ");
   scanf("%s", fileName);

   if ( ( filePtr = fopen(fileName, "r+") ) != NULL)

      if ( ( tempFilePtr = tmpfile() ) != NULL) {
         printf("\nThe file before modification is:\n");

         while ( ( c = getc(filePtr) ) != EOF) {
            putchar(c);
            putc(c == '\t' ? ' ': c, tempFilePtr);
         }

         rewind(tempFilePtr);
         rewind(filePtr);
         printf("\n\nThe file after modification is:\n");

         while ( ( c = getc(tempFilePtr) ) != EOF) {
            putchar(c);
            putc(c, filePtr);
         }

      }
      else
         printf("Unable to open temporary file\n");

   else
      printf("Unable to open %s\n", fileName);

   return 0;
}
```

Fig. 14.6 Cómo utilizar los archivos temporales (parte 1 de 2).


```
This program changes tabs to spaces.
Enter a file to be modified: data

The file before modification is:
0       1       2       3       4
        5       6       7       8       9

The file after modification is:
0 1 2 3 4
 5 6 7 8 9
```

Fig. 14.6 Cómo utilizar los archivos temporales (parte 2 de 2).

14.10 Manejo de señales

Un evento inesperado o *señal*, puede hacer que termine un programa en forma prematura. Algunos eventos inesperados son *interrupciones* (escribir **<ctrl> c** en un sistema UNIX o en DOS), *instrucciones ilegales, violaciones de segmentación, órdenes de terminación provenientes del sistema operativo* y *excepciones de punto flotante* (división entre cero o la multiplicación de valores de punto flotante grandes). Mediante la función ***signal***, la *biblioteca de manejo de señales* da la capacidad de *atrapar* eventos inesperados. La función **signal** recibe dos argumentos un número de señal entero y un apuntador a la función de manejo de señal. Las señales pueden ser generadas por la función ***raise*** que toma como argumento un número de señal entero. En la figura 14.7 se resumen las señales estándar, definidas en el archivo de cabecera ***signal.h***. El programa de la figura 14.8 demuestra el uso de las funciones **signal** y **raise**.

El programa de la figura 14.8 utiliza la función **signal** para atrapar una señal interactiva (**SIGINT**). El programa empieza llamando **signal** mediante **SIGINT** y un apuntador a la función **signal_handler** (recuerde que el nombre de una función es un apuntador al principio de la misma). Cuando se genera una señal del tipo **SIGINT**, se pasa el control a la función **signal_handler**, se imprime un mensaje y se le da al usuario la opción de continuar con la

Señal	Explicación
SIGABRT	Terminación anormal del programa (como una llamada a la función **abort**).
SIGFPE	Una operación aritmética errónea, como sería una división entre cero o una operación que resulte en un desbordamiento.
SIGILL	Detección de una instrucción ilegal.
SIGINT	Recepción de una señal de atención interactiva.
SIGSEGV	Un acceso inválido a almacenamiento.
SIGTERM	Una solicitud de terminación definida al programa.

Fig. 14.7 Las señales definidas en el encabezado **signal.h**.

ejecución normal del programa. Si el usuario desea continuar con la ejecución, el manejador de señal es reinicializado, llamando otra vez a **signal** (algunos sistemas requieren que el manejador de señal sea reinicializado), y el control regresa al punto en el programa donde la señal fue detectada. En este programa, se utiliza la función **raise** para simular una señal interactiva. Se escoge un número al azar entre **1** y **50**. Si el número es **25**, entonces se llama a **raise** para generar la señal. Por lo regular, la señales interactivas son iniciadas desde fuera del programa. Por ejemplo, si en un sistema UNIX o DOS se escribe **<ctrl> c** durante la ejecución del programa, se genera una señal interactiva, que termina la ejecución del programa. El manejo de señales puede ser utilizado para atrapar una señal interactiva y evitar que el programa se dé por terminado.

14.11 Asignación dinámica de memoria: funciones Calloc y Realloc

En el capítulo 12, "Estructuras de datos", se presentó la noción de la asignación dinámica de la memoria, mediante el uso de la función **malloc**. Como se explicó en el capítulo 12, tratándose de ordenamiento rápido, búsqueda y acceso a los datos los arreglos resultan mejores que las listas enlazadas. Sin embargo, por lo regular los arreglos son *estructuras estáticas de datos*. Para la asignación dinámica de memoria, la biblioteca general de utilerías (**stdlib.h**) contiene otras dos funciones —**calloc** y **realloc**. Estas funciones pueden ser utilizadas para crear y modificar *arreglos dinámicos*. Como fue mostrado en el capítulo 7, "Apuntadores", un apuntador a un arreglo puede ser marcado con un subíndice, de la misma forma que un arreglo. Entonces, un apuntador a una porción contigua de memoria generada por **calloc**, puede ser manipulado como si fuera un arreglo. La función **calloc** asigna de forma dinámica la memoria para un arreglo. El prototipo para **calloc** es

```
void *calloc(size_t nmemb, size_t size);
```

Recibe dos argumentos —el número de elementos (**nmemb**) y el tamaño de cada elemento (**size**)— e inicializa los elementos del arreglo a cero. La función regresa un apuntador a la memoria asignada, o un apuntador **NULL** si no se asigna memoria.

La función **realloc** modifica el tamaño de un objeto asignado mediante una llamada a **malloc, calloc** o **realloc** anterior. Siempre y cuando la cantidad de memoria asignada sea mayor que la cantidad ya asignada, el contenido del objeto original no es modificado. De lo contrario, se conservará el contenido sin modificación hasta el tamaño del nuevo objeto. El prototipo correspondiente a **realloc** es

```
void *realloc(void *ptr, size_t size);
```

La función **realloc** toma dos argumentos un apuntador al objeto original (**ptr**) y el nuevo tamaño del objeto (**size**). Si **ptr** es **NUL**, **realloc** funciona en forma idéntica a **malloc**. Si **size** es **0** y **ptr** no es **NULL**, se libera la memoria correspondiente al objeto. De lo contrario, si **ptr** no es **NULL** y el tamaño es mayor que **0**, **realloc** intenta asignar un nuevo bloque de memoria para el objeto. Si el nuevo espacio no puede ser asignado, el objeto al cual señala **ptr** se conserva sin modificar. La función **realloc** regresa ya sea un apuntador a la memoria reasignada, o un apuntador **NULL**.

14.12 La bifurcación incondicional: Goto

A lo largo del texto hemos insistido en la importancia de utilizar técnicas de programación estructurada para construir software confiable, que resulte fácil de depurar, mantener y modificar. En algunos casos, el rendimiento es de mayor importancia que una estricta adherencia a las


```
/* Using signal handling */

#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <time.h>

void signal_handler(int);

main()
{
   int i, x;

   signal(SIGINT, signal_handler);
   srand(clock());

   for (i = 1; i <= 100; i++) {
      x = 1 + rand() % 50;

      if (x == 25)
         raise(SIGINT);

      printf("%4d", i);

      if (i % 10 == 0)
         printf("\n");
   }

   return 0;
}

void signal_handler(int signalValue)
{
   int response;

   printf("%s%d%s\n%s",
          "\nInterrupt signal (", signalValue, ") received.",
          "Do you wish to continue (1 = yes or 2 = no)? ");

   scanf("%d", &response);

   while (response != 1 && response != 2) {
      printf("(1 = yes or 2 = no)? ");
      scanf("%d", &response);
   }

   if (response == 1)
      signal(SIGINT, signal_handler);
   else
      exit(EXIT_SUCCESS);
}
```

Fig. 14.8 Cómo utilizar el manejo de señales (parte 1 de 2).

```
   1   2   3   4   5   6   7   8   9  10
  11  12  13  14  15  16  17  18  19  20
  21  22  23  24  25  26  27  28  29  30
  31  32  33  34  35  36  37  38  39  40
  41  42  43  44  45  46  47  48  49  50
  51  52  53  54  55  56  57  58  59  60
  61  62  63  64  65  66  67  68  69  70
  71  72  73  74  75  76  77  78  79  80
  81  82  83  84  85  86  87  88
Interrupt signal (4) received.
Do you wish to continue (1 = yes or 2 = no)? 1
  89  90
  91  92  93  94  95  96  97  98  99  100
```

Fig. 14.8 Cómo utilizar el manejo de señales (parte 2 de 2).

técnicas de programación estructurada. En estos casos, pueden utilizarse algunas técnicas de programación no estructuradas. Por ejemplo, podemos utilizar **break** para terminar la ejecución de una estructura de repetición, antes de que se convierta en falsa la condición de continuación de ciclo. Esto ahorrará repeticiones innecesarias del ciclo, si antes de la terminación de éste la tarea ha sido terminada.

Otro ejemplo de programación no estructurada es el *enunciado* **goto** —una bifurcación incondicional. El resultado del enunciado **goto** es un cambio en el flujo de control del programa al primer enunciado que siga a la *etiqueta* especificada en el enunciado **goto**. Una etiqueta es un identificador seguido por dos puntos. Una etiqueta deberá aparecer en la misma función que el enunciado **goto** al cual se hace referencia. El programa de la figura 14.9 utiliza enunciados **goto** para ciclar diez veces e imprimir el valor de contador cada una de las veces. Después de inicializar **count** a **1**, el programa verifica **count** para determinar si es mayor que **10** (la etiqueta **start** es pasada por alto, porque las etiquetas no ejecutan ninguna acción). De ser así, el control se transfiere desde **goto** al primer enunciado después de la etiqueta **end**. De lo contrario, se imprime **count** y se incrementa, y el control se transfiere de **goto** al primer enunciado después de la etiqueta **start**.

En el capítulo 3 indicamos que para escribir cualquier programa sólo eran requeridas 3 estructuras de control: secuencia, selección y repetición. Cuando se siguen las reglas de la programación estructurada, es posible crear estructuras de control muy anidadas, de las cuales es difícil salir con eficacia. En situaciones como esas, algunos programadores utilizan enunciados **goto**, como una salida rápida de una estructura muy anidada. Esto elimina la necesidad de probar múltiples condiciones para salir de una estructura de control.

Sugerencia de rendimiento 14.3

El enunciado **goto** *puede ser utilizado para salir con eficacia de estructuras de control muy anidadas.*

Observación de ingeniería de software 14.3

El enunciado **goto** *debe ser utilizado sólo en aplicaciones orientadas a rendimiento. El enunciado* **goto** *no es estructurado y puede llevar a programas más difíciles de depurar, mantener y modificar.*


```
/* Using goto */
#include <stdio.h>

main()
{
   int count = 1;

   start:                          /* label */
      if (count > 10)
         goto end;

      printf("%d  ", count);
      ++count;
      goto start;

   end:                            /* label */
      putchar('\n');

   return 0;
}
```

```
   1  2  3  4  5  6  7  8  9  10
```

Fig. 14.9 Cómo utilizar **goto**.

Resumen

- En muchos sistemas de cómputo y en particular, en sistemas UNIX y DOS es posible redirigir la entrada a un programa y redirigir la salida de un programa.
- La entrada se redirige desde la línea de comandos de UNIX y de DOS utilizando el símbolo de redirección de entrada (<)o mediante una tubería (|).
- La salida es redirigida desde la línea de comandos de UNIX y de DOS utilizando el símbolo de redirección de salida (>) o el símbolo de agregar salida (>>). El símbolo de redirección de salida sólo almacena la salida de programa en un archivo, y el símbolo de agregar salida, agrega la salida al final de un archivo.
- Las macros y definiciones del encabezado de argumentos variables **stdarg.h** proporcionan las capacidades necesarias para construir funciones utilizando listas de argumentos de longitud variable.
- Unos puntos suspensivos (...) en una función prototipo indican un número variable de argumentos.
- El tipo **va_list** es adecuado para contener información necesaria para las macros **va_start**, **va_arg**, y **va_end**. Para tener acceso a los argumentos en una lista de argumentos de longitud variable, deberá ser declarado un objeto del tipo **va_list**.
- Antes de poder tener acceso a los argumentos de una lista de argumentos de longitud variable, se invoca la macro **va_start**. La macro inicializa el objeto declarado mediante **va_list**, para su uso con las macros **va_arg**, y **va_end**.

- La macro **va_arg** se expande a una expresión del valor y del tipo del siguiente argumento de la lista de argumentos de longitud variable. Cada invocación de **va_arg** modifica el objeto declarado en **va_list**, de tal forma que el objeto señale al siguiente argumento dentro de la lista.
- La macro **va_end** facilita un regreso normal de una función cuya lista de argumentos variables fue referida por la macro **va_start**.
- En muchos sistemas y en particular en DOS y en UNIX incluyendo los parámetros **int argc** y **char *argv[]** en la lista de parámetros de **main**, es posible pasar argumentos a **main** desde la línea de comandos. El parámetro **argc** recibe el número de argumentos de la línea de comandos. El parámetro **argv** es un arreglo de cadenas, en el cual se almacenan los argumentos actuales de la línea de comandos.
- La definición de una función debe estar en su totalidad contenida en un archivo, no puede estar distribuida en dos o más archivos.
- Las variables globales deberán ser declaradas en cada uno de los archivos en las que serán utilizadas.
- Los prototipos de función pueden extender el alcance de una función más allá del archivo en el cual ha sido definida (en un prototipo de función no es requerido el especificador **extern**). Esto se lleva a cabo incluyendo a el prototipo de función en cada archivo en el cual sea invocada la función y compilando juntos ambos archivos.
- El especificador de clase de almacenamiento **static**, al ser aplicado a la variable global o a una función, impide que ésta sea utilizada por cualquier función que no haya sido definida en el mismo archivo. Esto se conoce como enlace interno. Las variables globales y las funciones que en sus definiciones no hayan sido precedidas por **static** tienen enlace externo —se puede tener acceso a ellas en otros archivos, si estos archivos contienen las declaraciones y/o los prototipos de función adecuadas.
- El especificador **static** se usa por lo general con funciones de utilería, que son llamadas sólo por funciones en un archivo en particular. Si una función no es requerida fuera de un archivo particular, deberá ser puesto en aplicación el principio del mínimo privilegio mediante el uso de **static**.
- Al construir programas extensos en múltiples archivos fuente, la compilación de programas se hace tediosa si al efectuar pequeñas modificaciones a un archivo se tiene que recompilar todo el programa. Muchos sistemas proporcionan utilerías especiales que recompila sólo el archivo de programa modificado. En sistemas UNIX la utilería se llama **make**. La utilería **make** lee un archivo llamado **makefile** que tiene instrucciones para compilar y enlazar el programa.
- La función **exit** obliga a la terminación de un programa como si se hubiera ejecutado normalmente.
- La función **atexit** registra una función en un programa, para que sea llamada a la terminación del programa es decir, ya sea cuando el programa termina por llegar al final de **main** o cuando se invoca a **exit**.
- La función **atexit** toma como argumento un apuntador a una función (es decir, el nombre de la función). Las funciones llamadas a la terminación del programa no pueden tener argumentos, y no pueden regresar un valor. Pueden registrarse hasta 32 funciones para su ejecución a la terminación del programa.

- La función **exit** toma un argumento. Por lo regular el argumento es la constante simbólica **EXIT_SUCCESS** o la constante simbólica **EXIT_FAILURE**. Si **exit** se llama con **EXIT_SUCCESS**, es regresado el valor definido por la puesta en práctica correspondiente a la terminación exitosa al entorno llamador. Si **exit** es llamado mediante **EXIT_FAILURE** el valor definido por la puesta en práctica correspondiente a la terminación no exitosa, es regresado.
- Cuando se invoca la función **exit**, cualesquiera funciones registradas con **atexit** son invocadas en orden inverso a su registro, todos los flujos asociados con el programa son vaciados y cerrados, y el control regresa al entorno huésped.
- El estándar ANSI (An90) indica que cuando se utiliza **volatile** para calificar un tipo, la naturaleza del acceso a un objeto de dicho tipo depende de la puesta en práctica. Kernighan y Ritchie (Ke88) indican que el calificador **volatile** se utiliza para suprimir varios tipos de optimizaciones.
- C proporciona sufijos enteros y de punto flotante para especificar los tipos de constantes enteras y de punto flotante. Los sufijos enteros son: **u** o bien **U** por un entero **unsigned**, **l** o **L** por un entero **long**, y **ul** o **UL** por un entero **unsigned long**. Si una constante entera no tiene sufijo, su tipo queda determinado por el primer tipo capaz de almacenar un valor de dicho tamaño (primero **int**, luego **long int** y por último **unsigned long int**). Los sufijos de punto flotante son: **f** o **F** para un **float**, y **l** o **L** para un **long double**. Una constante de punto flotante sin sufijo es del tipo **double**.
- C también proporciona capacidades para procesamiento de archivos binarios, pero algunos sistemas de cómputo no aceptan archivos binarios. Si los archivos binarios no son aceptados, y un archivo se abre en el modo de archivo binario, el archivo será procesado como archivo de texto.
- La función **tempfile** abre un archivo temporal en modo **"wb+"**. A pesar de que éste es un modo de archivo binario, algunos sistemas procesan los archivos temporales como archivos de texto. Un archivo temporal existe en tanto no sea cerrado mediante **fclose** o en tanto termine el programa.
- La biblioteca de manejo de señales proporciona la capacidad de atrapar eventos inesperados mediante la función **signal**. La función **signal** recibe dos argumentos un número de señal entero y un apuntador a la función de manejo de señal.
- La señales también pueden ser generadas mediante la función **raise** y un argumento entero.
- La biblioteca general de utilerías (**stdlib.h**) tiene dos funciones para la asignación dinámica de memoria **calloc** y **realloc**. Estas funciones pueden ser utilizadas para crear arreglos dinámicos.
- La función **calloc** recibe dos argumentos el número de elementos (**nmemb**) y el tamaño de cada elemento (**size**), e inicializa a cero los elementos del arreglo. La función regresa ya sea un apuntador a la memoria asignada, o un apuntador **NULL** si la memoria no ha sido asignada.
- La función **realloc** modifica el tamaño de un objeto asignado por una llamada a **malloc**, **calloc** o **realloc** previa. El contenido del objeto original no es modificado, siempre y cuando la cantidad de memoria asignada sea mayor que la cantidad ya asignada.
- La función **realloc** toma dos argumentos un apuntador al objeto original (**ptr**) y el nuevo tamaño del objeto (**size**). Si **ptr** es **NULL**, **realloc** funciona en forma idéntica a **malloc**. Si **size** es **0** y el apuntador recibido no es **NULL**, la memoria para el objeto es liberada. De lo contrario si **ptr** no es **NULL** y el tamaño es mayor que 0, **realloc** intenta asignar un nuevo

contrario si `ptr` no es `NULL` y el tamaño es mayor que 0, `realloc` intenta asignar un nuevo bloque de memoria para el objeto. Si el nuevo espacio no puede ser asignado, el objeto al cual señala `ptr`, se conserva sin modificar. La función `realloc` regresa ya sea un apuntador a la memoria reasignada, o un apuntador `NULL`.

- El resultado del enunciado `goto` es una modificación en el flujo de control del programa. La ejecución del programa continúa en el primer enunciado inmediatamente después de la etiqueta especificada en el enunciado `goto`.
- Una etiqueta es un identificador seguido por dos puntos. Debe de aparecer una etiqueta en la misma función que en el enunciado `goto` que se refiere a ella.

Terminología

símbolo de agregar salida >>
`argc`
`argv`
`atexit`
`calloc`
argumentos en la línea de comandos
`const`
arreglos dinámicos
evento
`exit`
enlace externo
especificador de clase de almacenamiento `extern`
`EXIT_FAILURE`
`EXIT_SUCCESS`
sufijo `float` (`f` o `F`)
excepción de punto flotante
enunciado `goto`
redirección de E/S
instrucción ilegal
enlace interno
interrupción
sufijo `long double` (`l` o `L`)
sufijo `long integer` (`l` o `L`)
`make`
`makefile`
tubería |
entubamiento
`raise`
`realloc`
símbolo de redirección de entrada <
símbolo de redirección de salida >
violación de segmentación
`signal`
biblioteca de manejo de señales
`signal.h`
especificador de clase de almacenamiento `static`
`stdarg.h`
`archivo temporal`
`tmpfile`
atrapar
sufijo entero `unsigned` (`u` o bien `U`)
sufijo entero `unsigned long` (`ul` o bien `UL`)
`va_arg`
`va_end`
`va_list`
`va_start`
lista de argumentos de longitud variable
`volatile`

Error común de programación

14.1 Colocar puntos suspensivos en la mitad de una lista de parámetros de función. Los puntos suspensivos sólo pueden ser colocados al final de la lista de parámetros.

Sugerencias de portabilidad

14.1 Algunos sistemas no aceptan nombres de variables globales o nombres de funciones con más de 6 caracteres. Esto deberá ser tomado en cuenta al escribir programas que se planea serán transportados a varias plataformas.

14.2 Cuando esté escribiendo programas portátiles utilice archivos de texto.

Sugerencias de rendimiento

14.1 Las variables globales aumentan el rendimiento debido a que son de forma directa accesibles por cualquier función —se elimina la sobrecarga de pasar datos a función.

14.2 Considere utilizar archivos binarios en lugar de archivos de texto en aquellas aplicaciones que exijan alto rendimiento

14.3 El enunciado `goto` puede ser utilizado para salir con eficacia de estructuras de control muy anidadas.

Observaciones de ingeniería de software

14.1 Las variables globales deberían de evitarse, a menos de que el rendimiento de la aplicación sea crítico, porque violan el principio del mínimo privilegio.

14.2 Las funciones pueden ser comunes para muchas aplicaciones. En casos como éstos, estas funciones deberán ser almacenadas en sus propios archivos fuente, y cada archivo fuente deberá tener su correspondiente archivo de cabecera, que contenga los prototipos de función. Esto permite a los programadores de distintas aplicaciones reutilizar el mismo código, mediante la inclusión del archivo de cabecera apropiado, y compilar su aplicación con el archivo fuente correspondiente.

14.3 El enunciado `goto` debe ser utilizado sólo en aplicaciones orientadas a rendimiento. El enunciado `goto` no es estructurado y puede llevar a programas más difíciles de depurar, mantener y modificar.

Ejercicios de autoevaluación

14.1 Llene los espacios vacíos de cada uno de los siguientes:

a) El símbolo _____ se utiliza para redirigir datos de entrada del teclado para que provengan de un archivo.
b) El símbolo _____ se utiliza para redirigir la salida a pantalla para que se coloque en un archivo.
c) El símbolo _____ se utiliza para agregar la salida de un programa al final de un archivo.
d) La _____ se utiliza para redirigir la salida de un programa como entrada de otro programa.
e) Una _____ en la lista de parámetros de una función indica que la función puede recibir un número variable de argumentos.
f) La macro _____ debe de ser invocada antes de que se pueda tener acceso a los argumentos de una lista de argumentos de longitud variable.
g) Se utiliza la macro _____ para tener acceso a los argumentos individuales de una lista de argumentos de longitud variable.
h) La macro _____ facilita un regreso normal de una función cuya lista de argumentos variable fue referida por la macro `va_start`.
i) El argumento _____ de `main` recibe el número de argumentos en la línea de comandos.
j) El argumento _____ de `main` almacena argumentos en la línea de comandos como cadenas de caracteres.
k) La utilería de UNIX _____ lee un archivo llamado _____ que contiene instrucciones para compilar y enlazar un programa formado de múltiples archivos fuente. La utilería también recompila el archivo si desde la última vez que fue compilado éste ha sido modificado.
l) La función _____ obliga a un programa a terminar su ejecución.
m) La función _____ registra una función que deberá ser llamada a la terminación normal del programa.
n) El calificador de tipo _____ indica que un objeto no deberá ser modificado después de haberse inicializado.
o) Se puede agregar un _____ entero o de punto flotante a una constante entera o de punto flotante para especificar el tipo exacto de la misma.

p) La función _____ abre un archivo temporal que existirá en tanto no se cierre o la ejecución del programa se termine.
q) La función _____ puede ser utilizada para atrapar eventos inesperados.
r) La función _____ genera una señal desde dentro de un programa.
s) La función _____ asigna dinámicamente la memoria para un arreglo que inicializa los elementos a cero.
t) La función _____ modifica el tamaño de un bloque de memoria ya asignada dinámicamente.

Respuestas a los ejercicios de autoevaluación

14.1 a) redirige entrada (<). b) redirige salida (>). c) agrega salida (>>). d) tubería (|). e) puntos suspensivos (...). f) `va_start`. g) `va_arg`. h) `va_end`. i) `argc`. j) `argv`. k) `make, makefile`. l) `exit`. m) `atexit`. n) `const`. o) sufijo. p) `tempfile`. q) `signal`. r) `raise`. s) `calloc`. t) `realloc`.

Ejercicios

14.2 Escriba un programa que calcule el producto de una serie de enteros que son pasados a la función `product` utilizando una lista de argumentos de longitud variable. Pruebe su función con varias llamadas, cada una con un número diferente de argumentos.

14.3 Escriba un programa que imprima los argumentos en la línea de comandos del programa.

14.4 Escriba un programa que ordene un arreglo de enteros en orden ascendente o en orden descendente. El programa deberá utilizar argumentos en la línea de comandos, para pasar o el argumento `-a` para orden ascendente o el argumento `-d` para orden descendente. (Nota: este es el formato estándar en UNIX para pasar opciones a un programa.)

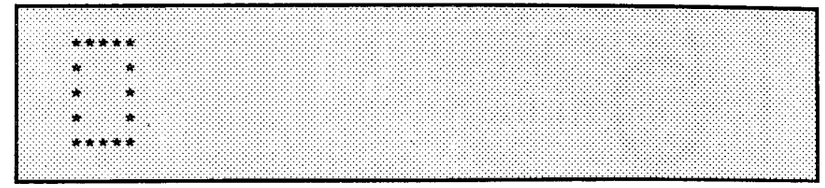
14.5 Escriba un programa que coloque un espacio entre cada carácter dentro de un archivo. El programa deberá primero escribir el contenido de un archivo a modificarse a un archivo temporal con espacios entre cada carácter, y a continuación copiar el archivo de regreso al archivo original. Esta operación deberá sobreescribir el contenido original del archivo.

14.6 Lea los manuales de su sistema para determinar qué señales contienen la biblioteca de manejo de señales (`signal.h`). Escriba un programa que contenga los manejadores de señales correspondientes a las señales estándar `SIGABRT` y `SIGINT`. El programa deberá probar el método de atrapar de estas señales mediante el llamado a las funciones `abort` para generar una señal del tipo `SIGABRT`, y mediante la escritura de `<ctrl> c` para generar una señal del tipo `SIGINT`.

14.7 Escriba un programa que asigne de forma dinámica un arreglo de enteros. El tamaño del arreglo deberá ser introducido desde el teclado. Los elementos del arreglo deberán ser valores asignados introducidos desde el teclado. Imprima los valores del arreglo. A continuación, reasigne la memoria para el arreglo a la mitad del número actual de elementos. Imprima los valores restantes en el arreglo para confirmar que corresponden a la primera mitad de los valores del arreglo original.

14.8 Escriba un programa que tome dos argumentos de la línea de comandos que son nombres de archivo, lea los caracteres del primer archivo un carácter a la vez, y escriba los caracteres en orden inverso en el segundo archivo.

14.9 Escriba un programa que utilice enunciados `goto` para simular una estructura de ciclado anidada, que imprima un cuadro de asteriscos como sigue:



El programa deberá utilizar sólo los siguientes tres enunciados `printf`:

```
printf("*");
printf(" ");
printf("\n");
```

15

C++ como un "C mejorado"

Objetivos

- Familiarizarse con las mejoras de C++ a C.
- Reconocer la razón porqué C es una base para subsecuentes estudios de la programación en general y de C ++ en particular.

Lo mejor es verte
con mi estimado.
El gran lobo feroz a la pequeña caperucita roja

Sinopsis

Resumen • Terminología • Errores comunes de programación • Prácticas sanas de programación • Sugerencia de portabilidad • Sugerencias de rendimiento • Observación de ingeniería de software • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios • Lectura recomendada • Apéndice: recursos de C++.

15.1 Introducción

¡Bienvenido a C++! C++ es una mejoría sobre muchas de las características de C, y proporciona capacidades de programación orientada a objetos *(OOP,* por *object-oriented programming)* que promete mucho para incrementar la productividad, calidad y reutilización del software. El resto de este capítulo analiza muchas de las mejoras que C++ tiene sobre C.

Los diseñadores de C y los responsables de sus primeras puestas en práctica jamás anticiparon que este lenguaje resultaría en un fenómeno como éste (lo mismo es cierto para el sistema operativo UNIX). Cuando un lenguaje de programación se torna tan arraigado como C, nuevas necesidades demandan que el lenguaje evolucione, en lugar de que sólo sea remplazado por un nuevo lenguaje.

C++ fue desarrollado por Bjarne Stroustrup en los Laboratorios Bell (St86) y originalmente fue llamado "C con clases". El nombre C++ incluye el operador de incremento (++) de C, para indicar que C++ es una versión mejorada de C.

C++ es un super conjunto de C, por lo que, para compilar los programas existentes de C, los programadores pueden utilizar un compilador C++ y posteriormente modificar de forma gradual estos programas a C++. En este momento, ya algunos proveedores importantes de software ofrecen compiladores C++ y no ofrecen productos C por separado.

Aún no existe un estándar ANSI, a pesar de que un comité ANSI está desarrollando una proposición. Muchas personas creen que para mediados de los años noventa, la mayor parte de los entornos de programación C se convertirán a C++.

15.2 Comentarios de una sola línea de C++

Con frecuencia los programadores insertan un pequeño comentario al final de una línea de código. C requiere que un comentario sea delimitado mediante `/*` y `*/`. C++ le permitirá a que empiece un comentario con `//` y que utilice el resto de la línea para texto del comentario; el fin de la línea da de manera automática por terminado el comentario. Esta forma de comentario ahorra algunos golpes de tecla, pero sólo es aplicable a comentarios que no continúen a la línea siguiente.

Por ejemplo, el comentario C

```
/* This is a single-line comment. */
```

requiere de ambos delimitadores `/*` y `*/`, aun para un comentario de una línea. La versión en una línea de C++ de lo anterior es

```
// This is a single-line comment.
```

Para comentarios de más de una línea, como

```
/* This is one way to   */
/* Write neat multiple- */
/* line comments.       */
```

La notación C++ pudiera aparecer más concisa como en

```
// This is one way to
// Write neat multiple-
// line comments.
```

Pero recuerde que los comentarios C pueden extenderse a varias líneas, por lo que en realidad, sólo se necesita un par (en vez de que de los tres pares que hemos utilizado) de delimitadores de comentario, como en

```
/* This is one way to
   Write neat multiple-
   line comments.       */
```

Dado que C++ es un super conjunto de C, en un programa C++ ambas formas de comentario son aceptables.

Error común de programación 15.1

*Olvidar cerrar un comentario de estilo C mediante */.*

Práctica sana de programación 15.1

*Usar los comentarios // de estilo C++, evita errores de comentarios sin terminar, que ocurren al olvidarse de cerrar los comentarios de estilo C con */.*

Este error en apariencia sencillo puede llevar a errores muy sutiles, algunos de los cuales pueden deslizarse sin detección por parte del compilador. El efecto de omitir el `*/` es que el compilador supone que el comentario aún no ha terminado, y que el comentario continúa a lo largo de las líneas subsiguientes hasta que alcanza otro `*/`, correspondiente a otro comentario, o el final del archivo del programa. Esto podría dar como resultado que el compilador ignore alguna parte clave del programa.

15.3 Flujo de entrada/salida de C++

C++ ofrece una alternativa a las llamadas de función **`printf`** y **`scanf`** para manejar la entrada/salida de los tipos y cadenas de datos estándar. Por ejemplo, el diálogo simple

```
printf("Enter new tag: ");
scanf("%d", &tag);
printf("The new tag is: %d\n", tag);
```

se escribe en C++ como

```
cout << "Enter new tag: ";
cin >> Tag;
cout << "The new tag is: " << tag << '\n';
```

El primer enunciado utiliza el *flujo estándar de salida* **`cout`** y el operador << (el *operador de inserción de flujo* que se pronuncia *"colocar en"*). El enunciado se lee

La cadena **`"Enter new tag"`** *es colocada en el flujo de salida* **`cout`**.

Note que el operador de inserción de flujo es también el operador de desplazamiento a la izquierda a nivel de bits. El segundo enunciado utiliza el *flujo de entrada estándar* **`cin`** y el operador >> (el *operador de extracción de flujo,* que se pronuncia *"obtener de"*). El enunciado se lee

Obtener un valor para tag del flujo de entrada **`cin`**.

Note que el operador de extracción de flujo también es un operador de desplazamiento a la derecha a nivel de bits cuando el argumento izquierdo es un tipo entero. Los operadores de inserción y de extracción de flujo, a diferencia de **`printf`** y de **`scanf`**, no requieren de cadenas de formato y de especificadores de conversión para indicar los tipos de datos que son extraídos o introducidos. C++ tiene muchos ejemplos como éste, en los cuales de forma automática "sabe" qué tipos utilizar. También note que, cuando se utiliza con el operador de extracción de flujo, la variable **`tag`** no es precedida por el operador de dirección **`&`**, como es requerido en el caso de **`scanf`**.

Para utilizar entradas/salidas de flujo, los programas C++ deben incluir el archivo de cabecera ***`iostream.h`***. El programa de la figura 15.1 solicita las variables **`myAge`** y **`friendsAge`** y determina si usted es de más edad, menos edad o de la misma que su amigo. Note la colocación de las declaraciones de edad, justo antes de la referenciación de las edades en los enunciados de entrada. En el capítulo 19, "Flujo de entrada/salida" se da una descripción detallada de las características del flujo de entrada/salida de C++.

Práctica sana de programación 15.2

Utilizar entrada/salida orientada a flujo de tipo C++ hace los programas más legibles (y menos sujetos a errores), que sus contrapartidas escritas en C mediante las llamadas de función **`printf`** *y* **`scanf`**.


```
// Simple stream input/output
#include <iostream.h>

main()
{
   cout << "Enter your age: ";
   int myAge;
   cin >> myAge;

   cout << "Enter your friend's age: ";
   int friendsAge;
   cin >> friendsAge;

   if (myAge > friendsAge)
      cout << "You are older.\n";
   else
      if (myAge < friendsAge)
         cout << "You are younger.\n";
      else
         cout << "You and your friend are the same age.\n";

   return 0;
}
```

```
Enter your age: 23
Enter your friend's age: 20
You are older.
```

```
Enter your age: 20
Enter your friend's age: 23
You are younger.
```

```
Enter your age: 20
Enter your friend's age: 20
You and your friend are the same age.
```

Fig. 15.1 Flujo de E/S y los operadores de inserción y extracción de flujo.

15.4 Declaraciones en C++

En un bloque en C, todas las declaraciones deben aparecer antes de cualquier enunciado ejecutable. En C++, las declaraciones pueden ser colocadas en cualquier parte de un enunciado ejecutable, siempre y cuando las declaraciones antecedan el uso de lo que se está declarando. Por ejemplo

```
cout << "Enter two integers: ";
int x, y;
cin >> x >> y;
cout << "The sum of " << x << " and " << y
     << " is " << x + y << '\n';
```

declara las variables **x** y **y** después del enunciado ejecutable **cout**, pero antes que sean utilizadas en el enunciado subsecuente **cin**. También, las variables pueden ser declaradas en la sección de inicialización de una estructura **for** dichas variables se mantienen en alcance hasta el final del bloque en el cual la estructura **for** está definida. Por ejemplo,

```
for (int i = 0; i <= 5; i ++)
   count << i <<  '\n';
```

declara la variable **i** ser un entero y la inicializa a **0** en la estructura **for**.

El alcance de una variable local C++ empieza en su declaración y se extiende hasta la llave derecha de cierre (**}**). Por lo tanto, los enunciados anteriores a la declaración variable no pueden referirse a la variable, aun si éstos están en el mismo bloque. Las declaraciones de variable no pueden ser colocadas en la condición de una estructura **while, do/while, for**, o **if.**

Práctica sana de programación 15.3

Colocar la declaración de una variable cerca de su primer uso puede hacer los programas más legibles.

Error común de programación 15.2

Declarar una variable después de que haya sido referenciada en un enunciado.

15.5 Cómo crear nuevos tipos de datos en C++

C++ proporciona la capacidad de crear tipos definidos por el usuario mediante el uso de la palabra reservada **enum**, la palabra reservada **struct**, la palabra reservada **union** y la nueva palabra reservada **class**. Al igual que en C, las enumeraciones C++ son declaradas mediante **enum**. Sin embargo; a diferencia de C, una enumeración en C++, cuando se declara, se convierte en un tipo nuevo. Para declarar de variable del nuevo tipo la palabra reservada **enum** no es requerida. Lo mismo es cierto en el caso de **struct, union** y **class**. Por ejemplo, las declaraciones

```
enum Boolean {FALSE, TRUE};

struct Name {
   char first[10];
   char last[10];
};
union Number {
   int i;
   float f;
};
```

crean tres tipos de datos, definidos por usuario, con los nombres de etiqueta **Boolean, Name** y **Number**. Los nombres de etiqueta pueden ser utilizados para declarar variables, como sigue:

```
Boolean done = FALSE;
Name student;
Number x;
```


Estas declaraciones crean la variable **Boolean done** (inicializada a **FALSE**), la variable **Name student** y la variable **Number x.**

Como en C, las enumeraciones por omisión son valuadas iniciándose en cero y cada elemento subsecuente se incrementa en uno. Por lo tanto, la enumeración **Boolean** asigna el valor **0** a **FALSE** y **1** a **TRUE**. Cualquier elemento en una enumeración puede ser asignado un valor entero. Los elementos subsecuentes, que no sean asignados a un valor, recibirán de manera automática el valor del elemento anterior, incrementado en uno.

15.6 Prototipos de función y verificación de tipo

El prototipo de función le permiten al compilador de C verificar por tipo la exactitud de las llamadas de función. En ANSI C, los prototipos de función son opcionales. En C++ los prototipos de función son requeridas para todas las funciones. Una función definida en un archivo, antes de cualquier llamada a la misma, no requiere de un prototipo de función por separado. En este caso, el encabezado de función actúa como prototipo de función. C++ también requiere que se declaren todos los parámetros de función en los paréntesis de la definición de función y del prototipo. Por ejemplo, una función **square**, que toma un entero de argumento y regresa un entero tiene el prototipo:

```
int square(int);
```

Las funciones que no regresan un valor se declaran con el tipo de regreso **void.**

Error común de programación 15.3

Un intento de regresar un valor de una función **void** *o de utilizar el resultado de una llamada a una función* **void.**

Para especificar en C una lista vacía de parámetros, entre paréntesis se coloca la palabra reservada **void**. Si no se coloca nada en los paréntesis de un prototipo de función de C, para dicha función se desactiva toda verificación de argumentos y nada se supone en relación con el número de argumentos y los tipos de argumentos a la función. Cualquier llamada de función correspondiente a esta función puede pasar cualesquiera argumentos que desee, sin que el compilador reporte error alguno.

En C++, una lista vacía de parámetros se especifica escribiendo **void** o absolutamente nada en los paréntesis. La declaración

```
void print();
```

especifica que la función **print** no toma argumentos y no regresa un valor. En la figura 15.2 se muestran ambas formas de declarar en C++ y de usar las funciones que no toman argumentos.

Sugerencia de portabilidad 15.1

El significado de una lista de función de parámetros vacía es muy distinto en C++ que en C. En C, significa que se deshabilita toda verificación de argumentos. En C++, significa que la función no toma argumentos. Por lo tanto, si se compilan en C++, los programas en C que utilizan esta característica pudieran ejecutarse en forma diferente.

Error común de programación 15.4

Los programas en C++ no se compilarán a menos de que sean proporcionadas los prototipos de función para cada una de las funciones o que cada función quede definida antes de ser utilizada.

```
// Functions that take no arguments
#include <iostream.h>

void f1();
void f2(void);

main()
{
   f1();
   f2();

   return 0;
}

void f1()
{
   cout << "Function f1 takes no arguments\n";
}

void f2(void)
{
   cout << "Function f2 also takes no arguments\n";
}
```

```
Function f1 takes no arguments
Function f2 also takes no arguments
```

Fig. 15.2 Dos maneras de declarar y utilizar funciones que no toman argumentos.

15.7 Funciones en línea

Desde el punto de vista de la ingeniería del software es una buena idea poner en práctica un programa como un conjunto de funciones, pero las llamadas de función involucran sobrecarga en tiempo de ejecución. C++ tiene *funciones en línea* que ayudan a reducir la sobrecarga por llamadas de función especial para pequeñas funciones. El calificador **inline** colocado en la definición de función antes del tipo de regreso de una función "aconseja" al compilador que genere una copia del código de la función "in situ" (cuando sea apropiado), a fin de evitar una llamada de función. La contrapartida es que en el programa se insertan muchas copias de código de la función, en vez de, cada vez que se llama a la función, tener una copia de la función a la cual pasarle el control. El compilador puede ignorar el calificador **inline** y típicamente así lo hará para todas, a excepción de las funciones más pequeñas.

Observación de ingeniería del software 15.1

Cualquier modificación a una función `inline` *requiere que sean recompilados todos los clientes de dicha función. Esto pudiera resultar de importancia en algunas situaciones de desarrollo y mantenimiento de programas.*

Las funciones en línea ofrecen ventajas en comparación con las macros de preprocesador (vea el capítulo 13) que expanden código en línea. Una ventaja es que las funciones **inline** son iguales a cualquier otra función de C++. Por lo tanto, en llamadas a las funciones **inline** se ejecutará

una adecuada verificación de tipo; las macros de preprocesador no aceptan verificación de tipo. Otra ventaja es que las funciones **inline** eliminan los efectos colaterales inesperados, asociados con un uso inapropiado de las macros de preprocesador. Por último, las funciones **inline** pueden ser depuradas mediante un programa depurador. El depurador no reconoce a las macros de preprocesador como unidades especiales, porque sólo son sustituciones de texto, efectuadas por el preprocesador antes de la compilación del programa. Un depurador puede ayudar a localizar errores lógicos resultado de sustituciones de macros, pero no puede atribuir dichos errores a macros específicas.

Práctica sana de programación 15.4

El calificador `inline` *deberá ser utilizado sólo tratándose de funciones pequeñas, de uso frecuente.*

Sugerencia de rendimiento 15.1

Usar funciones `inline` *puede reducir tiempo de ejecución, pero puede aumentar el tamaño del programa.*

El programa de la figura 15.3 utiliza la función **inline** llamada **cube** para calcular el volumen de un cubo del lado **s**. La palabra reservada **const** en la lista de parámetros de la función cube, le indica al compilador que la función no modifica a la variable **s**. En la sección 15.9 se analiza con mayor detalle la palabra reservada **const**.

Las macros de preprocesador son operaciones definidas en una directiva de preprocesador **#define**. Una macro está compuesta de un nombre (identificador de la macro) y texto de remplazo. Las macros pueden ser definidas sin o con lista de argumentos. Las macros sin argumentos son procesadas como si fueran constantes simbólicas dentro del programa —el texto de remplazo sustituye el identificador de la macro. Las macros con argumentos sustituyen los

```
// Using an inline function to calculate
// the volume of a cube
#include <iostream.h>

inline float cube(const float s) { return s * s * s; }

main()
{
   cout << "Enter the side length of your cube:  ";

   float side;

   cin >> side;
   cout << "Volume of cube with side "
        << side << " is " << cube(side) << '\n';

   return 0;
}
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

Fig. 15.3 Cómo utilizar una función en línea para calcular el volumen de un cubo.

argumentos dentro del texto de remplazo y a continuación, expanden la macro dentro del programa.

Considere la siguiente macro de un argumento, para calcular la superficie de un cuadrado:

```
#define VALIDSQUARE(x)     (x) * (x)
```

El preprocesador localiza dentro del programa cada ocurrencia de **VALIDSQUARE(x)**, sustituye **x** (ya sea un valor o una expresión) en el texto de remplazo, y expande la macro dentro del programa. Por ejemplo,

```
cout < VALIDSQUARE(7);
```

se expande a

```
cout < (7) * (7);
```

Los paréntesis en el texto de remplazo obligan a un orden correcto de evaluación de la operación de la macro. Por ejemplo,

```
cout << VALIDSQUARE(2 + 3);
```

se expande a

```
cout << (2 + 3) * (2 + 3);
```

que se evalúa correctamente como `5 * 5 = 25`. El preprocesador no evalúa expresiones incluidas como argumentos de una macro; sólo remplaza cada instancia del nombre del argumento en el texto de remplazo, con una copia de la totalidad de la expresión. Por lo tanto, los paréntesis son necesarios, para asegurar que los argumentos son correctamente evaluados.

Considere una macro correspondiente, pero sin paréntesis en el texto de remplazo:

```
#define INVALIDSQUARE(x)   x * x
```

Si damos 2 + 3 como el argumento, la macro se expande como sigue:

```
2 + 3 * 2 +3
```

Esta expresión queda evaluada de forma incorrecta como `2 + 6 + 3 = 11` porque la multiplicación tiene una precedencia más alta que la suma.

C++ tiene las funciones en línea para eliminar los problemas asociados con las macros, para eliminar la sobrecarga asociada con las llamadas de función, y para asegurar la verificación de argumentos para las funciones. Los argumentos de las funciones en línea se evalúan antes de ser "pasados" a la función. Por lo tanto los paréntesis encerrando cada instancia de cada argumento en el cuerpo de una función en línea no son necesarios.

La función en línea

```
inline int square(int x) { return x * x; }
```

calcula el cuadrado de su argumento entero **x**. La llamada **square(2 + 3)** evalúa el argumento `2 + 3 = 5` y substituye este valor dentro del cuerpo de la función. El programa de la figura 15.4 demuestra las macros **VALIDSQUARE** y **INVALIDSQUARE**, así como la función en línea **square**.


```
// Examples of valid macros, invalid macros
// and inline functions
#include <iostream.h>

#define VALIDSQUARE(x)     (x) * (x)
#define INVALIDSQUARE(x)   x * x

inline int square(int x) { return x * x; }

main()
{
   cout << "  VALIDSQUARE(2 + 3) = "
        << VALIDSQUARE(2 + 3)
        << "\nINVALIDSQUARE(2 + 3) = "
        << INVALIDSQUARE(2 + 3)
        << "\n        square(2 + 3) = "
        << square(2 + 3) << '\n';

   return 0;
}
```

```
  VALIDSQUARE(2 + 3) = 25
INVALIDSQUARE(2 + 3) = 11
        square(2 + 3) = 25
```

Fig. 15.4 Macros de preprocesador y funciones en línea.

La figura 15.5 contiene una lista completa de las palabras reservadas de C++. La figura muestra las palabras reservadas comunes a C y a C++, y a continuación resalta las palabras reservadas, únicas en C++. Cada una de las palabras reservadas nuevas de C++ es explicada con detalle más adelante en el libro.

Error común de programación 15.5

C++ es un lenguaje en evolución y algunas de sus características pudieran no estar disponibles en su computadora. Usar características no puestas en práctica causará errores de sintaxis.

15.8 Parámetros por referencia

En C, todas las llamadas de función son llamadas por valor. En C las llamadas por referencia son simuladas pasando un apuntador a un objeto y obteniendo a continuación acceso al objeto desreferenciando el apuntador en la función llamada. Recuerde que en C los nombres de arreglos son ya apuntadores (constantes), por lo que los arreglos son automáticamente pasados en llamada simulada por referencia. Otros lenguaje de programación ofrecen formas directas de llamada por referencia como los parámetros **var** en Pascal. C++ corrige esta debilidad de C al ofrecer *parámetros de referencia.*

Un parámetro de referencia es un seudónimo de su argumento correspondiente. Para indicar que un parámetro de función es pasado por referencia, sólo coloque ampersand (**&**) después del tipo del parámetro en el prototipo de función (exactamente de la misma forma en que pondría un * después del tipo parámetro, para indicar que un parámetro es el apuntador a una variable). Utilice

Palabras reservadas en C++

C y C++

`auto`	`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`	`else`
`enum`	`extern`	`float`	`for`	`goto`
`if`	`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`	`struct`
`switch`	`typedef`	`union`	`unsigned`	`void`
`volatile`	`while`			

C++ únicamente

`asm`	Medio definido por la puesta en práctica de utilización de lenguaje de ensamble a lo largo de C++ (vea los manuales correspondientes a su sistema).
`catch`	Maneja una excepción generada por un `throw`.
`class`	Define una nueva clase. Pueden crearse objetos de esta clase.
`delete`	Destruye un objeto de memoria creado con `new`.
`friend`	Declara una función o una clase que sea un "friend (amigo)" de otra clase. Los amigos pueden tener acceso a todos los miembros de datos y a todas las funciones miembro de una clase.
`inline`	Avisa al compilador que una función particular deberá ser generada en línea, en vez de requerir de una llamada de función.
`new`	Asigna dinámicamente un objeto de memoria "en la tienda libre" —memoria adicional disponible para el programa en tiempo de ejecución. Determina automáticamente el tamaño del objeto.
`operator`	Declara un operador " homónimo".
`private`	Un miembro de clase accesible a funciones miembro y a funciones `friend` de la clase de miembros `private`.
`protected`	Una forma extendida de acceso `private`; también se puede tener acceso a los miembros `protected` por funciones miembro de clases derivadas y amigos de clases derivadas.
`public`	Un miembro de clase accesible a cualquier función.
`template`	Declare como construir una clase o una función, usando una variedad de tipos.
`this`	Un apuntador declarado en forma implícita en toda función de miembro no `static` de una clase. Señala al objeto al cual esta función miembro ha sido invocada.
`throw`	Transfiere control a un manejador de excepción o termina la ejecución del programa si no puede ser localizado un manejador apropiado.
`try`	Crea un bloque que contiene un conjunto de números que pudieran generar excepciones, y habilita el manejo de excepciones para cualquier excepción generada.
`virtual`	Declara una función virtual.

Fig. 15.5 Las palabras reservadas en C++.

la misma regla convencional en el encabezado de función al enlistar el tipo de parámetros. Por ejemplo, la declaración

```
int &count
```

en un encabezado de función puede ser leído "`count` es una referencia a `int`". En la llamada de función, sólo mencione la variable por su nombre y automáticamente será pasada por referencia. Entonces, el mencionar en el cuerpo de la función la variable por su nombre local, de hecho la refiere a la variable original en la función llamadora, y la variable original puede ser modificada de manera directa por la función llamada.

La figura 15.6 compara la llamada por valor, la llamada por referencia mediante apuntadores, y la llamada por referencia mediante parámetros de referencia. Los argumentos en las llamadas a `squareByValue` y `squareByReference` son idénticos. Es imposible saber si cualquiera de estas funciones modifica sus argumentos sin verificar los prototipos de función o las definiciones de función. Por esta razón, algunos programadores de C++ prefieren que los argumentos modificables sean pasados a las funciones utilizando apuntadores, y los argumentos no modificables sean pasados a las funciones utilizando referencias a constantes. Las referencias a constantes proporcionan la eficiencia del paso utilizando apuntadores y evitan la modificación de los argumentos del programa llamador. Para especificar una referencia a una constante, coloque el calificador `const` en la declaración de parámetros antes del especificador de tipo (en la sección 15.9 se analiza `const` en detalle). Note la colocación de `*` y de `&` en las listas de parámetros de ambas funciones. Algunos programadores de C++ prefieren escribir `int *bPtr` en vez de `int* bPtr`, e `int& cRef` en vez de `int &cRef`.

Práctica sana de programación 15.5

Utilice apuntadores para pasar argumentos que pudieran ser modificados por la función llamada, y utilice referencias a constantes para pasar argumentos extensos, que no serán modificados.

Error común de programación 15.6

Dado que en el cuerpo de la función llamada los parámetros de referencia se mencionan sólo por nombre, el programador pudiera de forma inadvertida tratar a los parámetros de referencia como parámetros en llamada por valor. Esto puede causar efectos colaterales inesperados, si las copias originales de las variables son modificadas por la función llamadora.

Sugerencia de rendimiento 15.2

En el caso de objetos extensos, utilice un parámetro de referencia `const` para simular la apariencia y la seguridad de una llamada por valor, pero evitando la sobrecarga de pasar una copia de dicho objeto extenso.

La referencia también puede ser utilizada como un seudónimo para otras variables dentro de una función. Por ejemplo, el código

```
int count = 1;      // declare integer variable count
int &c = count;     // create c as an alias for count

++c;                // increment count (using its alias)
```

incrementa la variable `count` utilizando su seudónimo `c`. Las variables de referencia deben ser inicializadas en sus declaraciones (véase las figuras 15.7 y 15.8), y no pueden ser reasignadas como seudónimos a otras variables. Una vez declarada una referencia como un seudónimo para otra variable, todas las operaciones que supuestamente se ejecuten sobre el seudónimo (es decir, sobre la referencia) de hecho se ejecutan sobre la variable original misma. El seudónimo es sólo

```
// Comparing call by value, call by reference with
// pointers, and call by reference with references
#include <iostream.h>

int squareByValue(int);
void squareByPointer(int *);
void squareByReference(int &);

main()
{
   int x = 2, y = 3, z = 4;

   cout << "x = " << x << " before squareByValue\n"
        << "Value returned by squareByValue: "
        << squareByValue(x)
        << "\nx = " << x << " after squareByValue\n\n";

   cout << "y = " << y << " before squareByPointer\n";
   squareByPointer(&y);
   cout << "y = " << y << " after squareByPointer\n\n";

   cout << "z = " << z << " before squareByReference\n";
   squareByReference(z);
   cout << "z = " << z << " after squareByReference\n";

   return 0;
}

int squareByValue(int a)
{
   return a *= a;   // caller's argument not modified
}

void squareByPointer(int *bPtr)
{
   *bPtr *= *bPtr;  // caller's argument modified
}

void squareByReference(int &cRef)
{
   cRef *= cRef;    // caller's argument modified
}
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

y = 3 before squareByPointer
y = 9 after squareByPointer

z = 4 before squareByReference
z = 16 after squareByReference
```

Fig. 15.6 Un ejemplo de llamada por referencia.

otro nombre de la variable original para el seudónimo —no se reserva espacio. Las referencias no pueden ser desreferenciadas mediante el apuntador de operador de indirección (*). Las referencias no pueden ser utilizadas para ejecutar "aritmética de referencia" (como cuando utilizamos apuntadores en aritmética de apuntadores para señalar a otros elementos de un arreglo). Otras operaciones inválidas sobre referencias incluyen señalar a referencias, tomar las direcciones de referencias, y comparar referencias (de hecho cada una de estas operaciones no generará un error de sintaxis; en vez de ello cada una de estas operaciones se efectuará en la variable para la cual la referencia es un seudónimo).

Las funciones pueden regresar apuntadores o referencias, pero esto puede resultar peligroso. Cuando se regresa un apuntador o una referencia a una variable declarada en la función llamada, la variable deberá ser declarada **`static`** dentro de dicha función. De lo contrario, el apuntador o referencia se refiere a una variable automática que al terminar la función se descarta; dicha variable se dice que está "indefinida" y el comportamiento de programa sería impredecible.

Error común de programación 15.7

No inicializar una variable de referencia al declararse.

Error común de programación 15.8

Intentar reasignar una referencia previa declarada como seudónimo a otra variable.

Error común de programación 15.9

Intentar desreferenciar una variable de referencia mediante un operador de indirección de apuntadores (recuerde que una referencia es un seudónimo correspondiente a una variable, y no un apuntador a una variable).

```
// References must be initialized
#include <iostream.h>

main()
{
   int x = 3, &y;   // Error: y must be initialized

   cout << "x = " << x << '\n'
        << "y = " << y << '\n';
   y = 7;
   cout << "x = " << x << '\n'
        << "y = " << y << '\n';

   return 0;
}
```

```
Compiling FIG15_7.CPP:
Error FIG15_7.CPP 6: Reference variable 'y' must be
   initialized
```

Fig. 15.7 Intento de uso de una referencia no inicializada.

```
// References must be initialized
#include <iostream.h>

main()
{
   int x = 3, &y = x;  // y is now an alias for x

   cout << "x = " << x << '\n'
        << "y = " << y << '\n';
   y = 7;
   cout << "x = " << x << '\n'
        << "y = " << y << '\n';

   return 0;
}
```

```
x = 3
y = 3
x = 7
y = 7
```

Fig. 15.8 Cómo utilizar una referencia inicializada.

Error común de programación 15.10

Regresar un apuntador o una referencia a una variable automática en una función llamada.

15.9 El calificador Const

En la sección 15.7 se ilustró la utilización del calificador **const** en la lista de parámetros de una función, para especificar que un argumento pasado a la función no es modificable en dicha función. El calificador **const** también puede ser utilizado para declarar las supuesta llamadas *"variables constantes"* (en vez de declarar constantes simbólicas en el preprocesador mediante **#define**) como en

```
const float PI = 3.14159;
```

Esta declaración genera la "variable constante" **PI** y la inicializa a **3.14159**. La variable al declararse debe ser inicializada con una expresión constante, y después no puede ser modificada (figura 15.9 y figura 15.10). Las "variables constantes" también se conocen como *constantes nombradas* o *variables de sólo lectura*. Note que el término "variable constante" es un oxímoron (contradicción), es decir, una contradicción en términos similares a "camarón jumbo" o "quemadura de congelador". (¡Por favor envíenos sus oximorones favoritos vía nuestra dirección de correo electrónico incluida en el prefacio. Gracias!).

Las variables constantes pueden ser colocadas en cualquier parte en que se espere una expresión constante. Por ejemplo, la siguiente declaración de arreglo utiliza la variable **const** para especificar el tamaño del arreglo:

```
const int arraySize = 100;
int array[arraySize];
```



```
// A const object must be initialized
main()
{
   const int x;  // Error: x must be initialized

   x = 7;        // Error: cannot modify a const variable

   return 0;
}
```

```
Compiling FIG15_9.CPP:
Error FIG15_9.CPP 4: Constant variable 'x' must be
   initialized
Error FIG15_9.CPP 6: Cannot modify a const object
```

Fig. 15.9 Un objeto **const** debe ser inicializado.

```
// Using a properly initialized constant variable
#include <iostream.h>

main()
{
   const int x = 7;  // initialized constant variable

   cout << "The value of constant variable x is: "
        << x << '\n';

   return 0;
}
```

```
The value of constant variable x is: 7
```

Fig. 15.10 Cómo inicializar correctamente y cómo utilizar una variable constante.

Las variables constantes también pueden ser colocadas en archivos de cabecera.

Error común de programación 15.11

*Usar variables **const** en declaraciones de arreglos y colocar variables **const** en archivos de cabecera (que están incluidos en varios archivos de fuente del mismo programa), ambos son ilegales en C pero legales en C++.*

Práctica sana de programación 15.6

*Una ventaja del uso de variables **const** en lugar de constantes simbólicas, es que las variables **const** son visibles con un depurador simbólico; las constantes simbólicas **#define** no lo son.*

Existen otros usos comunes para calificador **const**. Por ejemplo, puede ser declarado un apuntador constante:

```
int *const iPtr = &integer;
```

Esto declara a **iptr** como un apuntador constante a un entero. El valor al cual señala **iptr** puede ser modificado, pero **iptr** no puede ser asignado para señalar a otra posición diferente de memoria.

Un apuntador a un objeto constante puede ser declarado como sigue:

```
const int *iPtr = &integer;
```

Esto declara a **iptr** como un apuntador a una constante entera. El valor al cual **iptr** se refiere no puede ser modificado mediante **iptr**, pero **iptr** puede ser asignado para señalar a otra posición de memoria. Por lo tanto, **iptr** es un apuntador a través del cual se puede leer un valor, pero éste no puede ser modificado (en efecto, un apuntador de "sólo lectura"). El compilador protege los datos referenciados por apuntadores de sólo lectura, evitando que dichos apuntadores sean asignados a apuntadores que no sean de lectura solamente.

Error común de programación 15.12

Inicializar una variable **const** *con una expresión no* **const** *como sería una variable que no ha sido declarada* **const**.

Error común de programación 15.13

Intentar modificar una variable constante.

Error común de programación 15.14

Intentar modificar un apuntador constante.

15.10 Asignación dinámica de memoria mediante new y delete

Los operadores ***new*** y ***delete*** de C++ le permiten a los programas llevar a cabo la asignación dinámica de memoria. En ANSI C, la asignación dinámica de memoria por lo general se lleva a cabo con las funciones estándar de biblioteca **malloc** y **free**. Considere la declaración

```
typeName *ptr;
```

donde **typeName** es cualquier tipo (como **int**, **float**, **char**, etcétera). En ANSI C, el enunciado siguiente asigna en forma dinámica un objeto **typeName**, regresa un apuntador **void** al objeto, y asigna dicho apuntador a **ptr**:

```
ptr = malloc(sizeof(typeName));
```

En C la asignación dinámica de memoria requiere una llamada de función a **malloc** y una referencia explícita al operador **sizeof** (o una mención explícita al número necesario de bytes). La memoria se asigna sobre el *montón* (memoria adicional disponible para el programa en tiempo de ejecución). También, en puestas en práctica anteriores a ANSI C, el apuntador regresado por **malloc** debe ser explícitamente convertido (cast) al tipo apropiado de apuntador con el convertidor explícito (cast) (**typeName***).

En C++, el enunciado

```
ptr = new typeName
```

asigna memoria para un objeto del tipo **typeName** partiendo de la *tienda libre* del programa (término utilizado por C++ para la memoria adicional disponible para el programa en tiempo de ejecución). El operador **new** crea automáticamente un objeto del tamaño apropiado, y regresa un apuntador del tipo apropiado. Si mediante **new** no se puede asignar memoria, se regresa un apuntador nulo (para representar un apuntador nulo los programadores C++ utilizan el valor **0**, en vez de **NULL**).

Para liberar en C++ el espacio para este objeto se utiliza el siguiente enunciado:

```
delete ptr;
```

En C, se invoca la función **free** con el argumento **ptr**, a fin de desasignar memoria. El operador **delete** sólo puede ser utilizado para desasignar memoria ya asignada mediante el operador **new**. Aplicar **delete** a un apuntador previamente desasignado, puede llevar a errores inesperados durante la ejecución del programa. Aplicar **delete** a un apuntador nulo no tiene efecto en la ejecución del programa.

Error común de programación 15.15

Desasignar memoria mediante **delete** *no originalmente asignada mediante el operador* **new**.

C++ permite un *inicializador* para un objeto recién asignado. Por ejemplo,

```
float* thingPtr = new float (3.14159);
```

inicializa a **3.14159** un objeto **float** recién asignado.

También mediante **new** los arreglos pueden ser creados dinámicamente. El siguiente enunciado asigna dinámicamente un arreglo de un subíndice de 100 enteros y asigna el apuntador regresado por **new** al apuntador entero **arrayPtr**:

```
int *arrayPtr;
arrayPtr = new int [100]; // creates array dynamically
```

Para desasignar la memoria asignada dinámicamente para **arrayPtr** por **new**, utilice el enunciado

```
delete [] arrayPtr;
```

En el capítulo 16, analizaremos la asignación dinámica de memoria de objetos **class**. Veremos que los operadores **new** y **delete** ejecutan otras tareas (como llamar de forma automática a las funciones constructor y destructor de una **class**) lo que hace que el uso de **new** y de **delete** sea más poderoso y más apropiado que el uso de **malloc** y de **free**. Por ahora, asegúrese de utilizar corchetes con **delete** al desasignar arreglos de objetos.

Error común de programación 15.16

Usar **delete** *sin corchetes ([y]) al borrar arreglos de objetos dinámicamente asignados (esto resulta un problema sólo en el caso de arreglos que contengan elementos de tipos definidos por el usuario)*

15.11 Argumentos por omisión

Las llamadas de función pudieran por lo común pasar un valor particular de un argumento. El programador puede definir que dicho argumento es un *argumento por omisión*, y el programador puede introducir el valor por omisión de dicho argumento. Cuando en una llamada de función es omitido un argumento por omisión, el valor por omisión de dicho argumento es automáticamente pasado en la llamada.

Los argumentos por omisión deben ser los argumentos que aparecen más a la derecha (los últimos) en una lista de parámetros de función. Al llamar a una función con dos o más argumentos por omisión, si un argumento omitido no es el argumento más a la derecha en la lista de argumentos, todos los argumentos a la derecha de dicho argumento también deben de ser omitidos. Los argumentos por omisión deberán ser especificados junto con la primera ocurrencia del nombre de la función —típicamente en el prototipo en un archivo de cabecera. Los argumentos por omisión también pueden ser utilizados con funciones **inline**.

En la figura 15.11 se demuestra el uso de los argumentos por omisión para calcular el volumen de una caja. A los tres argumentos se les ha dado el valor por omisión de **1**. La primera llamada a la función **inline boxVolume** no especifica ningún argumento y, por lo tanto, utiliza tres valores por omisión. La segunda llamada pasa un argumento **length** y, por lo tanto, utiliza valores por omisión para los argumentos **width** y **height**. La tercera llamada pasa los argumentos para **length y width** y, por lo tanto, utiliza el valor por omisión para el argumento **width**. La última llamada pasa argumentos para **length**, **width** y **height** y, por lo tanto, no utiliza valores por omisión.

Práctica sana de programación 15.7

El uso de argumentos por omisión puede simplificar la escritura de las llamadas de función. Sin embargo, algunos programadores sienten que resulta más claro especificar explícitamente todos los argumentos.

Error común de programación 15.17

Especificar e intentar utilizar un argumento por omisión que no es el argumento más a la derecha (el último) (simultáneamente dar como por omisión todos los argumentos más a la derecha).

15.12 Operador de resolución de alcance unario

Es posible declarar variables locales y globales con un mismo nombre. En C, en tanto esté en alcance la variable local, todas las referencias a dicho nombre de variable pertenecerán a la variable local —dentro del alcance la variable local la variable global no estará visible. C++ dispone del *operador de resolución de alcance unario (::)* para tener acceso a una variable global cuando está en alcance una variable local con el mismo nombre. El operador de resolución de alcance unario no puede ser utilizado para tener acceso a una variable del mismo nombre en un bloque externo. Se puede tener acceso a una variable global directa, sin el operador de resolución de alcance unario, si el nombre de la variable global no es el mismo que el nombre de la variable local en alcance. En el capítulo 16, analizaremos el uso del *operador de resolución de alcance binario* con clases.

En la figura 15.12 se demuestra el operador de resolución de alcance unario con variables locales y globales del mismo nombre. A fin de enfatizar que las versiones local y global de la variable **value** son distintas, el programa declara una de las variables como **float** y la otra como **int.**


```
// Using default arguments
#include <iostream.h>

// Calculate the volume of a box
inline int boxVolume(int length = 1, int width = 1,
                     int height = 1)
   { return length * width * height; }

main()
{
   cout << "The default box volume is: "
        << boxVolume()
        << "\n\nThe volume of a box with length 10,\n"
        << "width 1 and height 1 is: "
        << boxVolume(10)
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 1 is: "
        << boxVolume(10, 5)
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 2 is: "
        << boxVolume(10, 5, 2)
        << '\n';

   return 0;
}
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Fig. 15.11 Cómo utilizar los argumentos por omisión.

Error común de programación 15.18

Intentar tener acceso a una variable no global en un bloque externo, utilizando un operador de resolución de alcance unario.

Práctica sana de programación 15.8

Evite usar dentro de un programa variables del mismo nombre para distintos fines. Aunque en varias circunstancias esto es permitido, puede resultar confuso.

15.13 Homonimia de funciones

En C, declarar dos funciones del mismo nombre en el mismo programa es un error de sintaxis. C++ permite que sean definidas varias funciones del mismo nombre, siempre que estos nombres

```
// Using the unary scope resolution operator
#include <iostream.h>

float value = 1.2345;

main()
{
   int value = 7;

   cout << "Local value = " << value
        << "\nGlobal value = " << ::value << '\n';

   return 0;
}
```

```
Local value = 7
Global value = 1.2345
```

Fig. 15.12 Cómo utilizar el operador de resolución de alcance unario.

de funciones indiquen diferentes conjuntos de parámetros (por lo menos en lo que se refiere a sus tipos). Esta capacidad se llama *homonimia de funciones*. Cuando se llama a una función homónima, el compilador C++ selecciona de forma automática la función correcta examinando el número, tipos y orden de los argumentos de la llamada. La homonimia de la función se utiliza por lo común para crear varias funciones del mismo nombre, que ejecutan tareas similares, sobre tipos de datos diferentes.

Práctica sana de programación 15.9

La homonimia de funciones que ejecutan tareas muy relacionadas, puede hacer que los programas sean más legibles y comprensibles.

La figura 15.13 utiliza la función homónima **square** para calcular el cuadrado de un **int** y el cuadrado de un **double**. En el capítulo 17 analizaremos cómo hacer la homonimia de operadores para definir cómo deberán de operar sobre objetos de tipos de datos definidos por el usuario. En la sección 15.15 se presentan las funciones plantilla para la ejecución de tareas idénticas sobre muchos distintos tipos de datos. En el capítulo 20 se analizan las funciones plantilla y las clases plantilla con detalle.

Se pueden distinguir las funciones homónimas mediante su *firma* —una combinación del nombre de la función y de sus tipos de parámetros. El compilador codifica cada identificador de función en forma especial (conocido a veces como *mutilación de nombre o decoración de nombre*), utilizando el número y el tipo de sus parámetros, a fin de habilitar un *enlace a prueba de tipo*. Un enlace a prueba de tipo asegura que se llama a la función homónima correcta, y que los argumentos concuerdan con los parámetros. El compilador detecta y reporta los errores de enlace. El programa de la figura 15.14 fue compilado por el compilador Borland C++. Los nombres de función cifrados, producidos en lenguaje de ensamble por Borland C++, aparecen en la ventana de salida. Cada nombre "decorado" empieza con un @, seguido por el nombre de la función. La lista de parámetros codificados empieza con **$q**. En la lista de parámetros correspondiente a la función **nothing2**, **z** representa a un **char**, **i** representa a un **int**, **pf**


```
// Using overloaded functions
#include <iostream.h>

int square(int x) { return x * x; }

double square(double y) { return y * y; }

main()
{
   cout << "The square of integer 7 is "
        << square(7)
        << "\nThe square of double 7.5 is "
        << square(7.5) << '\n';

   return 0;
}
```

```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

Fig. 15.13 Cómo utilizar funciones homónimas.

representa a un **float ***, y **pd** representa a un **double ***. En la lista de parámetros correspondiente a la función **nothing1**, **i** representa a un **int**, **f** representa a un **float**, **zc** representa a un **char**, y **pi** representa a un **int***. Las dos funciones **square** se reconocen por medio de sus listas de parámetros; una especifica **d** para **double** y la otra especifica **i** para **int**. Los tipos de regreso de las cuatro funciones no se especifican en los nombres codificados. Note que el codificado de los nombres de funciones es específico a cada compilador. Por lo tanto, una función compilada con Borland C++ pudiera no tener el mismo nombre "decorado" que en otros compiladores.

Las funciones homónimas pueden tener diferentes tipos de regreso, pero deben tener diferentes listas de parámetros.

Error común de programación 15. 19

Generará un error de sintaxis crear funciones homónimas con listas de parámetros idénticas pero distintos tipos de regreso.

Para distinguir entre funciones con un mismo nombre el compilador sólo utiliza las listas de parámetros. Las funciones homónimas no necesariamente tienen el mismo número de parámetros. Al hacer la homonimia de funciones utilizando parámetros por omisión, los programadores deberán tener cuidado, ya que ello podría causar ambigüedad.

Error común de programación 15. 20

Una función con argumentos por omisión omitidos pudiera ser llamada en forma idéntica a otra función homónima; esto resultará en un error de sintaxis.

Error común de programación 15. 21

*La homonimia ayuda a eliminar el uso de las macros **#define** de C, que ejecutan tareas sobre múltiples tipos de datos, y utiliza las características poderosas de verificación de tipo de C++, que al utilizar macros no están disponibles.*

```
// Name mangling
int square(int x) { return x * x; }

double square(double y) { return y * y; }

void nothing1(int a, float b, char c, int *d) {}

char *nothing2(char a, int b, float *c, double *d) { return 0; }

main()
{
   return 0;
}
```

```
public _main
public @nothing2$qzcipfpd
public @nothing1$qifzcpi
public @square$qd
public @square$qi
```

Fig. 15.14 Decoración de nombres para habilitar enlaces a prueba de tipo.

15.14 Especificaciones de enlace

Es posible, desde un programa C++, llamar funciones escritas y compiladas con un compilador C. Como se indicó en la sección 15.13, C++ cifra en especial los nombres de la función para enlaces a prueba de tipo. C, por su parte, no codifica sus nombres de función. Por lo tanto, cuando se haga algún intento para enlazar el código C con código C++, una función compilada en C no será reconocida, porque el código C++ espera un nombre de función especialmente codificado. C++ le permite al programador dar *especificaciones de enlace* para informar al compilador que una función fue compilada en un compilador C, y evitar que el nombre de la función sea codificado por dicho compilador C++. Las especificaciones de enlace son útiles cuando se han desarrollado extensas bibliotecas de funciones especializadas, y el usuario, o no tiene acceso al código fuente para recompilarlas en C++, o no tiene el tiempo para convertir las funciones de bibliotecas de C a C++.

Para informar al compilador que una o varias funciones han sido compiladas en C, escriba los prototipos de función como sigue:

```
extern "C" function prototype    // single function

extern "C"              // multiple functions
{
   function prototypes
}
```

Estas declaraciones le informan al compilador que las funciones especificadas no están compiladas en C++, por lo que no deberá hacerse sobre las funciones enlistadas en la especificación de enlace el codificado de nombres. Estas funciones entonces podrán ser correctamente enlazadas

con el programa. Los entornos C++ incluyen normalmente las bibliotecas estándar de C y para dichas funciones no se requiere que el programador utilice especificaciones de enlace.

15.15 Plantillas de función

Las funciones homónimas se utilizan por lo regular para ejecutar operaciones similares sobre distintos tipos de datos. Si para cada tipo las operaciones son idénticas, esto se puede llevar a cabo de manera más compacta mediante *plantillas de función*, una capacidad introducida en versiones recientes de C++. El programador escribe una definición de plantilla de función. Basado en los tipos de los argumentos proporcionados en las llamadas a esta función, C++ genera de forma automática funciones de código objeto por separado para manejar cada tipo de llamada en forma correcta. En C, esta tarea puede ser llevada a cabo mediante macros creados con **`#define`**. Sin embargo, las macros presentan la posibilidad de serios efectos colaterales y no permiten que el compilador ejecute verificación de tipo. Las plantillas de función proporcionan, como las macros, una solución compacta, pero permiten verificación completa de tipo.

Todas las definiciones de plantillas de función empiezan con la palabra reservada **`template`**, seguida por una lista de parámetros formales a la plantilla de función encerrados en corchetes angulares (< y >). Cada parámetro formal es precedido por la palabra reservada **`class`**. Los parámetros formales se utilizan como tipos incorporados, o como tipos definidos por el usuario, para definir los tipos de los argumentos a la función, para definir el tipo de regreso de la función, y para declarar variables dentro de la función. A continuación se coloca la definición de función y se define como cualquier otra función.

La siguiente plantilla de función también es utilizada en el programa completo de la figura 15.15.

```
template <CLASS T>
void printArray(T *array, const int count)
{
   for (int i = 0; i < count; i++)
      cout << array[i] << " ";
   cout << '\n';
}
```

Esta plantilla de función declara un parámetro formal único **`T`** como el tipo del arreglo a imprimirse mediante la función **`printArray`**. Cuando el compilador detecta la llamada a **`printArray`** en el código fuente del programa, se substituye el tipo del primer argumento de **`printArray`** por la **`T`** en toda la definición de la plantilla, y C++ genera una plantilla de función completa, para la impresión de un arreglo de un tipo de datos especificado. A continuación se compila la nueva función creada. En la figura 15.15 se ejemplifican tres funciones una espera un arreglo **`int`**, otra espera un arreglo **`float`** y otra un arreglo **`char`**. La ejemplificación del tipo **`int`** es:

```
void printArray(int *array, const int count)
{
   for (int i = 0; i  count; i++)
      cout << array[i] << " ";
   cout << '\n';
}
```

Cada parámetro formal en la definición de plantilla debe aparecer en la lista de parámetros de la función por lo menos una vez. El nombre de un parámetro formal sólo puede ser utilizado

una vez en la lista de parámetros formales de la definición de plantilla. Los nombres de los parámetros formales entre funciones de plantilla no es necesario que deban ser únicos.

En la figura 15.15 se ilustra el uso de la función de plantilla **printArray**.

Error común de programación 15.22

No colocar la palabra reservada **class** *antes de cualquier parámetro formal de una plantilla de función.*

Error común de programación 15.23

No usar en la firma de función todos los parámetros formales de una plantilla de función.

```
// Using template functions
#include <iostream.h>

template <class T>
void printArray(T *array, const int count)
{
   for (int i = 0; i < count; i++)
      cout << array[i] << " ";

   cout << '\n';
}

main()
{
   const int aCount = 5, bCount = 7, cCount = 6;
   int a[aCount] = {1, 2, 3, 4, 5};
   float b[bCount] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
   char c[cCount] = "HELLO";  // 6th position for null

   cout << "Array a contains:\n";
   printArray(a, aCount);  // integer version

   cout << "Array b contains:\n";
   printArray(b, bCount);  // float version

   cout << "Array c contains:\n";
   printArray(c, cCount);  // character version

   return 0;
}
```

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

Fig. 15.15 Cómo utilizar funciones plantilla.

Resumen

- C++ es un superconjunto de C, por lo que los programadores pueden utilizar un compilador C++ para compilar sus programas existentes en C, y de ahí modificar de forma gradual estos programas a C++.
- C requiere que un comentario quede delimitado por **/*** y ***/**. C++ permite que un comentario empiece con **//** y que termine al final de la línea.
- C++ permite que las declaraciones aparezcan en cualquier parte en que un enunciado ejecutable pueda aparecer. Las declaraciones variables pueden también aparecer en la sección de inicialización de una estructura **for**.
- C++ proporciona una alternativa a las funciones **printf** y **scanf** para manejar la entrada/salida de los tipos de datos estándar y de las cadenas. El flujo de salida **cout**, y el operador **<<** (operador de inserción de flujo, que se lee "colocar en") le permite la salida de los datos. El flujo de entradas **cin** y el operador **>>** (el operador de extracción de flujo, que se lee "obtener de") permite que se introduzcan datos.
- C++ le permite al programador que cree tipos de datos definidos por el usuario, mediante las palabras reservadas **enum**, **struct**, **union** y **class**. El nombre de etiqueta de la enumeración, estructura, unión o clase puede entonces ser utilizada para definir variables del nuevo tipo.
- Los programas no se compilan a menos de que un prototipo de función esté incluida para cada función o una función esté definida antes de su uso (en cuyo caso no es requerida un prototipo de función por separado).
- Una función que no regrese un valor se declara con el tipo de regreso **void**. Si se hace algún intento ya sea de que la función regrese un valor o de utilizar en la función llamadora el resultado de la invocación de la función, el compilador generará un error.
- En C++, una lista de parámetros vacía se especifica con paréntesis vacíos, o con **void** entre paréntesis. En C, una lista de parámetros vacía desactiva la verificación de argumentos para la función.
- Las funciones en línea eliminan la sobrecarga de llamadas de función. El programador utiliza la palabra reservada **inline** para avisarle al compilador que genere código de función en línea (siempre que sea posible) a fin de minimizar llamadas de función. El compilador puede decidir ignorar el consejo **inline**.
- C++ ofrece una forma directa de llamada por referencia mediante el uso de parámetros de referencia. Para indicar que un parámetro de función es pasado por referencia, haga seguir al tipo de parámetro en el prototipo de función por un **&**. En la llamada de función, mencione la variable por nombre y será pasada en llamada por referencia. En la función llamada, el mencionar la variable por su nombre local, de hecho se refiere a la variable original en la función llamadora. Por lo tanto, la variable original puede ser modificada directamente por la función llamada.
- Los parámetros de referencia también pueden ser creados para uso local, como seudónimo de otras variables dentro de una función. Las variables de referencia deben ser inicializadas en sus declaraciones, y no pueden ser reasignadas como seudónimos de otras variables. Una vez declarada una variable de referencia como un seudónimo de otra variable, todas las operaciones supuestamente ejecutadas sobre el seudónimo de hecho se ejecutan sobre la variable.

- El calificador **const** también puede crear "variables constantes". Una variable constante debe ser inicializada al declarar la variable con una expresión constante, y después no puede ser modificada. Las variables constantes a menudo se llaman constantes o variables de sólo lectura. Las variables constantes pueden ser colocadas en cualquier lugar en donde se espere una expresión constante. Otros usos comunes del calificador **const** incluye apuntadores constantes, apuntadores a constantes y referencias constantes.
- Los operadores **new** y **delete** en C++ ofrecen una mejor forma de llevar a cabo la asignación dinámica de memoria que con las llamadas de función **malloc** y **free** en C. El operador **new** toma como operando un tipo, crea automáticamente un objeto del tamaño correcto, y regresa un apuntador del tipo correcto. El operador **delete** toma un apuntador a un objeto asignado con **new** y libera la memoria.
- C++ permite al programador especificar los argumentos por omisión y sus valores por omisión. Si en una llamada a una función se omite un argumento por omisión, se utiliza el valor por omisión de dicho argumento. Los argumentos por omisión deben ser los argumentos más a la derecha (los últimos) en la lista de parámetros de una función. Los argumentos por omisión deberán ser especificados en la primera ocurrencia del nombre de función.
- El *operador de resolución de alcance unario (::)* le permite a un programa tener acceso a una variable global, cuando está en alcance una variable local con el mismo nombre.
- Es posible definir varias funciones con el mismo nombre, pero con distintos tipos de parámetros. Esto se llama homonimia de función. Cuando es llamada una función homónima, el compilador selecciona automáticamente la función correcta, mediante el examen del número y tipo de los argumentos en la llamada.
- Las funciones homónimas pueden tener diferentes valores de regreso, y deben de tener diferentes listas de parámetros. Dos funciones que sólo difieran en el tipo de regreso darán como resultado un error de compilación.
- En algunos casos, es necesario llamar funciones escritas y compiladas con un compilador de C (por ejemplo, las funciones de biblioteca especializadas de una empresa que no hayan sido convertidas a C++ y recompiladas). C++ procesa los nombres de sus funciones en forma distinta a como lo hace C. Por lo tanto cuando se intente enlazar código C con código C++, una función compilada en C no será reconocida. C++ dispone de especificaciones de enlace, para informar al compilador que una función fue compilada sobre un compilador C, y evitar que se procese el nombre de la función como si fuera una función C++.
- Las plantillas de función permiten la creación de funciones que ejecutan las mismas operaciones sobre distintos tipos de datos, pero la plantilla de función se define sólo una vez.

Terminología

sufijo ampersand (**&**)
asm
catch
cin (flujo de entrada)
const
variable constante
cout (flujo de salida)
class
argumentos de función por omisión
operador **delete**
objetos dinámicos
tienda libre
friend
homonimia de funciones
operador "obtener de" (**>>**)
inicializador
función miembro **inline**
<iostream.h>
biblioteca iostream
especificaciones de enlace
constante nombrada
operador **new**
palabra reservada **operator**
homonimia
operador "colocar en" (**<<**)
variable de lectura solamente
parámetro de referencia
tipos de referencia
signatura
comentario en una sola línea (**//**)
operador de extracción de flujo (**>>**)
operador de inserción de flujo (**<<**)
template
función plantilla
this
throw
try
enlace a prueba de tipo
operador de resolución de alcance unario (**::**)
virtual

Errores comunes de programación

15.1 Olvidar cerrar un comentario de estilo C mediante ***/**.
15.2 Declarar una variable después de que haya sido referenciada en un enunciado.
15.3 Un intento de regresar un valor de una función **void** o de utilizar el resultado de una llamada a una función **void**.
15.4 Los programas en C++ no se compilarán a menos de que sean proporcionadas los prototipos de función para cada una de las funciones, o que cada función quede definida antes de ser utilizada.
15.5 C++ es un lenguaje en evolución y algunas de sus características pudieran no estar disponibles en su computadora. Usar características no puestas en práctica causará errores de sintaxis.
15.6 Dado que en el cuerpo de la función llamada los parámetros de referencia se mencionan sólo por nombre, el programador pudiera pasar por inadvertido y tratar a los parámetros de referencia como parámetros en llamada por valor. Esto puede causar efectos colaterales inesperados, si las copias originales de las variables son modificadas por la función llamadora.
15.7 No inicializar una variable de referencia al declararse.
15.8 Intentar reasignar una referencia ya declarada como seudónimo a otra variable.
15.9 Intentar desreferenciar una variable de referencia mediante un operador de indirección de apuntadores (recuerde que una referencia es un seudónimo correspondiente a una variable, y no un apuntador a una variable).
15.10 Regresar un apuntador o una referencia a una variable automática en una función llamada.
15.11 Usar variables **const** en declaraciones de arreglos y colocar variables **const** en archivos de cabecera (que están incluidos en varios archivos de fuente del mismo programa), ambos son ilegales en C pero legales en C++.
15.12 Inicializar una variable **const** con una expresión no **const** como sería una variable que no ha sido declarada **const**.
15.13 Intentar modificar una variable constante.
15.14 Intentar modificar un apuntador constante.
15.15 Desasignar memoria mediante **delete** no originalmente asignada mediante el operador **new**.
15.16 Usar **delete** sin corchetes (**[** y **]**) al borrar arreglos de objetos dinámicamente asignados (esto resulta un problema sólo en el caso de arreglos que contengan elementos de tipos definidos por el usuario)
15.17 Especificar e intentar utilizar un argumento por omisión que no es el argumento más a la derecha (el último) (simultáneamente dar como por omisión todos los argumentos más a la derecha).
15.18 Intentar tener acceso a una variable no global en un bloque externo, utilizando un operador de resolución de alcance unario.
15.19 Generará un error de sintaxis crear funciones homónimas con listas de parámetros idénticas pero distintos tipos de regreso.
15.20 Una función con argumentos por omisión omitidos pudiera ser llamada en forma idéntica a otra función homónima; esto resultará en un error de sintaxis.

15.21 La homonimia ayuda a eliminar el uso de las macros `#define` de C, que ejecutan tareas sobre múltiples tipos de datos, y utiliza las características poderosas de verificación de tipo de C++, que al utilizar macros no están disponibles.

15.22 No colocar la palabra reservada `class` antes de cualquier parámetro formal de una plantilla de función.

15.23 No usar en la firma de función todos los parámetros formales de una plantilla de función.

Prácticas sanas de programación

15.1 Usar los comentarios `//` de estilo C++, evita errores de comentarios sin terminar, que ocurren al olvidarse de cerrar los comentarios de estilo C con `*/`.

15.2 Utilizar entrada/salida orientada a flujo de tipo C++ hace los programas más legibles (y menos sujetos a errores), que sus contrapartidas escritas en C mediante las llamadas de función `printf` y `scanf`.

15.3 Colocar la declaración de una variable cerca de su primer uso puede hacer los programas más legibles.

15.4 El calificador `inline` deberá ser utilizado únicamente tratándose de funciones pequeñas, de uso frecuente.

15.5 Utilice apuntadores para pasar argumentos que pudieran ser modificados por la función llamada, y utilice referencias a constantes para pasar argumentos extensos, que no serán modificados.

15.6 Una ventaja del uso de variables `const` en lugar de constantes simbólicas, es que las variables `const` son visibles con un depurador simbólico; las constantes simbólicas `#define` no lo son.

15.7 El uso de argumentos por omisión puede simplificar la escritura de las llamadas de función. Sin embargo, algunos programadores sienten que resulta más claro especificar de manera explícita todos los argumentos.

15.8 Evite usar dentro de un programa variables del mismo nombre para distintos fines. Aunque en varias circunstancias esto es permitido, puede resultar confuso.

15.9 La homonimia de funciones que ejecutan tareas muy relacionadas, puede hacer que los programas sean más legibles y comprensibles.

Sugerencias de rendimiento

15.1 Usar funciones `inline` puede reducir tiempo de ejecución, pero puede aumentar el tamaño del programa.

15.2 En el caso de objetos extensos, utilice un parámetro de referencia `const` para simular la apariencia y la seguridad de una llamada por valor, pero evitando la sobrecarga de pasar una copia de dicho objeto extenso.

Sugerencia de portabilidad

15.1 El significado de una lista de función de parámetros vacía es dramáticamente distinto en C++ que en C. En C, significa que se deshabilita toda verificación de argumentos. En C++, significa que la función no toma argumentos. Por lo tanto, si se compilan en C++, los programas en C que utilizan esta característica pudieran ejecutarse en forma diferente.

Observación de ingeniería de software

15.1 Cualquier modificación a una función `inline` requiere que sean recompilados todos los clientes de dicha función. Esto pudiera resultar de importancia en algunas situaciones de desarrollo y mantenimiento de programas.

Ejercicios de autoevaluación

15.1 Llene cada uno de los siguientes espacios vacíos :

a) En C++, es posible tener varias funciones con el mismo nombre, cada una de ellas operando sobre diferentes tipos o números de argumentos. Esto se llama _____ de funciones.

b) La _____ permite el acceso a una global variable con el mismo nombre que una variable en el alcance actual.

c) El operador _____ asigna dinámicamente un nuevo objeto.

d) En C, suponga que `a` y `b` son variables enteras y que formamos la suma `a + b`. Ahora suponga que `c` y `d` son variables de punto flotante y que formamos la suma `c + d`. Los dos operadores `+` se están utilizando aquí para fines claramente distintos. Esto es un ejemplo de una propiedad que tiene C++, y que también tiene C, llamada _____ de función.

e) Dos objetos de flujo que hemos analizado y que están predefinidos en C++ son _____ y _____.

f) Las palabras reservadas _____, _____, _____ y _____ son utilizadas para crear nuevos tipos de datos en C++.

g) El calificador _____ sólo se utiliza para declarar variables de lectura.

h) C++ ofrece _____ para permitir a las funciones compiladas en un compilador de C que se enlacen correctamente con un programa de C++.

i) Las funciones _____ habilitan a un sola función para que pueda ser definida para ejecutar una tarea sobre muchos tipos distintos de datos.

15.2 (Verdadero/falso). Al escribir un comentario en forma de bloque que requiera de muchas líneas de texto, es más conciso el utilizar el delimitador de comentarios de C++ `//` que los delimitadores normales de C `/*` y `*/`.

15.3 En C++ , ¿por qué debería un prototipo de función contener una declaración de tipo de parámetro como `float&`?

15.4 (Verdadero/falso). Todas las llamadas en C++ se efectúan en llamada por valor.

15.5 Explique por qué en C++ los operadores `<<` y `>>` son de homonimia.

15.6 Escriba segmentos de programa en C++ para que lleven a cabo cada uno de los siguientes:

a) Escriba la cadena `"Welcome to C!"` al flujo de salida estándar `cout`.

b) Lea el valor de la variable `age` a partir del flujo de entrada estándar `cin`.

15.7 Escriba un programa en C++ completo que utilice entrada/salida de flujo y una función `inline` llamada `sphereVolume` para solicitarle al usuario el radio de una esfera, y calcular e imprimir el volumen de dicha esfera, utilizando la asignación `volume = (4/3) PI * pow(radius, 3)`.

15.8 ¿Qué pasaría en C si declarase la misma función dos veces con distintos tipos de argumentos? ¿Qué pasaría en C++?

Respuestas a los ejercicios de autoevaluación

15.1 a) homonimia. b) operador de resolución de alcance unario (`::`). c) `new`. d) homonimia. e) `cin`, `cout`. f) `enum`, `struct`, `union`, `class`. g) `const`. h) especificaciones de enlace. i) plantilla.

15.2 Falso. Los delimitadores de tipo C normales, necesita cada uno sólo ser escrito una vez, en tanto que el delimitador C++ `//` necesita aparecer al principio de cada nueva línea.

15.3 Dado que el programador está declarando un parámetro de referencia del tipo "referencia a" `float` para tener acceso mediante llamada por referencia a la variable de argumento original.

15.4 Falso. C++ permite llamada por referencia directa mediante el uso de parámetros de referencia en adición al uso de apuntadores.

15.5 El operador >> es a la vez el operador de desplazamiento a la derecha y el operador de extracción de flujo, dependiendo de dónde está utilizado. El operador << es a la vez el operador de desplazamiento de la izquierda y el operador de inserción de flujo dependiendo de dónde se utilice.

15.6 a) `cout << "Welcome to C!";`
b) `cin >> age;`

15.7
```
// Inline function that calculates the volume of a sphere
#include <iostream.h>

const float PI = 3.24159;

inline float sphereVolume(const float r) {return
                          4.0 / 3.0 * PI * r * r * r;)
main()
{
   float radius;

   cout << "Enter the length of the radius of your sphere: ";
   cin >> radius;
   cout << "Volume of sphere with radius " << radius <<
      "is " << sphereVolume(radius) << '\n';
   return 0;
}
```

15.8 En C obtendría un error de compilación indicando que las dos funciones tienen el mismo nombre. En C++ esto es un ejemplo de homonimia de funciones lo que está permitido, y no habría error.

Ejercicios

15.9 Suponga una organización que actualmente enfatiza la programación en C, y desea convertir a un entorno de programación orientado a objetos en C++. ¿Cuál sería la estrategia apropiada para pasar de entornos de C a entornos de C++?

15.10 Escriba enunciados orientados a flujo de tipo C++ para llevar a cabo cada uno de los siguientes:
a) Mostrar **"HELLO"** en la pantalla.
b) Introducir un valor para la variable `float` llamada **temperature** desde el teclado.

15.11 Compare la entrada/salida de tipo **printf/scanf** con la entrada/salida orientada a flujos de tipo C++.

15.12 En este capítulo, mencionamos que existen muchas áreas en la cual C++ "sabe" automáticamente qué tipos utilizar. Enliste tantas de éstas como usted pueda.

15.13 Escriba un programa C++ que utilice entradas/salidas orientadas a flujos, que solicite siete enteros y determine e imprima su máximo.

15.14 Escriba un programa C++ completo que utilice entrada/salida de flujos y una función `inline` de nombre **circleArea**, para solicitarle al usuario el radio de un círculo, y que calcule e imprima el área de dicho círculo.

15.15 Escriba un programa C++ completo con las tres funciones alternas especificadas más abajo, que cada una de ellas sólo añada 1 a la variable **count** definida en **main**. A continuación compare los tres métodos alternos. Las tres funciones son
a) Función **add1CallByValue**, que pasa una copia de **count** en llamada por valor, añade 1 a la copia y regresa el nuevo valor.
b) Función **add1ByPointer**, que pasa el acceso a la variable **count** vía una llamada por referencia simulada mediante apuntadores, y utiliza el operador de desreferenciación * para añadir uno a la copia original de **count** en **main**.

c) La función **add1ByReference**, que pasa **count** con una llamada por referencia verdadera vía un parámetro de referencia, y añade 1 a la copia original de **count** mediante su seudónimo (es decir el parámetro de referencia).

15.16 ¿Cuál es el propósito del operador de resolución de alcance unario?

15.17 Compare la asignación dinámica de memoria mediante los operadores de C++ **new** y **delete**, con asignación de memoria dinámica mediante las funciones de la biblioteca estándar de C **malloc** y **free**.

15.18 Enliste las varias características de C++ que fueron presentadas en este capítulo, y que representan el conjunto de mejorías no orientadas a objetos a C.

15.19 Escriba un programa que utilice una plantilla de función llamada **min**, para determinar el menor de dos argumentos. Pruebe el programa utilizando pares de enteros de caracteres y de números de punto flotante.

15.20 Escriba un programa que utilice una plantilla de función llamada **max**, para determinar el mayor de tres argumentos. Pruebe el programa utilizando pares de enteros, caracteres y números de punto flotante.

15.21 Determine si los siguientes segmentos de programa contienen errores. Para cada uno de los errores, explique cómo deben ser corregidos. Nota: para un segmento particular del programa, es posible que dentro de dicho segmento no existan errores.

a)
```
main()
{
   cout << x;
   int x = 7;
   return 0;
}
```
b)
```
template <class A>
int sum(int num1, int num2, int num3)
{
   return num1 + num2 + num3;
}
```
c)
```
/* This is a comment
```
d)
```
void printResults(int x, int y)
{
   cout << "The sum is " << x + y << '\n';
   return x + y;
}
```
e)
```
char *s = "character string";
delete s;
```
f)
```
float &ref;
*ref = 7;
```
g)
```
int x;
const int y = x;
```
h)
```
template <A>
A product(A num1, A num2, A num3)
{
   return num1 * num2 * num3;
}
```
i)
```
const double pi = 3.14159;
pi = 3;
```
j)
```
// This is a comment
```
k)
```
char *s = new char[10];
delete s;
```
l)
```
double cube(int);
int cube(int);
```

16

Clases y abstracción de datos

Objetivos

- Comprender los conceptos de ingeniería de software correspondientes a encapsulado y ocultamiento de datos.
- Comprender las nociones de abstracción de datos y de tipos de datos abstractos (ADT).
- Ser capaz de crear ADT, es decir clases, en C++.
- Comprender como crear, utilizar y destruir objetos de clase.
- Ser capaz de controlar el acceso a miembros de objetos de datos y a funciones miembro.
- Empezar a valorizar las ventajas de la orientación a objetos.

Mi objeto todo sublime, terminaré a tiempo.
W. S. Gilbert

¿Es éste un mundo para ocultar virtudes?
William Shakespeare
Décimo segunda noche

Tienes bien merecidos a tus sirvientes públicos.
Adlai Stevenson

Caras íntimas en lugares públicos
son más sabias y más agradables
Que caras públicas en lugares íntimos.
W. H. Auden

Sinopsis

16.1 Introducción

Ahora empezamos nuestra introducción a la orientación a objetos. Veremos que la orientación a objetos es una forma natural de pensar en relación con el mundo y de escribir programas de computación. ¿Por qué, entonces, desde la primera página de este libro no empezamos con la orientación a objetos? ¿Por qué estamos posponiendo hasta el capítulo 16 la programación orientada a objetos en C++? La respuesta es que los objetos que construiremos estarán compuestos en parte por piezas de programas estructurados, por lo que necesitábamos establecer primero una base de programación estructurada.

En los primeros capítulos, nos concentramos en la metodología "convencional" de la programación estructurada. En este capítulo, presentamos conceptos básicos (es decir, "objeto pensamiento") y terminología (es decir " objeto lenguaje ") de la programación orientada a objetos. En capítulos subsecuentes, atacaremos temas más sustanciales y problemas retadores utilizando las técnicas del *diseño orientado a objetos* (*OOD, por object-oriented design*), analizamos enunciados de problemas típicos que requieren la construcción de sistemas, determinamos qué

objetos necesitamos para poner en práctica los sistemas, determinamos qué atributos tendrán que tener los objetos, determinamos qué comportamientos necesitarán mostrar dichos objetos y especificamos cómo deberán interactuar los objetos uno con el otro para cumplir con las metas generales del sistema.

Empezamos presentando parte de la terminología clave de la orientación a objetos. Mire a su alrededor en el mundo real. Por todas partes donde mire los verá *¡objetos!* Personas, animales, plantas, automóviles, aviones, edificios, cortadoras de pastos, computadoras y demás. Los seres humanos pensamos en términos de objetos. Tenemos la capacidad maravillosa de la *abstracción,* que nos permite ver una imagen en pantalla como personas, aviones, árboles y montañas, en vez de puntos individuales de color. Podemos, si así lo deseamos, pensar en términos de playas, en vez de en granos de arena, bosques en lugar de árboles, y casas en lugar de tabiques.

Pudiéramos estar inclinados a dividir los objetos en dos categorías: objetos animados y objetos inanimados. De alguna forma, los objetos animados están "vivos". Se mueven y hacen cosas. Los objetos inanimados, como las toallas, no parecen hacer mucho. Simplemente "están ahí". Todos estos objetos, sin embargo, sí tienen algunas cosas en común. Todos tienen *atributos,* como tamaño, forma, color, peso y demás. Todos ellos exhiben varios *comportamientos*, por ejemplo, un balón rueda, rebota, se infla y se desinfla; un bebé llora, duerme, gatea, camina y parpadea; un automóvil se puede acelerar, frenar, girar, etcétera.

Los seres humanos aprenden lo relacionado con los objetos estudiando sus atributos y observando su comportamiento. Objetos diferentes pueden tener muchos atributos iguales y mostrar comportamientos similares. Se pueden hacer comparaciones, por ejemplo, entre bebés y adultos, entre seres humanos y chimpancés. Automóviles, camiones, pequeños carros rojos y patines tienen mucho en común.

La programación orientada a objetos (OOP) hace modelos de los objetos del mundo real mediante sus contrapartes en software. Aprovecha las relaciones de clase, donde objetos de una cierta clase, como la clase de vehículos, tienen las mismas características. Aprovecha las relaciones de *herencia*, e inclusive las relaciones de *herencia múltiple,* donde clases recién creadas de objetos se derivan heredando características de clases existentes, pero poseyendo características únicas, propias de ellos mismos. Los bebés tienen muchas características de sus padres, pero ocasionalmente padres de baja estatura tienen hijos altos.

La programación orientada a objetos nos proporciona una forma más natural e intuitiva de observar el proceso de programación, es decir *haciendo modelos* de objetos del mundo real, de sus atributos y de sus comportamientos. OOP también hace modelos de la comunicación entre los objetos. De la misma forma que las personas se envían *mensajes* uno al otro (por ejemplo, un sargento al comando de tropas para ponerlas en posición de firmes), los objetos también se comunican mediante mensajes.

OOP *encapsula* datos (atributos) y funciones (comportamiento) en paquetes llamados *objetos*; los datos y las funciones de un objeto están muy unidos. Los objetos tienen la propiedad de *ocultar la información*. Esto significa que aunque los objetos puedan saber cómo comunicarse unos con otros mediante *interfaces* bien definidas, a los objetos por lo regular no se les está permitido saber cómo funcionan otros objetos los detalles de puesta en práctica quedan ocultos dentro de los objetos mismos. Seguramente es posible conducir un automóvil eficaz sin saber los detalles de cómo es el funcionamiento interno de motores, transmisiones y sistemas de escape. Veremos por qué este ocultamiento de la información es tan crucial para una buena ingeniería de software.

En C y en otros *lenguajes de programación procedurales*, la programación tiende a ser *orientada a la acción*, en tanto que en la programación C++ tiende a ser *orientada al objeto*. En

C, la unidad de programación es la *función*. En C++, la unidad de programación es la *clase* a partir de la cual eventualmente los objetos son *producidos* (es decir son creados).

Los programadores de C se concentran en escribir funciones. Grupos de acciones que ejecutan alguna tarea común se agrupan en funciones, y a su vez las funciones se agrupan para formar programas. Es cierto que en C los datos son importantes, pero la óptica es que los datos existen de forma primordial para soportar las acciones que las funciones ejecutan. Los *verbos* en una especificación de sistema ayudan al programador de C a determinar el conjunto de funciones que juntas funcionarán para poner en práctica el sistema.

Los programadores de C++ se concentran en crear sus propios *tipos definidos por el usuario* conocidos como *clases*. Cada clase contiene datos junto con un conjunto de funciones que manipula dichos datos. Los componentes de datos de una clase se llaman *miembros de datos*. Los componentes de función de una clase se llaman *funciones miembros*. Al igual que un ejemplo de un tipo incorporado como **int** se conoce como una *variable*, un ejemplo de un tipo definido por usuario (es decir, una clase) se conoce como un *objeto*. El foco de atención en C++ está sobre los objetos, en vez de sobre las funciones. Los *nombres* en una especificación de sistema ayudan al programador de C++ a determinar el conjunto de clases, a partir de las cuales serán creados los objetos que funcionarán conjuntamente para poner en práctica el sistema.

Las clases en C++ son un desarrollo o evolución natural de la idea de **struct** de C. Antes de continuar con los detalles del desarrollo de las clases en C++, repasaremos las estructuras y construiremos un tipo definido por el usuario basado en una estructura. La debilidad que expondremos mediante este enfoque, nos ayudará a racionalizar y motivar el concepto de clase.

16.2 Definiciones de estructura

Considere la siguiente definición de estructura:

```
struct Time {
   int hour;      // 0-23
   int minute;    // 0-59
   int second;    // 0-59
};
```

Demos un repaso a la terminología de las estructuras (vea los capítulos 10 y 12). La palabra reservada **struct** presenta la definición de estructura. El identificador **Time** es la *etiqueta de estructura*. La etiqueta de estructura le da nombre a la definición de la estructura, y se utiliza para declarar variables del tipo estructura. En este ejemplo, el nombre del tipo estructura es **Time** (a diferencia del nombre de tipo más largo **struct Time**, que se requiere en C). Los nombres declarados en las llaves de la definición de estructura, son los *miembros* de la estructura. Los miembros de una misma estructura deben tener nombres únicos, pero dos estructuras distintas pueden contener sin conflicto miembros con un mismo nombre. Cada definición de estructura debe terminar con un punto y coma. Como veremos pronto, la explicación anterior es también válida para las clases.

La definición de **Time** contiene tres miembros del tipo **int**: **hour**, **minute**, y **second**. Los miembros de estructura pueden ser de cualquier tipo. Una estructura no puede, sin embargo, contener una ocurrencia de sí misma. Por ejemplo, en la definición de estructura correspondiente a **Time** no puede ser declarado un miembro del tipo **Time** Sin embargo, podría ser incluido un apuntador a una estructura **Time**. Una estructura que contenga un miembro, que es un apuntador al mismo tipo de estructura, se conoce como una *estructura autorreferenciada*. Las estructuras autorreferenciadas son útiles para formar estructuras de datos enlazados (vea el capítulo 12).

La definición anterior de estructura no reserva ningún espacio en memoria; más bien, la definición crea un nuevo tipo de datos, que es utilizado para declarar variables. Las variables de estructura se declaran, al igual que las variables de otros tipos. La declaración

```
Time timeObject, timeArray[10], *timePtr;
```

declara **TimeObject** como una variable del tipo **Time**, **TimeArray**, como un arreglo de 10 elementos del tipo **Time**, y **TimePtr** como un apuntador a un objeto **Time**.

16.3 Cómo tener acceso a miembros de estructuras

Se tiene acceso a miembros de una estructura (o de una clase) utilizando los *operadores de acceso a miembros* el *operador punto* (**.**) y el *operador flecha* (**->**). El operador punto da acceso a un miembro de estructura (o de clase) vía el nombre de variable del objeto o vía una referencia al objeto. Por ejemplo, para imprimir el miembro **hour** de la estructura **timeObject**, utilice el enunciado

```
cout << timeObject.hour;
```

El operador de flecha que consiste de un signo (**-**) y de un signo mayor que (**>**) sin espacios intermedios da acceso a un miembro de estructura (o de clase) vía un apuntador al objeto. Suponga que el apuntador **timePtr** ha sido declarado para señalar a un objeto **Time**, y que la dirección de la estructura **timeObject** ha sido asignada a **timePtr**. Para imprimir el miembro **hour** de la estructura **timeObject** mediante el apuntador **timePtr**, utilice el enunciado

```
cout << timePtr->hour;
```

La expresión **timePtr->hour** es equivalente a **(*timePtr).hour**, que desreferencia el apuntador y da acceso al miembro **hour** mediante el uso del operador de punto. Aquí se necesitan los paréntesis, porque el operador de punto (**.**) tiene una precedencia más alta que el operador de desreferenciación de apuntadores (*). El operador de flecha y el operador de miembro de estructura, junto con los paréntesis y los corchetes (**[]**) tienen la precedencia de operadores más alta (de todos los operadores hasta ahora analizados) y se asocian de izquierda a derecha.

16.4 Cómo poner en práctica mediante un struct un tipo Time definido por el usuario

En la figura 16.1 se crea un tipo de estructura definida por el usuario **Time** con tres miembros enteros: **hour**, **minute** y **second**. El programa define una estructura **Time** llamada **dinnerTime**, y utiliza el operador de punto para inicializar los miembros de la estructura con los valores **18** para **hour**, **30** para **minute** y **0** para **second**. A continuación, el programa imprime la hora en formato militar y en formato estándar. Note que las funciones de impresión reciben referencias a las estructuras constantes **Time**. Esto hace que las estructuras **Time** sean pasadas por referencia a las funciones de impresión eliminando por lo tanto, la sobrecarga de copia asociada al pasar por valor estructuras a las funciones y también evita que las funciones de impresión modifiquen la estructura **Time**. En el capítulo 17 analizaremos los objetos **const** y las funciones miembro **const**.

Sugerencia de rendimiento 16.1

Las estructuras por lo regular pasan en llamada por valor. Para evitar la sobrecarga de copiar una estructura, pase la estructura en llamada por referencia.

```
// FIG16_1.CPP
// Create a structure, set its members, and print it.
#include <iostream.h>

struct Time {     // structure definition
   int hour;      // 0-23
   int minute;    // 0-59
   int second;    // 0-59
};

void printMilitary(const Time &);  // prototype
void printStandard(const Time &);  // prototype

main()
{
   Time dinnerTime;     // variable of new type Time

   // set members to valid values
   dinnerTime.hour = 18;
   dinnerTime.minute = 30;
   dinnerTime.second = 0;

   cout << "Dinner will be held at ";
   printMilitary(dinnerTime);
   cout << " military time,\nwhich is ";
   printStandard(dinnerTime);
   cout << " standard time." << endl;

   // set members to invalid values
   dinnerTime.hour = 29;
   dinnerTime.minute = 73;
   dinnerTime.second = 103;

   cout << "\nTime with invalid values: ";
   printMilitary(dinnerTime);
   cout << endl;
   return 0;
}

// Print the time in military format
void printMilitary(const Time &t)
{
   cout << (t.hour < 10 ? "0" : "") << t.hour << ":"
        << (t.minute < 10 ? "0" : "") << t.minute << ":"
        << (t.second < 10 ? "0" : "") << t.second;
}

// Print the time in standard format
void printStandard(const Time &t)
{
   cout << ((t.hour == 0 || t.hour == 12) ? 12 : t.hour % 12)
        << ":" << (t.minute < 10 ? "0" : "") << t.minute
        << ":" << (t.second < 10 ? "0" : "") << t.second
        << (t.hour < 12 ? " AM" : " PM");
}
```

Fig. 16.1 Cómo crear una estructura, definir sus miembros e imprimirla (parte 1 de 2).


```
Dinner will be held at 18:30:00 military time,
which is 6:30:00 PM standard time.

Time with invalid values: 29:73:103
```

Fig. 16.1 Cómo crear una estructura, definir sus miembros e imprimirla (parte 2 de 2).

Sugerencia de rendimiento 16.2

A fin de evitar la sobrecarga de la llamada por valor y aún así obtener el beneficio de que la información original del llamador quede protegida contra modificaciones, pase argumentos de tamaño extenso como referencias `const`.

Existen inconvenientes para la creación de nuevos tipos de datos utilizando estructuras de esta forma. Dado que la inicialización no es específicamente requerida, es posible tener datos sin inicializar, con los problemas consiguientes. Aún si los datos están inicializados, pudieran no estar de forma correcta inicializados. A los miembros de una estructura se les pueden asignar valores inválidos (como lo hicimos en la figura 16.1) porque el programa tiene acceso directo a los datos. El programa asignó valores incorrectos a los tres miembros del objeto **`time dinnerTime`**. Si es modificada la puesta en marcha del **`struct`**, deberán de ser modificados todos los programas que utilizan el **`struct`**. Esto es debido a que el programador está manipulando de manera directa el tipo de datos. No existe "interfaz" con el programa para asegurarse que el programador utiliza correctamente el tipo de datos, y para asegurar que los datos se conservan en un estado consistente.

Observación de ingeniería de software 16.1

Es importante escribir programas que sean comprensibles y fáciles de mantener. Las modificaciones son la regla más que la excepción. Los programadores deberán prever que su código será modificado. Como veremos, las clases facilitan la capacidad de modificación de los programas.

Existen otros problemas asociados con las estructuras de tipo C. No pueden ser impresas como unidad, más bien sus miembros deben ser impresos, y se les debe dar formato uno por uno. Se podría escribir una función para imprimir los miembros de una estructura en algún formato apropiado. En el capítulo 18, "Homonimia de operadores", se ilustra cómo hacer la homonimia del operador << para habilitar los objetos de un tipo de estructura o de clase para que sean impresos con facilidad. Las estructuras no pueden ser comparadas en su totalidad; deben ser comparadas miembro por miembro. En el capítulo 18 también se ilustra cómo hacer la homonimia de los operadores de igualdad y relacionales, a fin de comparar objetos de tipos de estructura y de clase.

En la sección siguiente se vuelve a poner en práctica nuestra estructura **`Time`** como una clase y se demuestran algunas de las ventajas de la creación de tipos de datos abstractos con las clases. Veremos que en C++ las clases y las estructuras pueden ser utilizadas prácticamente en forma idéntica. La diferencia entre ambas es la accesibilidad por omisión asociada con los miembros de cada una. Esto será explicado en breve.

16.5 Cómo poner en práctica un tipo de dato abstracto Time con una clase

Las clases permiten que el programador modele objetos que tienen *atributos* (representados como *miembros de datos*) y *comportamientos* u *operaciones* (representados como *funciones*

miembro). Los tipos contienen miembros de datos y funciones miembro y en C++ son por lo regular definidos mediante la palabra reservada **class**.

Las funciones miembro en otros lenguajes de programación orientados a objetos a veces se llaman *métodos*, y son invocados en respuesta a *mensajes* enviados a un objeto. Un mensaje corresponde a una llamada de función miembro.

Una vez que se haya definido una clase, el nombre de la clase puede ser utilizado para declarar objetos de dicha clase. En la figura 16.2 se muestra una definición simple para la clase **Time**.

Nuestra definición de la clase **Time** empieza con la palabra reservada **class**. El *cuerpo* de la definición de clase se delimita mediante llaves izquierdas y derechas ({ y }). La definición de clase termina con un punto y coma. Nuestra definición de clase **Time** y nuestra definición de estructura **Time** cada una de ellas contiene tres miembros enteros **hour**, **minute** y **second**.

Error común de programación 16.1

Olvidar el punto y coma al final de una definición de clase.

Las partes restantes de la definición de clase son nuevas. La etiqueta **public:** y **private:** se conocen como especificadores de acceso de miembro. Cualquier miembro de datos o función miembro declarado después del especificador de acceso de miembro **public:** (y antes del siguiente especificador de acceso de miembro) es accesible, siempre que el programa tenga acceso a un objeto de la clase **Time**. Cualquier miembro de datos o función miembro declarada después del especificador de acceso de miembro **private:** (y hasta el siguiente especificador de acceso de miembro) sólo es accesible a las funciones miembros de la clase. Los especificadores de acceso de miembro terminan siempre con dos puntos (:) y pueden aparecer varias veces en una definición de clase.

Práctica sana de programación 16.1

Para mayor claridad y legibilidad, utilice cada especificador de acceso de miembro sólo una vez en una definición de clase. Coloque primero los miembros públicos, donde sean fácilmente localizables.

La definición de clase contiene prototipos para las cuatro siguientes funciones miembros después del especificador de acceso de miembros **public: Time, setTime, printMilitary** y **printStandard**. Estas son las *funciones miembro públicas*, o *servicios públicos* o *interfaz de la clase*. Estas funciones serán usadas por los clientes (usuarios) de la clase para manipular los datos de la clase.

```
class Time {
public:
   Time();
   void setTime(int, int, int);
   void printMilitary();
   void printStandard();
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};
```

Fig. 16.2 Definición simple de la **class Time**.

Una función miembro con el mismo nombre que la clase se llama una función *constructor* de dicha clase. Un constructor es una función miembro especial, que inicializa los miembros de datos de un objeto de clase. Cuando se crea un objeto de una clase se llama a la función constructor de dicha clase.

Los tres miembros enteros aparecen después del especificador de acceso de miembro **private:**. Esto indica que estos miembros de datos de la clase son accesibles sólo a las funciones miembro y como veremos en el siguiente capítulo, a los amigos de la clase. Entonces, los miembros de datos solamente son accesibles por las cuatro funciones cuyos prototipos aparecen en la definición de clase (o por amigos de la clase). Por lo regular los miembros de datos aparecen listados en la porción **private:** de una clase y normalmente las funciones miembro aparecen listados en la porción **public:**. Como veremos más adelante es posible que existan funciones miembro **private:** y datos **public:**.

Una vez definida la clase, puede ser utilizada como un tipo en declaraciones como sigue:

```
Time sunset,                  // object of type Time
     arrayOfTimes[5],         // array of Time objects
     *pointerToTime,          // pointer to a Time object
     &dinnerTime = sunset;    // reference to a Time object
```

El nombre de clase se convierte en un nuevo especificador de tipo. Pueden existir muchos objetos de una clase, igual que pueden existir muchas variables del tipo **int**. El programador puede crear nuevos tipos de clase según requiera. Esta es una de las muchas razones por las que C++ es un *lenguaje extensible*.

En la figura 16.3 se utiliza la clase **Time**. El programa produce un objeto de la clase **Time** llamado **t**. Cuando se produce el objeto, en forma automática es llamado el constructor **Time** e inicializa a **0** cada miembro de dato privado. La hora es entonces impresa en formatos militar y estándar, a fin de confirmar que los miembros han sido inicializados de forma correcta. A continuación, se ajusta la hora utilizando la función miembro **setTime** y se vuelve a imprimir la hora en ambos formatos. A continuación la función miembro **setTime** intenta establecer los miembros de datos a valores inválidos, y otra vez se imprime la hora en ambos formatos.

```
// FIG16_3.CPP
// Time class.
#include <iostream.h>

// Time abstract data type (ADT) definition
class Time {
public:
   Time();                          // default constructor
   void setTime(int, int, int);     // set hour, minute and second
   void printMilitary();            // print military time format
   void printStandard();            // print standard time format

private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};
```

Fig. 16.3 Puesta en práctica de tipos de datos abstractos Time como una clase (parte 1 de 3).

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

```
// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}

// Print Time in military format
void Time::printMilitary()
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}

// Print time in standard format
void Time::printStandard()
{
   cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
        << ":" << (minute < 10 ? "0" : "") << minute
        << ":" << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}

// Driver to test simple class Time
main()
{
   Time t;  // instantiate object t of class Time

   cout << "The initial military time is ";
   t.printMilitary();
   cout << "\nThe initial standard time is ";
   t.printStandard();

   t.setTime(13, 27, 6);
   cout << "\n\nMilitary time after setTime is ";
   t.printMilitary();
   cout << "\nStandard time after setTime is ";
   t.printStandard();

   t.setTime(99, 99, 99);  // attempt invalid settings
   cout << "\n\nAfter attempting invalid settings:\n"
        << "Military time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();
   cout << endl;
   return 0;
}
```

Fig. 16.3 Puesta en práctica de tipos de datos abstractos Time como una clase (parte 2 de 3).


```
The initial military time is 00:00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00:00:00
Standard time: 12:00:00 AM
```

Fig. 16.3 Puesta en práctica de tipos de datos abstractos Time como una clase (parte 3 de 3).

Otra vez, note que los miembros de datos **hour**, **minute** y **second** están antecedidos por la etiqueta **private:**. Los miembros de datos privados de una clase Por lo regular no son accesibles fuera de la misma. (Otra vez, veremos en el capítulo 17, que los amigos de una clase pueden tener acceso a los miembros privados de dicha clase.) La filosofía aquí es que la representación de datos reales utilizados dentro de una clase no debe importarle a los clientes de dicha clase. En este sentido, la puesta en práctica de una clase se dice que está *oculta* a sus clientes. Este *ocultamiento de la información* promueve la capacidad de modificación del programa y simplifica la percepción del cliente de una clase.

Observación de ingeniería de software 16.2

Los clientes de una clase utilizan la clase sin conocer los detalles internos de cómo está puesta en práctica dicha clase. Si se modifica la puesta en práctica de la clase (para mejorar el rendimiento, por ejemplo) no es necesario modificar los clientes de la clase. Esto facilita mucho la modificación de sistemas.

En este programa, el constructor **Time** sólo inicializa a 0 los miembros de datos (es decir, el equivalente en hora militar de 12 AM). Esto asegura que cuando es creado el objeto está en un estado consistente. Los valores inválidos no pueden ser almacenados en los miembros de datos, porque al crearse el objeto **Time** el constructor es automáticamente llamado y todos los intentos subsiguientes para modificar los miembros de datos son analizados por la función **setTime**.

Observación de ingeniería de software 16.3

Las funciones miembro por lo regular están formadas por unas pocas líneas de código, porque ninguna lógica es requerida para determinar si son válidos los miembros de datos.

Note que los miembros de datos de una clase no pueden ser inicializados donde son declarados en el cuerpo de la clase. Estos miembros de datos deberán ser inicializados por el constructor de la clase, o las funciones "set" les podrán asignar valores.

Error común de programación 16.2

Intentar inicializar de forma explícita un miembro de datos de una clase.

La función con el mismo nombre que la clase, pero precedido por un *carácter tilde (~)* se llama el *destructor* de dicha clase. El destructor se ocupa del trabajo de terminación en cada objeto de clase, antes que la memoria correspondiente al objeto sea reclamada por el sistema. En este ejemplo no se incluye un destructor. Analizaremos con mayor detalle constructores y destructores, más adelante en este capítulo, y en el capítulo 17.

Note que las funciones que la clase proporciona al mundo exterior están precedidas por la etiqueta **public:**. Las funciones públicas ponen en práctica las capacidades que la clase proporciona a sus clientes. Las funciones públicas de una clase se conocen como *interfaz* o *interfaz pública* de la clase.

Observación de ingeniería de software 16.4

Los clientes tienen acceso a la interfaz de una clase, pero no deberán tener acceso a la puesta en práctica de la clase.

La definición de clase contiene declaraciones de los miembros de datos de la clase y de las funciones miembro de la clase. Las declaraciones de funciones miembro son los prototipos de función, que fueron analizados en capítulos anteriores. Las funciones miembro pueden ser definidas dentro de una clase, pero es una práctica sana de programación definir las funciones fuera de la definición de clase.

Práctica sana de programación 16.2

Defina todas, a excepción de las más pequeñas funciones miembro, por afuera de la definición de clase. Esto ayuda a separar la interfaz de la clase de su puesta en práctica.

Note el uso del operador de *resolución de alcance binario (::)* en cada definición de función miembro, que en la figura 16.3 sigue a la definición de clase. Una vez definida una clase y sus funciones miembro declaradas, estas funciones deben ser definidas. Cada función miembro de clase puede ser definida en directo al cuerpo de la clase (en vez de incluir el prototipo de función de la clase) o la función puede ser definida después del cuerpo de la clase. Cuando una función miembro se define después de su correspondiente definición de clase, el nombre de función es antecedido por el nombre de clase y por el operador de resolución de alcance binario (**::**). Dado que diferentes clases pueden tener los mismos nombres de miembros, el operador de resolución de alcance "une" el nombre del miembro con el nombre de la clase, para fijar en forma única las funciones miembro de una clase en particular.

Aún cuando una función miembro declarada en una definición de clase pudiera definirse fuera de esa definición de clase, esa función miembro aún queda dentro del *alcance de la clase*, es decir su nombre es conocido sólo por otros miembros de la clase, a menos que se haga referencia a él vía un objeto de la clase, vía una referencia a un objeto de la clase, o vía un apuntador a un objeto de la clase. Diremos en breve más en relación con el alcance de la clase.

Es posible definir las funciones miembro en el cuerpo de la definición de clase. Las funciones que tengan más de una o dos líneas, se definen normalmente por afuera de la definición de clase, esto ayuda a separar la interfaz de clase de la puesta en práctica de la clase. Si se define una función miembro en una definición de clase, la función miembro quedará automáticamente en línea. Las funciones miembro definidas fuera de la definición de clase pudieran ser puestas en línea, mediante el uso explícito de la palabra reservada **inline**. Recuerde que el compilador se reserva el derecho de poner o no en línea cualquier función.

Observación de ingeniería de software 16.5

Declarar funciones miembro dentro de una definición de clase y definir dichas funciones miembro por fuera de dicha definición de clase separa la interfaz de una clase de su puesta en práctica. Esto promueve buena ingeniería de software.

Sugerencia de rendimiento 16.3

Definir una función miembro pequeña dentro de una definición de clase automática coloca la función miembro en línea (si el compilador así lo decide). Esto puede mejorar el rendimiento, pero no promueve una mejor ingeniería de software.

Resulta interesante que las funciones miembro **printMilitary** y **printStandard** no toman argumentos. Esto es debido a que las funciones miembro saben que deben imprimir los miembros de datos del objeto particular **Time** para los cuales han sido invocadas. Esto hace que las llamadas de funciones miembro sean más concisas que las llamadas de función convencionales de la programación procedural.

Observación de ingeniería de software 16.6

Usar un enfoque de programación orientado a objetos a menudo puede simplificar las llamadas de función al reducir el número de parámetros a pasarse. Este beneficio de la programación orientada a objetos se deriva del hecho que el encapsulado de los miembros de datos y las funciones miembros dentro de un objeto le da a las funciones miembro el derecho de acceso a los miembros de datos.

Las clases simplifican la programación porque el cliente (o el usuario del objeto de clase) sólo necesita preocuparse de las operaciones encapsuladas o incrustadas dentro del objeto. Dichas operaciones por lo regular están diseñadas para ser orientadas al cliente, más bien que orientadas a la puesta en práctica. Los clientes no necesitan preocuparse con la puesta en práctica de la clase. De hecho las puestas en práctica cambian, pero con menos frecuencia que las interfaces. Cuando se modifique una puesta en práctica, aquel código que sea dependiente de la puesta en práctica, deberá ser modificado. Mediante la ocultación de la puesta en práctica, eliminamos la posibilidad de que otras partes del programa se hagan dependientes de los detalles de la puesta en práctica de la clase.

A menudo, no es necesario crear clases "desde cero". Más bien, pueden ser *derivadas*, partiendo de otras clases que contienen operaciones que las nuevas clases pueden utilizar. O las clases pueden incluir como miembros objetos de otras clases. Esta *reutilización del software* puede mejorar el rendimiento del programador de forma significativa. Se llama *herencia* derivar nuevas clases partiendo de clases existentes, y en el capítulo 19 se analizará en detalle. Incluir clases como miembros de otras clases se llama *composición*, y se analiza en el capítulo 17.

16.6 Alcance de clase y acceso a miembros de clase.

Los nombres de variables y los nombres de función declarados en una definición de clase, y los nombres de datos y funciones miembro de una clase, pertenecen al *alcance de dicha clase*. Las funciones no miembros se definen en *alcance de archivo*.

Dentro del alcance de clase, los miembros de clase son accesibles de inmediato por todas las funciones miembro de dicha clase y pueden ser referenciados sólo por su nombre. Fuera del alcance de una clase, los miembros de clase se referencían, ya sea a través del nombre del objeto, una referencia a un objeto, o un apuntador a un objeto.

Las funciones miembros de una clase pueden tener homónimas, pero sólo por funciones dentro del alcance de dicha clase. Para hacer la homonimia de una función miembro, sólo incluya en la definición de clase un prototipo para cada versión de la función homónima, y proporcione una definición de función por separado para cada una de las versiones de la función.

Error común de programación 16.3

Intentar hacer la homonimia de una función miembro mediante una función no incluida en el alcance de dicha clase.

Las funciones miembro tienen *alcance de función* dentro de una clase. Si la función miembro define una variable con el mismo nombre que una variable con alcance de clase, la variable con alcance de clase queda oculta por la variable con alcance de función, dentro del alcance de función. Se puede tener acceso a esta variable oculta mediante el operador de resolución de alcance precediendo el operador con el nombre de la clase. Se puede tener acceso a las variables globales ocultas mediante el operador de resolución de alcance unario (vea el capítulo 15).

Los operadores utilizados para tener acceso a los miembros de clase son idénticos a los operadores para tener acceso a los miembros de estructura. El *operador de selección de miembro de punto (.)* se combina con el nombre de una objeto o con una referencia a un objeto para tener acceso a los miembros del objeto. El *operador de selección de miembro de flecha (->)* se combina con un apuntador a un objeto para tener acceso a los miembros de dicho objeto.

El programa de la figura 16.4 utiliza una clase simple llamada **Count** con el miembro de datos público **x** del tipo **int**, y la función miembro pública **print** para ilustrar el acceso a los miembros de una clase mediante los operadores de selección de miembros. El programa produce tres variables del tipo **Count counter, counterRef** (una referencia a un objeto **Count**), y **counterPtr** (un apuntador a un objeto **Count**). La variable **counterRef** se declara para hacer referencia a **counter**, y la variable **counterPtr** se declara para señalar a **counter**. Es importante hacer notar que aquí el miembro de datos **x** se ha hecho público simplemente para demostrar cómo se tiene acceso a los miembros públicos. Como hemos indicado, típicamente los datos se hacen privados como lo haremos en todos los ejemplos subsiguientes.

16.7 Cómo separar la interfaz de la puesta en práctica

Uno de los principios fundamentales de buena ingeniería de software es la separación de la interfaz de la puesta en práctica. Esto facilita la modificación de los programas. Por lo que se refiere a los clientes de una clase, los cambios en la puesta en práctica de una clase no afectan al cliente, siempre y cuando no sea modificada la interfaz de la clase.

Observación de ingeniería de software 16.7

Coloque la declaración de la clase en un archivo de cabecera a incluirse por cualquier cliente que desee utilizar dicha clase. Esto forma la interfaz pública de la clase. Coloque las definiciones de las funciones miembro de la clase en un archivo fuente. Esto conforma la puesta en práctica de la clase.

Observación de ingeniería de software 16.8

Los clientes de una clase no necesitan ver el código fuente de la clase a fin de utilizarla. Los clientes deben, sin embargo, poder tener la capacidad de enlazarse con el código objeto de la clase.

Esto alienta a los fabricantes independientes de software (ISV, por *independent software vendors*) a proporcionar bibliotecas de clase, a la venta o para su licencia. En sus productos los ISV ofrecen sólo los archivos de cabecera y los módulos de objeto. Ninguna información propietaria se da a conocer como sería el caso si se proporcionara código fuente. La comunidad de usuarios de C++ se beneficia al tener disponibles mas bibliotecas de clase, producidas por los ISV.

De hecho, las cosas no son tan color de rosa. Los archivos de cabecera sí contienen alguna porción de la puesta en práctica, así como sugerencias en relación con ella. Las funciones miembro en línea, por ejemplo, necesitan estar en un archivo de cabecera, por lo que cuando el compilador compila a un cliente, el cliente pueda incluir en su lugar la definición de función en línea. Los miembros privados son listados en la declaración de clase dentro del archivo de cabecera, por lo


```
// FIG16_4.CPP
// Demonstrating the class member access operators . and ->
//
// Caution: In future examples we will use private data.
#include <iostream.h>

// Simple class Count
class Count {
public:
   int x;
   void print() { cout << x << '\n'; }
};

main()
{
   Count counter,                  // create counter object
         *counterPtr = &counter,   // pointer to counter
         &counterRef = counter;    // reference to counter

   cout << "Assign 7 to x and print using the object's name: ";
   counter.x = 7;          // assign 7 to data member x
   counter.print();        // call member function print

   cout << "Assign 8 to x and print using a reference: ";
   counterRef.x = 8;       // assign 8 to data member x
   counterRef.print();     // call member function print

   cout << "Assign 10 to x and print using a pointer: ";
   counterPtr->x = 10;     // assign 10 to data member x
   counterPtr->print();    // call member function print
   return 0;
}
```

```
Assign 7 to x and print using the object's name: 7
Assign 8 to x and print using a reference: 8
Assign 10 to x and print using a pointer: 10
```

Fig. 16.4 Cómo tener acceso a los miembros de datos de un objeto y a las funciones miembro a través del nombre del objeto, a través de una referencia o a través de un apuntador al objeto.

que estos miembros quedan visibles a los clientes, aún cuando los clientes no puedan tener acceso a ellos.

Observación de ingeniería de software 16.9

Aquella información de importancia a la interfaz con una clase debería estar incluida en el archivo de cabecera. Aquella información que sólo será utilizada internamente en la clase y que no será necesaria para los clientes de la clase, debería ser incluida en el archivo fuente no publicado. Este es otra vez otro ejemplo del principio del mínimo privilegio.

En la figura 16.5 el programa de la figura 16.3 se divide en varios archivos. Al construir un programa C++, cada definición de clase por lo regular es colocado en un *archivo de cabecera*, y

las definiciones de funciones miembro de dicha clase se colocan en los *archivos de código fuente* con el mismo nombre base. Los archivos de cabecera se incluyen (mediante **#include**) en cada uno de los archivos en los cuales se utiliza la clase, y se compila el archivo fuente de las definiciones de funciones miembro y se enlaza con el archivo que contiene el programa principal. Vea la documentación correspondiente a su compilador, a fin de determinar cómo compilar y enlazar programas formados de varios archivos fuente.

La figura 16.5 está formada del archivo de cabecera **time1.h** en el cual se declara la clase **Time**, el archivo **time1.cpp** en el cual se definen las funciones miembro de la clase **Time**, y del archivo **fig16_5.cpp** en el cual se define la función **main**. La salida para este programa es idéntica a la salida de la figura 16.3.

Note que la declaración de clase se encierra en el siguiente código de preprocesador (vea el capítulo 13):

```
// prevent multiple inclusions of header file
#ifndef TIME1_H
#define TIME1_H
   ...
#endif
```

Cuando elaboremos programas más extensos, se colocarán también otras definiciones y declaraciones en los archivos de cabecera. Las directivas anteriores de preprocesador evitan que se incluya el código entre **#ifndef** y **#endif**, si ya ha sido definido el nombre **TIME1_H**. Si el encabezado no ha sido incluido previamente en un archivo, el nombre **TIME1_H** queda definido mediante la directiva **#define** y los enunciados del archivo de cabecera serán incluidos. Si el encabezado ha sido ya incluido, **TIME1_H** ya está definido y el archivo de cabecera no se vuelve a incluir. Nota: la regla convencional que utilizamos para el nombre de la constante simbólica en las directrices de preprocesador es sólo el nombre del archivo de cabecera, con el carácter de subrayado substituyendo el punto.

Práctica sana de programación 16.3

*Utilice en un programa las directivas de preprocesador **#ifndef**, **#define** y **#endif** para evitar que los archivos de cabecera sean incluidos más de una vez.*

Práctica sana de programación 16.4

*Utilice el nombre del archivo de cabecera, con el punto reemplazado por un subrayado, en las directivas de preprocesador **#ifndef** y **#define** correspondientes a un archivo de cabecera.*

16.8 Cómo controlar el acceso a miembros

Las etiquetas **public:** y **private:** (así como **protected:**, como veremos en el capítulo 19, "Herencia") se utilizan para controlar el acceso a los miembros de datos y a las funciones miembro de una clase. El modo de acceso para las clases es **private:** por omisión, de tal forma que todos los miembros que aparezcan después del encabezado de clase y antes de la primera etiqueta son privados. Después de cada una de las etiquetas, el modo que fue invocado por dicha etiqueta será aplicado hasta la siguiente etiqueta, o hasta la llave derecha de terminación (}) de la definición de clase. Pueden repetirse las etiquetas **public:**, **private:** y **protected:**, pero este tipo de utilización es raro y puede ser confuso.


```
// TIME1.H
// Declaration of the Time class.
// Member functions are defined in TIME1.CPP

// prevent multiple inclusions of header file
#ifndef TIME1_H
#define TIME1_H

// Time abstract data type definition
class Time {
public:
   Time();                          // default constructor
   void setTime(int, int, int);     // set hour, minute and second
   void printMilitary();            // print military time format
   void printStandard();            // print standard time format
private:
   int hour;      // 0 - 23
   int minute;    // 0 - 59
   int second;    // 0 - 59
};

#endif
```

Fig. 16.5 Archivo de cabecera de la clase **Time** (parte 1 de 4).

```
// TIME1.CPP
// Member function definitions for Time class.
#include <iostream.h>
#include "time1.h"

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military time.
// Perform validity checks on the data values.
// Set invalid values to zero (consistent state).
void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}

// Print Time in military format
void Time::printMilitary()
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}
```

Fig. 16.5 Archivo fuente de las definiciones de funciones miembro de la clase **Time** (parte 2 de 4).

```
// Print time in standard format
void Time::printStandard()
{
   cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
        << ":" << (minute < 10 ? "0" : "") << minute
        << ":" << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}
```

Fig. 16.5 Archivo fuente de las definiciones de las funciones miembro de la clase **Time** (parte 3 de 4).

```
// FIG16_5.CPP
// Driver for Time1 class
// NOTE: Compile with TIME1.CPP
#include <iostream.h>
#include "time1.h"

// Driver to test simple class Time
main()
{
   Time t;  // instantiate object t of class time

   cout << "The initial military time is ";
   t.printMilitary();
   cout << "\nThe initial standard time is ";
   t.printStandard();

   t.setTime(13, 27, 6);
   cout << "\n\nMilitary time after setTime is ";
   t.printMilitary();
   cout << "\nStandard time after setTime is ";
   t.printStandard();

   t.setTime(99, 99, 99);  // attempt invalid settings
   cout << "\n\nAfter attempting invalid settings:\n"
        << "Military time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();
   cout << endl;
   return 0;
}
```

```
The initial military time is 00:00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00:00:00
Standard time: 12:00:00 AM
```

Fig. 16.5 Programa manejador de la clase **Time** (parte 4 de 4).

Se puede tener acceso a los miembros de clase privado sólo por miembros (y amigos) de dicha clase. Se puede tener acceso a los miembros públicos de una clase mediante cualquier función del programa.

El fin primordial de los miembros públicos es presentar a los clientes de la clase un panorama de los servicios que la clase proporciona. Este conjunto de servicios forma la *interfaz pública* de la clase. Los clientes de la clase no necesitan preocuparse de cómo la clase ejecuta sus tareas. Los miembros privados de una clase, así como las definiciones de sus funciones miembro públicas, no son accesibles a los clientes de una clase. Estos componentes forman la *puesta en práctica* de la clase.

Observación de ingeniería de software 16.10

C++ alienta a que los programas sean independientes de la puesta en práctica. Cuando se modifica la puesta en práctica de una clase utilizada mediante código independiente de la misma, dicho código no necesita ser cambiado, pero pudiera necesitar ser recopilado.

Error común de programación 16.4

Un intento por parte de una función, que no sea un miembro de una clase particular (o un amigo de dicha clase) de obtener acceso a un miembro privado de dicha clase.

En la figura 16.6, se demuestra que los miembros de clase privados son accesibles a través de la interfaz de clase pública mediante el uso de las funciones miembro públicos. Cuando se compila este programa, el compilador genera dos errores, indicando que no es accesible el miembro privado especificado en cada uno de los enunciados. La figura 16.6 incluye a **time1.h** y se compila con **time1.cpp** partiendo de la figura 16.5.

```
// FIG16_6.CPP
// Demonstrate errors resulting from attempts
// to access private class members.
#include <iostream.h>
#include "time1.h"

main()
{
   Time t;

   // Error: 'Time::hour' is not accessible
   t.hour = 7;

   // Error: 'Time::minute' is not accessible
   cout << "minute = " << t.minute;

   return 0;
}
```

```
Compiling FIG16_6.CPP:
Error FIG16_6.CPP 12: 'Time::hour' is not accessible
Error FIG6_6.CPP 15: 'Time::minute' is not accessible
```

Fig. 16.6 Intento erróneo de acceso a los miembros privados de una clase.

Práctica sana de programación 16.5

Si en una declaración de clase decide listar primero los miembros privados, utilice en forma explícita la etiqueta `private:` *a pesar de que por omisión se supone que* `private:` *es asumido. Esto mejora la claridad del programa. Nuestra preferencia es listar primero los miembros* `public:` *de una clase.*

Práctica sana de programación 16.6

A pesar del hecho que las etiquetas `public:` *y* `private:` *pueden ser repetidas y entremezcladas, liste primero en un grupo todos los miembros públicos de una clase y a continuación liste en otro grupo todos los miembros privados. Esto enfoca la atención del cliente en la interfaz pública de la clase, en vez de en la puesta en práctica de la misma.*

Observación de ingeniería de software 16.11

Conserve privados todos los miembros de datos de una clase. Proporcione funciones miembro públicas para definir los valores de los miembros de datos privados y para obtener los valores de los miembros de datos privados. Esta arquitectura ayuda a ocultar la puesta en práctica de una clase a la vista de sus clientes, lo que reduce errores y mejora la capacidad de modificación del programa.

El cliente de una clase puede ser una función miembro de otra clase, o puede ser una función global.

Para miembros de una clase el acceso por omisión es privado. El acceso a miembros de una clase puede ser definido en forma explícita a protegido o a público. El acceso por omisión correspondiente a miembros `struct` y a miembros de unión es `public`. El acceso a miembros de un `struct` puede ser definido en forma explícita a `public` o a `private` (o a `protected`, como veremos en el capítulo 19). Tratándose de una `unión`, los especificadores de acceso de miembros no pueden ser utilizados en forma explícita.

Observación de ingeniería de software 16.12

Los diseñadores de clase utilizan miembros privados, protegidos y públicos para obligar a la idea de ocultamiento de información y al principio del mínimo privilegio.

Note que los miembros de una clase son privados por omisión, por lo que nunca es necesario utilizar de forma explícita el especificador de acceso de miembro `private:` Muchos programadores prefieren, sin embargo, listar primero la interfaz a una clase (es decir, los miembros públicos de la clase); a continuación se listan los miembros privados, con lo que se requiere el uso explícito del especificador de acceso de miembros `private:` dentro de la definición de clase.

Práctica sana de programación 16.7

Evita confusión usar los especificadores de acceso de miembro `public:`, `protected:`, *y* `private:` *una vez en cada definición de clase.*

El simple hecho que los datos de clase sean `private` no necesariamente significa que los clientes no puedan efectuar modificaciones a dichos datos. Los datos pueden ser modificados por funciones miembro o amigos de dicha clase. Como veremos, dichas funciones deberán ser diseñadas para asegurar la integridad de los datos.

El acceso a los datos privados de una clase puede ser controlado cuidadosamente mediante el uso de las funciones miembro, llamadas *funciones de acceso*. Por ejemplo, para permitir a los clientes leer el valor de datos privados, la clase puede proporcionar una *función "get". Para*

permitir que los clientes modifiquen datos privados, la clase puede proporcionar una *función "set"*. Una modificación como ésta parecería violar el concepto de datos privados. Pero una función miembro *set* puede proporcionar capacidades de validación de datos (como la verificación de rangos) para asegurarse que el valor se define de forma correcta. Una función *get* no necesita exponer los datos en formato "en bruto"; más bien la función *get* puede editar los datos y limitar la parte de dichos datos que el cliente verá.

Observación de ingeniería de software 16.13

Hacer privados los miembros de datos de una clase y públicas las funciones miembro de la clase, facilita la depuración, porque los problemas con las manipulaciones de datos se localizan ya sea en las funciones miembros de la clase o en los amigos de la misma

16.9 Funciones de acceso y funciones de utilería

No todas las funciones miembro necesitan ser públicas para servir como parte de la interfaz de una clase. Algunas funciones miembro se conservan privadas, y sirven como funciones de utilería para otras funciones de la clase.

Observación de ingeniería de software 16.14

Las funciones miembro tienen tendencia a agruparse en varias categorías diferentes: funciones que leen y regresan el valor de miembros de datos privados, funciones que definen el valor de miembros de datos privados, funciones que ponen en práctica las características de la clase, y funciones que ejecutan varias tareas mecánicas para la clase, como la inicialización de objetos de clase, la asignación de objetos de clase, la conversión entre clases y tipos incorporados o entre clases y otras clases, y el manejo de memoria para objetos de clase.

Las funciones de acceso pueden leer o desplegar datos. Otro uso común de las funciones de acceso es probar la veracidad o falsedad de condiciones —dichas funciones a menudo se conocen como *funciones predicadas*. Un ejemplo de una función predicada sería una función `isEmpty` para cualquier clase contenedor, como es una lista enlazada, una pila o una cola. Un programa probaría `isEmpty` antes de intentar leer cualquier otro elemento del objeto contenedor. Una función predicada `isFull` pudiera probar un objeto de clase contenedor para determinar si ya no tiene espacio adicional.

En la figura 16.7 se demuestra el concepto de una *función de utilería*. Una función de utilería no forma parte de la interfaz de una clase. Más bien, es una función miembro privada, que da apoyo a la operación de las funciones miembro públicas de la clase. Las funciones de utilería no están concebidas para ser utilizadas por los clientes de una clase.

La clase `SalesPerson` tiene un arreglo de 12 cifras de ventas mensuales inicializadas a cero por el constructor y definidas mediante la función `setSales` en valores proporcionados por el usuario. La función miembro pública `printAnnualSales` imprime las ventas totales correspondientes a los 12 últimos meses. La función de utilería `total AnnualSales` totaliza las 12 cifras de ventas mensuales, para uso y beneficio de `printAnnualSales`. La función miembro `printAnnualSales` edita las cifras de ventas y las coloca en formato de moneda. En el capítulo 21 se explica con detalle cada una de las capacidades de formato de C++ utilizadas aquí. Por ahora, sólo damos una breve explicación. La llamada

```
setprecision(2)
```

indica que serán impresos dos dígitos de "precisión" a la derecha del punto decimal, exactamente como necesitamos para cantidades en moneda, tales como 23.47. Esta llamada se conoce como

un "manipulador de flujo parametrizado". Un programa que utilice este tipo de llamadas, también deberá contener la directiva de preprocesador

```
#include <iomanip.h>
```

La línea

```
setiosflags(ios::fixed | ios::showpoint);
```

hace que se muestre la cifra de ventas en el formato conocido como de punto fijo (a diferencia de la notación científica). La opción **showpoint** obliga a que aparezca el punto decimal y los ceros después del punto, aún si el número es una cantidad redonda de dólares, como 47.00. En C++, de no haberse definido la opción **showpoint**, un número como éste se imprimiría solo como 47.

16.10 Cómo inicializar objetos de clase: constructores

Después de que los objetos son creados, sus miembros pueden ser inicializados mediante funciones *constructor*. Un constructor es una función miembro de clase con el mismo nombre que la clase. El programador proporciona el constructor el cual entonces, cada vez que se crea un objeto de dicha clase, es invocado automáticamente. *Los miembros de datos de una clase no pueden ser inicializados en la definición de clase.* Más bien, los miembros de datos deben ser inicializados en un constructor de la clase o sus valores definidos más adelante después de que el objeto haya sido creado. Los constructores no pueden especificar tipos de regreso y valores de regreso. Los constructores pueden ser sujetos de homonimia, para permitir una variedad de maneras de inicializar objetos de una clase.

Error común de programación 16.5

Intentar inicializar explícitamente un miembro de datos de una clase dentro de la definición de clase.

```
// SALESP.H
// SalesPerson class definition
// Member functions defined in SALESP.CPP

#ifndef SALESP_H
#define SALESP_H

class SalesPerson {
public:
   SalesPerson();              // constructor
   void setSales();            // user supplies sales figures
   void printAnnualSales();

private:
   double sales[13];           // 12 monthly sales figures
   double totalAnnualSales(); // utility function
};

#endif
```

Fig. 16.7 Cómo utilizar una función de utilería (parte 1 de 3).


```
// SALESP.CPP
// Member functions for class SalesPerson
#include <iostream.h>
#include <iomanip.h>
#include "salesp.h"

// Constructor function initializes array
SalesPerson::SalesPerson()
{
   for (int i = 0; i <= 12; i++)
      sales[i] = 0.0;
}

// Function to set the 12 monthly sales figures
void SalesPerson::setSales()
{
   for (int i = 1; i <= 12; i++) {
      cout << "Enter sales amount for month "
           << i << ": ";
      cin >> sales[i];
   }
}

// Private utility function to total annual sales
double SalesPerson::totalAnnualSales()
{
   double total = 0.0;

   for (int i = 1; i <= 12; i++)
      total += sales[i];

   return total;
}

// Print the total annual sales
void SalesPerson::printAnnualSales()
{
   cout << setprecision(2)
        << setiosflags(ios::fixed | ios::showpoint)
        << "\nThe total annual sales are: $"
        << totalAnnualSales() << endl;
}
```

Fig. 16.7 Cómo utilizar una función de utilería (parte 2 de 3).

Error común de programación 16.6

Intentar declarar un tipo de regreso para un constructor, y/o intentar regresar un valor proveniente de un constructor.

Práctica sana de programación 16.8

Cuando sea apropiado (casi siempre), proporcione un constructor para asegurarse que todo objeto es inicializado correctamente, con valores significativos.

```
// FIG16_7.CPP
// Demonstrating a utility function
// Compile with SALESP.CPP
#include "salesp.h"

main()
{
   SalesPerson s;          // create SalesPerson object s

   s.setSales();
   s.printAnnualSales();

   return 0;
}
```

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

Total annual sales: $60120.59
```

Fig. 16.7 Cómo utilizar una función de utilería (parte 3 de 3).

Práctica sana de programación 16.9

Cada función miembro (y amigo) que modifique los miembros de datos privados de un objeto deberán asegurarse que los datos se conserven en un estado consistente.

Cuando se declara un objeto de una clase, se pueden dar *inicializadores* en paréntesis a la derecha del nombre del objeto y antes del punto y coma. Estos inicializadores son pasados como argumentos al constructor de la clase. Veremos pronto varios ejemplos de estas *llamadas de constructor.*

16.11 Cómo utilizar argumentos por omisión con los constructores

El constructor correspondiente a **time1.cpp** (figura 16.5) inicializó a **0 hour, minute** y **second** (es decir, a las 12 de media noche en hora militar). Los constructores pueden contener argumentos por omisión. La figura 16.8 vuelve a definir la función constructor **Time** para incluir para cada variable argumentos por omisión iguales a cero. Al dar al constructor argumentos por omisión, aun si a una llamada de constructor no se dan valores, el objeto está garantizado de estar en un estado consistente, debido a los argumentos por omisión. Un constructor suministrado

por el programador, y que utiliza sus argumentos por omisión, también se llama un constructor por omisión (puede ser invocado sin argumentos). El constructor utiliza el mismo tipo de código de validación que **setTime**, para asegurarse que el valor suministrado correspondiente a **hour** está en el rango de 0 a 23, y que los valores correspondientes a **minute** y **second** son cada uno de ellos dentro del rango 0 a 59. Si un valor está fuera de rango, se define a cero (este es un ejemplo de cómo asegurar que un miembro de datos se conserva en un estado consistente). El constructor podría llamar directamente a **setTime**, pero esto crearía la sobrecarga de una llamada de función adicional. El programa de la figura 16.8 inicializa cinco objetos **Time** uno con los tres argumentos en sus valores por omisión en la llamada de constructor, uno con un argumento definido, uno con dos definidos, uno con tres definidos y uno con tres argumentos inválidos especificados. Aparece mostrado el contenido de cada uno de los miembros de datos del objeto, después de la ejemplificación y de la inicialización.

Si para una clase no se define ningún constructor, el compilador creará un constructor por omisión. Dicho constructor no ejecutará ninguna inicialización, por lo que no se garantizará que esté en un estado consistente.

Práctica sana de programación 16.10

Incluya siempre un constructor que ejecute la incialización adecuada para su clase.

16.12 Cómo utilizar destructores

Un *destructor* es una función miembro especial de una clase. El nombre del destructor para una clase es la *tilde (~)* seguida por el nombre de la clase. Esta regla convencional de identificación tiene un atractivo intuitivo, porque el operador de tilde es el operador de complemento a nivel de bits, y , en cierto sentido, el destructor es el complemento del constructor.

```
// TIME2.H
// Declaration of the Time class.
// Member functions defined in TIME2.CPP

// prevent multiple inclusions of header file
#ifndef TIME2_H
#define TIME2_H

class Time {
public:
   Time(int = 0, int = 0, int = 0);  // default constructor
   void setTime(int, int, int);
   void printMilitary();
   void printStandard();
private:
   int hour;
   int minute;
   int second;
};

#endif
```

Fig. 16.8 Cómo usar un constructor mediante argumentos por omisión (parte 1 de 4).

```
// TIME2.CPP
// Member function definitions for Time class.
#include "time2.h"
#include <iostream.h>

// Constructor function to initialize private data.
// Default values are 0 (see class definition).
Time::Time(int hr, int min, int sec)
{
   hour = (hr >= 0 && hr < 24) ? hr : 0;
   minute = (min >= 0 && min < 60) ? min : 0;
   second = (sec >= 0 && sec < 60) ? sec : 0;
}

// Set values of hour, minute, and second.
// Invalid values are set to 0.
void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}

// Display time in military format: HH:MM:SS
void Time::printMilitary()
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}

// Display time in standard format: HH:MM:SS AM (or PM)
void Time::printStandard()
{
   cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
        << ":" << (minute < 10 ? "0" : "") << minute
        << ":" << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}
```

Fig. 16.8 Cómo usar un constructor con argumentos por omisión (parte 2 de 4).

Un destructor de clase es llamado automáticamente cuando un objeto de una clase se sale de alcance. De hecho el destructor mismo en realidad no destruye el objeto, más bien ejecuta *trabajos de terminación*, antes de que el sistema recupere el espacio de memoria del objeto con el fin de que pueda ser utilizado para almacenar nuevos objetos.

Un destructor no recibe ningún parámetro ni regresa ningún valor. Una clase sólo puede tener un destructor —la homonimia de destructores no está permitida.

Error común de programación 16.7

Intentar pasar argumentos a un destructor, regresar valores de un destructor, o hacer la homonimia de un destructor.


```
// FIG16_8.CPP
// Demonstrating a default constructor
// function for class Time.
#include <iostream.h>
#include "time2.h"

main()
{
   Time t1, t2(2), t3(21, 34), t4(12, 25, 42),
        t5(27, 74, 99);

   cout << "Constructed with:\n"
        << "all arguments defaulted:\n   ";
   t1.printMilitary();
   cout << "\n   ";
   t1.printStandard();

   cout << "\nhour specified; minute and second defaulted:\n   ";
   t2.printMilitary();
   cout << "\n   ";
   t2.printStandard();

   cout << "\nhour and minute specified; second defaulted:\n   ";
   t3.printMilitary();
   cout << "\n   ";
   t3.printStandard();

   cout << "\nhour, minute, and second specified:\n   ";
   t4.printMilitary();
   cout << "\n   ";
   t4.printStandard();

   cout << "\nall invalid values specified:\n   ";
   t5.printMilitary();
   cout << "\n   ";
   t5.printStandard();
   cout << endl;

   return 0;
}
```

Fig. 16.8 Cómo usar un constructor con argumentos por omisión (parte 3 de 4).

Note que no hemos proporcionado destructores para las clases que hasta ahora hemos presentado. De hecho, con clases sencillas rara vez se utilizan destructores. En el capítulo 18, veremos que los destructores son apropiados para aquellas clases cuyos objetos contienen almacenamiento dinámicamente asignado (por ejemplo, para arreglos y cadenas).

16.13 Cuándo son llamados los destructores y los constructores

Por lo regular, los constructores y los destructores son llamados de forma automática. El orden en el cual son hechas estas llamadas de función, depende del orden en el cual los objetos entran y salen de alcance. En general, las llamadas de destructor se efectúan en orden inverso a las

```
Constructed with:
all arguments defaulted:
      00:00:00
      12:00:00 AM
hour specified; minute and second defaulted:
      02:00:00
      2:00:00 AM
hour and minute specified; second defaulted:
      21:34:00
      9:34:00 PM
hour, minute, and second specified:
      12:25:42
      12:25:42 PM
all invalid values specified:
      00:00:00
      12:00:00 AM
```

Fig. 16.8 Cómo usar un constructor con argumentos por omisión (parte 4 de 4).

llamadas de constructor. Sin embargo, la duración de almacenamiento (persistencia) de los objetos puede modificar el orden en el cual los destructores son llamados.

Se llaman a los constructores para objetos declarados en alcance global al principio de la ejecución del programa. Los destructores correspondientes son llamados a la terminación del programa.

Los constructores son llamados para objetos locales automáticos, cuando estos objetos son declarados. Los destructores correspondientes son llamados cuando los objetos salen de alcance (es decir, cuando salen del bloque en el cual están declarados). Note que tanto los constructores como los destructores, correspondientes a los objetos locales automáticos, podrían ser llamados muchas veces, conforme los objetos entren y salgan de alcance.

Para los objetos locales estáticos los constructores son llamados una vez cuando estos objetos son declarados. Los destructores correspondientes son llamados a la terminación del programa.

El programa de la figura 16.9 demuestra el orden en el que se llaman los constructores y los destructores para objetos del tipo **CreateAndDestroy** en varios alcances. El programa declara **first** en alcance global. Su constructor es llamado conforme el programa inicia ejecución y su destructor es llamado a la terminación del programa, después de que todos los demás objetos han sido destruidos.

La función **main** declara tres objetos. Los objetos **second** y **fourth** son objetos automáticos locales, y el objeto **third** es un objeto local estático. Los constructores correspondientes a cada uno de estos objetos son llamados cuando es declarado cada uno de estos objetos. Cuando se llega al final de **main** los destructores para los objetos **fourth** y **second** son llamados en ese orden. Dado que el objeto **third** es estático, existirá desde el momento en que se declara, hasta la terminación del programa. El destructor para el objeto **third** es llamado antes del destructor para **first**, pero después de que hayan sido destruidos todos los demás objetos.

La función **create** declara tres objetos. Los objetos **fifth** y **seventh** son objetos automáticos locales, y el objeto **sixth** es un objeto local estático. Cuando se llega al final de **create** los destructores correspondientes a los objetos **seventh** y **fifth** son llamados en ese orden. Debido a que el objeto **sixth** es estático, existe desde el punto en el cual es declarado,


```
// CREATE.H
// Definition of class CreateAndDestroy.
// Member functions defined in CREATE.CPP.

#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy {
public:
   CreateAndDestroy(int);  // constructor
   ~CreateAndDestroy();    // destructor
private:
   int data;
};

#endif
```

Fig. 16.9 Cómo demostrar el orden en el cual son llamados los constructores y los destructores (parte 1 de 3).

```
// CREATE.CPP
// Member function definitions for class CreateAndDestroy
#include <iostream.h>
#include "create.h"

CreateAndDestroy::CreateAndDestroy(int value)
{
   data = value;
   cout << "Object " << data << "   constructor";
}

CreateAndDestroy::~CreateAndDestroy()
   { cout << "Object " << data << "   destructor " << endl; }
```

Fig. 16.9 Cómo demostrar el orden en el cual son llamados los constructores y los destructores (parte 2 de 3).

hasta la terminación del programa. El destructor para el objeto **sixth** es llamado antes de los destructores correspondientes a **third** y a **first**, pero después de que todos los otros objetos hayan sido destruidos.

16.14 Cómo utilizar miembros de datos y funciones miembro

Los miembros de datos privados pueden ser manipulados sólo por funciones miembro y los amigos de la clase. Una manipulación típica pudiera ser el ajuste del saldo bancario de un cliente (por ejemplo, un miembro de dato privado de la clase **BankAccount**), hecho por una función miembro **computeInterest**.

A menudo las clases proporcionan funciones miembro públicas para permitir a los clientes de la clase definir (es decir, escribir) u obtener (es decir, leer) los valores de los miembros de datos privados. Estas funciones no necesitan ser en específico funciones "set" y "get", pero a menudo lo son. En específico, una función miembro, que define el miembro de datos **interestRate**

```
// FIG16_9.CPP
// Demonstrating the order in which constructors and
// destructors are called.
#include <iostream.h>
#include "create.h"

void create(void);    // prototype

CreateAndDestroy first(1);  // global object

main()
{
   cout << "   (global created before main)\n";

   CreateAndDestroy second(2);        // local object
   cout << "   (local automatic in main)\n";

   static CreateAndDestroy third(3);  // local object
   cout << "   (local static in main)\n";

   create();  // call function to create objects

   CreateAndDestroy fourth(4);        // local object
   cout << "   (local automatic in main)\n";
   return 0;
}

// Function to create objects
void create(void)
{
   CreateAndDestroy fifth(5);
   cout << "   (local automatic in create)\n";

   static CreateAndDestroy sixth(6);
   cout << "   (local static in create)\n";

   CreateAndDestroy seventh(7);
   cout << "   (local automatic in create)\n";
```

```
Object 1 constructor (global created before main)
Object 2 constructor (local automatic in main)
Object 3 constructor (local static in main)
Object 5 constructor (local automatic in create)
Object 6 constructor (local static in create)
Object 7 constructor (local automatic in create)
Object 7 destructor
Object 5 destructor
Object 4 constructor (local automatic in main)
Object 4 destructor
Object 2 destructor
Object 6 destructor
Object 3 destructor
Object 1 destructor
```

Fig. 16.9 Córno demostrar el orden en que se llaman constructores y destructores (parte 3 de 3).

típicamente se llamaría **setInterestRate**, y una función miembro que obtiene el **interestRate**, típicamente sería llamada **getInterestRate**. Las funciones get también por lo común son conocidas como funciones "de consulta".

Parecería que proporcionar capacidades tanto de definir como de obtener resultaría en esencial lo mismo que el hacer públicos los miembros de datos. Esto es otra vez otra sutileza de C++, que hace que este lenguaje sea tan atractivo para la ingeniería de software. Si un miembro de datos es público, entonces el miembro de datos puede ser leído o escrito a voluntad por cualquier función del programa. Si un miembro de datos es privado, una función "get" pública parecería que podría permitir a otras funciones leer los datos a voluntad, pero la función get controlaría el formato y la exhibición de los datos. Una función pública "set" podría —y muy probablemente haría— escrutinizar con cuidado cualquier intento de modificación del valor del miembro de dato. Esto garantizaría que el nuevo valor es apropiado para ese elemento de dato. Por ejemplo, serían rechazados intentos de definir el día del mes a 37, intentos de definir el peso de una persona a un valor negativo, intentos de definir una cantidad numérica a un valor alfabético, intentos de definir una calificación de un examen como 185 (cuando el rango correcto es de 0 a 100), y así en lo sucesivo.

Observación de ingeniería de software 16.15

Hacer los miembros de datos privados y controlando su acceso, especialmente el acceso de escritura, a aquellos miembros de datos vía funciones miembro, ayuda a asegurar la integridad de los datos.

Los beneficios de la integridad de los datos no son automáticos sólo porque son hechos privados los miembros de datos, el programador debe de proveer la verificación de validez. C++, sin embargo, sí proporciona una estructura en la cual los programadores pueden diseñar mejores programas de una forma conveniente.

Práctica sana de programación 16.11

Las funciones miembro que definen los valores de los datos privados, deberán verificar que los nuevos valores propuestos son correctos; si no lo son, las funciones set deben colocar los miembros de datos privados en un estado consistente apropiado.

En la figura 16.10 se extiende nuestra clase **Time** a fin de incluir las funciones get y set para los miembros de datos privados **hour**, **minute** y **second**. Las funciones set controlan de forma estricta la definición de los miembros de datos. Cualquier intento de definir cualquier miembro de datos a un valor incorrecto, hará que el miembro de dato se especifique a cero (y dejando por lo tanto dicho miembro de dato en un estado consistente). Cada función get sólo regresa el valor apropiado del miembro de dato. El programa primero utiliza las funciones set para definir los miembros de dato privados del objeto **t** de **Time** a valores válidos, y a continuación utiliza las funciones get para recuperar dichos valores para su extracción. A continuación las funciones set intentan definir los miembros **hour** y **second** a valores inválidos y el miembro **minute** a un valor válido, y a continuación las funciones get recuperan los valores para su extracción. La salida confirma que los valores inválidos hacen que los miembros de datos se ajusten a cero. Por último, el programa especifica la hora a **11:58:00** e incrementa el valor de minutos en 3 con una llamada a la función **incrementMinutes**. La función **incrementMinutes** es una función no miembro, que utiliza las funciones get y set para incrementar de forma correcta el miembro **minute**. Aunque esto funciona, al emitir varias llamadas de función incurre en sobrecarga de rendimiento. En el siguiente capítulo analizamos el concepto de las funciones amigas, como un medio de eliminar esta sobrecarga de rendimiento.

```
// TIME3.H
// Declaration of the Time class.
// Member functions defined in TIME3.CPP

// prevent multiple inclusions of header file
#ifndef TIME3_H
#define TIME3_H

class Time {
public:
   Time(int = 0, int = 0, int = 0);  // constructor

   // set functions
   void setTime(int, int, int);  // set hour, minute, second
   void setHour(int);      // set hour
   void setMinute(int);    // set minute
   void setSecond(int);    // set second

   // get functions
   int getHour();          // return hour
   int getMinute();        // return minute
   int getSecond();        // return second

   void printMilitary();   // output military time
   void printStandard();   // output standard time
private:
   int hour;               // 0 - 23
   int minute;             // 0 - 59
   int second;             // 0 - 59
};

#endif
```

Fig. 16.10 Declaración de la clase **Time** (parte 1 de 5).

La disponibilidad de las funciones set en una clase, proporciona cierto rango de flexibilidad en el diseño de constructores.

Error común de programación 16.8

Un constructor puede llamar a otras funciones miembro de la clase como son las funciones set o get, pero dado que el constructor está inicializando el objeto, los miembros de datos pudieran no estar aún en un estado consistente. Puede causar errores utilizar miembros de datos antes de que hayan sido inicializados de manera adecuada.

Desde un punto de vista de ingeniería de software, las funciones set son ciertamente de importancia, porque pueden llevar a cabo verificación de validez. Las funciones set y get tienen otra ventaja importante de ingeniería de software.

Observación de ingeniería de software 16.16

Tener acceso a datos privados mediante las funciones miembro set y get no sólo protege los miembros de datos contra la recepción de valores inválidos, sino también aísla a los clientes de la clase de la representación de los miembros de datos. Entonces, si por alguna razón cambia la representación de los datos (típicamente para reducir la cantidad de almacenamiento requerido o para mejorar el rendimiento), sólo las funciones miembro necesitan modificarse —no es necesario que los clientes cambien, en tanto la interfaz proporcionada por las funciones miembro se conserve igual. Los clientes pueden, sin embargo, requerir ser recompilados.


```
// TIME3.CPP
// Member function definitions for Time class.

#include "time3.h"
#include <iostream.h>

// Constructor function to initialize private data.
// Default values are 0 (see class definition).
Time::Time(int hr, int min, int sec)
{
   hour = (hr >= 0 && hr < 24) ? hr : 0;
   minute = (min >= 0 && min < 60) ? min : 0;
   second = (sec >= 0 && sec < 60) ? sec : 0;
}

// Set the values of hour, minute, and second.
void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}

// Set the hour value
void Time::setHour(int h)
   { hour = (h >= 0 && h < 24) ? h : 0; }

// Set the minute value
void Time::setMinute(int m)
   { minute = (m >= 0 && m < 60) ? m : 0; }

// Set the second value
void Time::setSecond(int s)
   { second = (s >= 0 && s < 60) ? s : 0; }

// Get the hour value
int Time::getHour() { return hour; }

// Get the minute value
int Time::getMinute() { return minute; }

// Get the second value
int Time::getSecond() { return second; }

// Display military format time: HH:MM:SS
void Time::printMilitary()
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}
```

Fig. 16.10 Definiciones de funciones miembro para la clase **Time** (parte 2 de 5).

```
// Display standard format time: HH:MM:SS AM (or PM)
void Time::printStandard()
{
   cout << ((hour == 12) ? 12 : hour % 12) << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}
```

Fig. 16.10 Definiciones de funciones miembro para la clase **Time** (parte 3 de 5).

```
// FIG16_10.CPP
// Demonstrating the Time class set and get functions
#include <iostream.h>
#include "time3.h"

main()
{
   Time t;
   void incrementMinutes(Time &, const int);

   t.setHour(17);
   t.setMinute(34);
   t.setSecond(25);

   cout << "Result of setting all valid values:\n  Hour: "
        << t.getHour()
        << "  Minute: " << t.getMinute()
        << "  Second: " << t.getSecond() << "\n\n";

   t.setHour(234);       // invalid hour set to 0
   t.setMinute(43);
   t.setSecond(6373);    // invalid second set to 0

   cout << "Result of attempting to set invalid hour and"
        << " second:\n  Hour: " << t.getHour()
        << "  Minute: " << t.getMinute()
        << "  Second: " << t.getSecond() << "\n\n";

   t.setTime(11, 58, 0);
   incrementMinutes(t, 3);

   return 0;
}
```

Fig. 16.10 Cómo utilizar las funciones set y get (parte 4 de 5).

16.15 Una trampa sutil: cómo regresar una referencia a un miembro de datos privado

Una referencia a un objeto es un seudónimo del objeto mismo y por lo tanto puede ser utilizado del lado izquierdo de un enunciado de asignación. En este contexto, la referencia es un valor a la izquierda (*lvalue*) es muy aceptable, que puede recibir un valor. Una forma de aprovechar esta


```
void incrementMinutes(Time &tt, const int count)
{
   cout << "Incrementing minute " << count
        << " times:\nStart time: ";
   tt.printStandard();

   for (int i = 1; i <= count; i++) {
      tt.setMinute((tt.getMinute() + 1) % 60);

      if (tt.getMinute() == 0)
         tt.setHour((tt.getHour() + 1) % 24);

      cout << "\nminute + 1: ";
      tt.printStandard();
   }

   cout << endl;
}
```

```
Result of setting all valid values:
  Hour: 17  Minute: 34  Second: 25

Result of attempting to set invalid hour and second:
  Hour: 0  Minut: 43  Second: 0

Incrementing minute 3 times:
Start time: 11:58:00 AM
minute + 1: 11:59:00 AM
minute + 1: 12:00:00 PM
minute + 1: 12:01:00 PM
```

Fig. 16.10 Cómo utilizar funciones set y get (parte 5 de 5).

capacidad (¡desafortunadamente!) es hacer que una función miembro pública de una clase regrese una referencia a un miembro de datos privado de la misma clase.

En la figura 16.11 se utiliza una versión simplificada de la clase **Time**, para demostrar el regreso de una referencia a un miembro de datos privado. Tal regreso, de hecho efectúa una llamada a la función **badSetHour** ¡un seudónimo del miembro de datos privado **hour**! La llamada de función puede ser utilizada de cualquiera de las formas en que un miembro de datos privado pueda ser utilizado, ¡incluye un valor a la izquierda (*lvalue*) en un enunciado de asignación!

Práctica sana de programación 16.12

Nunca haga que una función miembro pública regrese una referencia no **const** *(o un apuntador) a un miembro de datos privado. Regresar una referencia como ésta viola el encapsulado de la clase.*

El programa empieza declarando el objeto **t** de **Time** y la referencia **hourRef** que está asignada a la referencia regresada por la llamada **t.badSetHour(20)**. El programa muestra el valor del seudónimo **hourRef**. A continuación, se utiliza el seudónimo para definir el valor de **hour** a 30 (un valor inválido) y se vuelve a mostrar el valor. Finalmente, se utiliza la llamada de función misma, como un valor a la izquierda (*lvalue*) y se le asigna el valor 74 (otro valor inválido), y este valor es mostrado.

```
// TIME4.H
// Declaration of the Time class.
// Member functions defined in TIME4.CPP

// prevent multiple inclusions of header file
#ifndef TIME4_H
#define TIME4_H

class Time {
public:
   Time(int = 0, int = 0, int = 0);
   void setTime(int, int, int);
   int getHour();
   int &badSetHour(int);  // DANGEROUS reference return
private:
   int hour;
   int minute;
   int second;
};

#endif
```

Fig. 16.11 Cómo regresar una referencia a un miembro de datos privado (parte 1 de 3).

```
// TIME4.CPP
// Member function definitions for Time class.
#include "time4.h"
#include <iostream.h>

// Constructor function to initialize private data.
// Calls member function setTime to set variables.
// Default values are 0 (see class definition).
Time::Time(int hr, int min, int sec)
   { setTime(hr, min, sec); }

// Set the values of hour, minute, and second.
void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}

// Get the hour value
int Time::getHour() { return hour; }

// BAD PROGRAMMING PRACTICE:
// Returning a reference to a private data member.
int &Time::badSetHour(int hh)
{
   hour = (hh >= 0 && hh < 24) ? hh : 0;

   return hour;  // DANGEROUS reference return
}
```

Fig. 16.11 Cómo regresar una referencia a un miembro de datos privado (parte 2 de 3).


```
// FIG16_11.CPP
// Demonstrating a public member function that
// returns a reference to a private data member.
// Time class has been trimmed for this example.

#include <iostream.h>
#include "time4.h"

main()
{
   Time t;
   int &hourRef = t.badSetHour(20);

   cout << "Hour before modification: "
        << hourRef << '\n';
   hourRef = 30;  // modification with invalid value
   cout << "Hour after modification: "
        << t.getHour() << '\n';

   // Dangerous: Function call that returns
   // a reference can be used as an lvalue.
   t.badSetHour(12) = 74;
   cout << "\n*********************************\n"
        << "BAD PROGRAMMING PRACTICE!!!!!!!!!\n"
        << "badSetHour as an lvalue, Hour: "
        << t.getHour()
        << "\n*********************************\n";

   return 0;
}
```

```
Hour before modification: 20
Hour after modification: 30

*********************************
BAD PROGRAMMING PRACTICE!!!!!!!!!
badSetHour as an lvalue, Hour: 74
*********************************
```

Fig. 16.11 Cómo regresar una referencia a un miembro de datos privado (parte 3 de 3).

16.16 Asignación por omisión en copia a nivel de miembro

El operador de asignación (=) es utilizado para asignar un objeto a otro objeto del mismo tipo. Dicha asignación se lleva normalmente a cabo mediante la *copia a nivel de miembro* —cada miembro de un objeto es copiado en forma individual al mismo miembro dentro de otro objeto (vea la figura 16.12). (Nota: la copia a nivel de miembro puede causar problemas serios, cuando sea utilizada con una clase cuyos miembros de datos contengan almacenamiento dinámicamente asignado; en el capítulo 18, "Homonimia de operadores", analizaremos estos problemas y mostraremos cómo resolverlos.)

```
// FIG16_12.CPP
// Demonstrating that class objects can be assigned
// to each other using default memberwise copy
#include <iostream.h>

// Simple Date class
class Date {
public:
   Date(int = 1, int = 1, int = 1990);  // default constructor
   void print();
private:
   int month;
   int day;
   int year;
};

// Simple Date constructor with no range checking
Date::Date(int m, int d, int y)
{
   month = m;
   day = d;
   year = y;
}

// Print the Date in the form mm-dd-yyyy
void Date::print()
{ cout << month << '-' << day << '-' << year; }

main()
{
   Date d1(7, 4, 1993), d2;  // d2 defaults to 1/1/90

   cout << "d1 = ";
   d1.print();
   cout << "\nd2 = ";
   d2.print();

   d2 = d1;     // assignment by default memberwise copy
   cout << "\n\nAfter default memberwise copy, d2 = ";
   d2.print();
   cout << endl;

   return 0;
}
```

```
d1 = 7-4-1993
d2 = 1-1-1990

After default memberwise copy, d2 = 7-4-1993
```

Fig. 16.12 Cómo asignar un objeto a otro mediante la copia a nivel de miembro por omisión.

Los objetos pueden ser pasados como argumentos de función y pueden ser regresados de las funciones. Este pasar y regresar se ejecuta por omisión —en llamada por valor se pasa o se regresa una copia del objeto (presentamos varios ejemplos en el capítulo 18, "Homonimia de operadores").

Sugerencia de rendimiento 16.4

Pasar un objeto en llamada por valor es bueno desde un punto de vista de seguridad, porque la función llamada no tiene acceso al objeto original, pero cuando se tiene que hacer una copia de un objeto grande, la llamada por valor puede degradar el rendimiento. Un objeto puede ser pasado en llamada por referencia, pasando ya sea un apuntador o una referencia al objeto. La llamada por referencia ofrece un buen rendimiento, pero es más débil desde un punto de vista de seguridad, debido a que la función llamada tiene acceso al objeto original. Una alternativa segura sería una referencia en llamada por `const`.

16.17 Reutilización del software

Las personas que escriben programas orientados a objetos se concentran en la puesta en práctica de clases útiles. Existe una enorme oportunidad de capturar y catalogar clases, de tal forma que sean accesibles a grandes segmentos de la comunidad de programadores. Existen en todo el mundo muchas *bibliotecas de clases*, y otras están siendo desarrolladas. Se hacen esfuerzos para que dichas bibliotecas sean muy accesibles. El software podrá entonces ser construido a partir de componentes existentes, bien definidos, cuidadosamente probados, bien documentados, portátiles y ampliamente disponibles. Este tipo de *reutilización de software* puede acelerar el desarrollo de software poderoso y de alta calidad. El desarrollo rápido de aplicaciones *(RAD por Rapid applications development)* se convierte en posible.

Antes de que se pueda alcanzar el potencial total de la reutilización del software, deben ser resueltos, sin embargo, problemas significativos. Necesitamos procesos para catalogar, procedimientos para licenciar, mecanismos de protección para asegurar que las copias maestras de clases no están corrompidas, procedimientos de descripción, de tal forma que los diseñadores de sistemas nuevos puedan determinar si los objetos existentes llenan sus necesidades, mecanismos de búsqueda para determinar qué clases están disponibles y que tan justamente cumplen los requisitos del diseñador de software, y así en lo sucesivo. Muchos problemas interesantes de investigación y desarrollo necesitan resolverse. Estos problemas *serán* resueltos, porque es enorme el valor potencial de sus soluciones.

Resumen

- Las estructuras son tipos de datos agregados construidos utilizando objetos de otros tipos.
- La palabra reservada `struct` introduce la definición de estructura. El cuerpo de una estructura queda definido entre llaves ({ y }). La definición de cada estructura debe terminar con un punto y coma.
- Se puede utilizar un nombre de etiqueta de estructura para declarar variables de un tipo de estructura.
- Las definiciones de estructura no reservan espacio en memoria; crean nuevos tipos de datos que son utilizados para declarar variables.
- Se tiene acceso a los miembros de una estructura o de una clase utilizando los operadores de acceso de miembro el operador punto (.) y el operador flecha (->). El operador punto da acceso a un miembro de estructura vía el nombre de variable del objeto o una referencia al objeto. El operador flecha da acceso a un miembro de estructura vía un apuntador al objeto.

- Los inconvenientes para la creación de nuevos tipos de datos utilizando los **`struct`** de tipo C son la posibilidad de tener datos sin inicializar; inicializaciones incorrectas; todos los programas que utilicen **`struct`** del estilo C deben ser transformados si se modifica la puesta en práctica de **`struct`**; y no se da protección para asegurar que los datos se conservan en un estado consistente con los valores de datos apropiados.
- Las clases le permiten al programador hacer modelos de objetos con atributos y comportamientos. Los tipos de clase pueden ser definidos en C++ utilizando las palabras reservadas **`class`** y **`struct`**, pero la palabra reservada **`class`** por lo regular es utilizada para este fin.
- El nombre de clase puede ser utilizado para declarar objetos de dicha clase.
- Las definiciones de clase empiezan con la palabra reservada **`class`**. El cuerpo de la definición de clase está delimitado por llaves (**`{`** y **`}`**). Las definiciones de clase terminan con un punto y coma.
- Cualquier miembro de datos o función miembro declarada después de **`public:`** en una clase es visible a cualquier función que tenga acceso a un objeto de dicha clase.
- Cualquier miembro de dato o función miembro declarada después de **`private:`** es sólo visible a amigos y otros miembros de dicha clase.
- Los especificadores de acceso de miembro siempre terminan con dos puntos (`:`) y pueden aparecer múltiples veces en una definición de clase.
- Los datos privados no son accesibles desde fuera de la clase.
- La puesta en práctica de una clase se dice que está oculta de sus clientes, es decir, está "encapsulada".
- Un constructor es una función miembro especial, con el mismo nombre de la clase, y se utiliza para inicializar los miembros de un objeto de clase. Los constructores son llamados cuando se produce o se ejemplifica un objeto de su clase.
- La función con el mismo nombre que la clase, pero antecedida con el carácter tilde (~), se llama un destructor.
- El conjunto de funciones miembro públicas de una clase se conoce como la interfaz de la clase o la interfaz pública.
- Cuando una función miembro se define por fuera de la definición de la clase, el nombre de función es antecedido por el nombre de la clase y por el operador de resolución de alcance binario (`::`).
- Las funciones miembro definidas utilizando el operador de resolución de alcance por fuera de una definición de clase, están dentro del alcance de la clase.
- Las funciones miembro definidas en una definición de clase están en línea en forma automática. El compilador se reserva el derecho de no considerar en línea a cualquier función.
- Llamar funciones miembro es más conciso que la llamada de funciones en la programación procedural, porque las funciones miembro asumen que los nombres no declarados son miembros de la clase en la cual la función miembro está definida.
- Dentro del alcance de una clase, los miembros de la clase pueden ser referenciados simplemente con sus nombres. Fuera del alcance de una clase, los miembros de la clase se referencían ya sea a través de un nombre de objeto, una referencia a un objeto o un apuntador a un objeto.
- Los operadores de selección de miembros `.` y `->` son utilizados para tener acceso a los miembros de clase.

- Un principio fundamental de buena ingeniería de software es separar la interfaz de la puesta en práctica para facilitar la capacidad de modificación del programa.
- Las definiciones de clase por lo regular están colocadas en los archivos de cabecera y las definiciones de función miembro están normalmente colocadas en los archivos de código fuente, con el mismo nombre base.
- El modo de acceso por omisión para las clases es **`private:`** de tal forma que todos los miembros después de un encabezado de clase y antes de la primera etiqueta se consideran como privados.
- Los miembros de clase públicos presentan una vista de los servicios que proporciona la clase.
- El acceso a los datos privados de una clase puede ser controlado con cuidado mediante el uso de las funciones miembro conocidas como funciones de acceso. Si una clase desea permitir que los clientes lean datos privados, la clase puede proporcionar una función "get". Para permitir que los clientes modifiquen datos privados, la clase puede proveer una función "set".
- Los miembros de datos de una clase normalmente se hacen privados y las funciones miembro de una clase a menudo se hacen públicos. Algunas funciones miembro se conservan privadas y sirven como funciones de utilería a otras funciones de la clase.
- Los miembros de datos de una clase no pueden ser inicializados en una definición de clase. Deben ser inicializados en un constructor, o sus valores pueden ser definidos más adelante, después de que su objeto haya sido creado.
- Los constructores pueden tener homónimos.
- Una vez que un objeto de clase esté correctamente inicializado, todas las funciones miembro que manipulan el objeto deberán asegurarse que el objeto se conserva en un estado consistente.
- Cuando se declara un objeto de una clase, pueden ser proporcionados los inicializadores. Estos inicializadores se pasan al constructor de la clase.
- Los constructores pueden especificar argumentos por omisión.
- Los constructores pudieran no especificar tipos de regreso, ni pudieran intentar regresar valores.
- Si no se define un constructor para una clase, el compilador crea un constructor por omisión. Un constructor por omisión suministrado por el compilador no lleva a cabo ninguna inicialización, por lo cual cuando se crea el objeto, no está garantizado que esté en un estado consistente.
- Un destructor de una clase es automáticamente llamado, cuando un objeto de una clase sale de alcance. El destructor mismo, de hecho no destruye el objeto, pero sí ejecuta trabajos de terminación, antes de que el sistema recupere el almacenamiento del objeto.
- Los destructores no reciben parámetros y no regresan valores. Una clase sólo puede tener un destructor.
- El operador de asignación (=) es utilizado para asignar un objeto a otro objeto del mismo tipo. Dicha asignación se ejecuta normalmente mediante copia a nivel de miembro —cada miembro de un objeto se copia de manera individual al mismo miembro en otro objeto.

Terminología

operador de dirección **`&`**
operador de referencia **`&`**
operador de resolución de alcance **`::`**
tipo de datos abstractos (ADT)

función de acceso
operador de asignación (=)
atributo
comportamiento
`class`
operador selector de miembro de clase (.)
declaración de clase
definición de clase
alcance de clase
nombre de etiqueta de clase
cliente de una clase
estado consistente para un miembro de datos
constructor
abstracción de datos
ocultamiento de datos
miembro de datos
tipo de datos
constructor por omisión
destructor
objetos dinámicos
encapsulado
alcance de entrada
extensibilidad
alcance de archivo
declaración hacia adelante de una clase
función get
objeto global
archivo de cabecera
puesta en práctica de una clase
ocultamiento de información
inicializar un objeto de clase
función miembro en línea
ejemplos de una clase
ejemplificación de un objeto de una clase
interfaz a una clase
salir de alcance
control de acceso de miembro
especificador de acceso de miembro
`función miembro`
inicializador de miembro
copia a nivel de miembro
mensaje
método
función no miembro
objeto local no estático
objeto
programación orientada a objetos (OOP)
función predicada
principio del mínimo privilegio
`private:`
programación procedural
`protected:`
interfaz pública de una clase
`public:`
función de consulta
desarrollo rápido de aplicaciones (RAD)
código reutilizable
operador de resolución de alcance (::)
servicios de una clase
función set
reutilización de software
archivo de código fuente
objeto local estático
trabajos de terminación
tilde (~) en nombre de destructor
tipo definido por usuario
función de utilería.

Errores comunes de programación

16.1 Olvidar el punto y coma al final de una definición de clase.
16.2 Intentar inicializar explícitamente un miembro de datos de una clase.
16.3 Intentar hacer la homonimia de una función miembro mediante una función no incluida en el alcance de dicha clase.
16.4 Un intento por parte de una función, que no sea un miembro de una clase particular (o un amigo de dicha clase) de obtener acceso a un miembro privado de dicha clase.
16.5 Intentar inicializar de forma explícita un miembro de datos de una clase dentro de la definición de clase.
16.6 Intentar declarar un tipo de regreso para un constructor, y/o intentar regresar un valor proveniente de un constructor.
16.7 Intentar pasar argumentos a un destructor, regresar valores de un destructor, o hacer la homonimia de un destructor.
16.8 Un constructor puede llamar a otras funciones miembro de la clase como son las funciones set o get, pero dado que el constructor está inicializando el objeto, los miembros de datos pudieran no estar aún en un estado consistente. Puede causar errores utilizar miembros de datos antes de que hayan sido adecuadamente inicializados.

Prácticas sanas de programación

16.1 Para mayor claridad y legibilidad, utilice cada especificador de acceso de miembro sólo una vez en una definición de clase. Coloque primero los miembros públicos, donde sean con fácilidad localizables.
16.2 Defina todas, a excepción de las más pequeñas funciones miembro, por afuera de la definición de clase. Esto ayuda a separar la interfaz de la clase de su puesta en práctica.
16.3 Utilice en un programa las directivas de preprocesador `#ifndef`, `#define` y `#endif` para evitar que los archivos de cabecera sean incluidos más de una vez.
16.4 Utilice el nombre del archivo de cabecera, con el punto reemplazado por un subrayado, en las directivas de preprocesador `#ifndef` y `#define` correspondientes a un archivo de cabecera.
16.5 Si en una declaración de clase decide listar primero los miembros privados, utilice en forma explícita la etiqueta `private:` a pesar de que por omisión se supone que `private:` es asumido. Esto mejora la claridad del programa. Nuestra preferencia es listar primero los miembros `public:` de una clase.
16.6 A pesar del hecho que las etiquetas `public:` y `private:` pueden ser repetidas y entremezcladas, liste primero en un grupo todos los miembros públicos de una clase y a continuación liste en otro grupo todos los miembros privados. Esto enfoca la atención del cliente en la interfaz pública de la clase, en vez de en la puesta en práctica de la misma.
16.7 Evita confusión usar los especificadores de acceso de miembro `public:`, `protected:`, y `private:` una sola vez en cada definición de clase.
16.8 Cuando sea apropiado (casi siempre), proporcione un constructor para asegurarse que todo objeto es inicializado de manera correcta, con valores significativos.
16.9 Cada función miembro (y amigo) que modifique los miembros de datos privados de un objeto deberán asegurarse que los datos se conserven en un estado consistente.
16.10 Incluya siempre un constructor que ejecute la incialización adecuada para su clase.
16.11 Las funciones miembro que definen los valores de los datos privados, deberán verificar que los nuevos valores propuestos son correctos; si no lo son, las funciones set deben colocar los miembros de datos privados en un estado consistente apropiado.
16.12 Nunca haga que una función miembro pública regrese una referencia no `const` (o un apuntador) a un miembro de datos privado. Regresar una referencia como ésta viola el encapsulado de la clase.

Sugerencias de rendimiento

16.1 Las estructuras por lo regular pasan en llamada por valor. Para evitar la sobrecarga de copiar una estructura, pase la estructura en llamada por referencia.
16.2 A fin de evitar la sobrecarga de la llamada por valor y aún así obtener el beneficio de que la información original del llamador quede protegida contra modificaciones, pase argumentos de tamaño extenso como referencias `const`.
16.3 Definir una función miembro pequeña dentro de una definición de clase automáticamente coloca la función miembro en línea (si el compilador así lo decide). Esto puede mejorar el rendimiento, pero no promueve una mejor ingeniería de software.
16.4 Pasar un objeto en llamada por valor es bueno desde un punto de vista de seguridad, porque la función llamada no tiene acceso al objeto original, pero cuando se tiene que hacer una copia de un objeto grande, la llamada por valor puede degradar el rendimiento. Un objeto puede ser pasado en llamada por referencia, pasando ya sea un apuntador o una referencia al objeto. La llamada por

referencia ofrece un buen rendimiento, pero es más débil desde un punto de vista de seguridad, debido a que la función llamada tiene acceso al objeto original. Una alternativa segura sería una referencia en llamada por `const`.

Observaciones de ingeniería de software

16.1 Es importante escribir programas que sean comprensibles y fáciles de mantener. Las modificaciones son la regla más que la excepción. Los programadores deberán prever que su código será modificado. Como veremos, las clases facilitan la capacidad de modificación de los programas.

16.2 Los clientes de una clase utilizan la clase sin conocer los detalles internos de cómo está puesta en práctica dicha clase. Si se modifica la puesta en práctica de la clase (para mejorar el rendimiento, por ejemplo) no es necesario modificar los clientes de la clase. Esto facilita mucho la modificación de sistemas.

16.3 Las funciones miembro por lo regular están formadas por unas pocas líneas de código, porque ninguna lógica es requerida para determinar si son válidos los miembros de datos.

16.4 Los clientes tienen acceso a la interfaz de una clase, pero no deberán tener acceso a la puesta en práctica de la clase.

16.5 Declarar funciones miembro dentro de una definición de clase y definir dichas funciones miembro por fuera de dicha definición de clase separa la interfaz de una clase de su puesta en práctica. Esto promueve buena ingeniería de software.

16.6 Usar un enfoque de programación orientado a objetos a menudo puede simplificar las llamadas de función al reducir el número de parámetros a pasarse. Este beneficio de la programación orientada a objetos se deriva del hecho que el encapsulado de los miembros de datos y las funciones miembros dentro de un objeto le da a las funciones miembro el derecho de acceso a los miembros de datos.

16.7 Coloque la declaración de la clase en un archivo de cabecera a incluirse por cualquier cliente que desee utilizar dicha clase. Esto forma la interfaz pública de la clase. Coloque las definiciones de las funciones miembro de la clase en un archivo fuente. Esto conforma la puesta en práctica de la clase.

16.8 Los clientes de una clase no necesitan ver el código fuente de la clase a fin de utilizarla. Los clientes deben, sin embargo, poder tener la capacidad de enlazarse con el código objeto de la clase.

16.9 Aquella información de importancia a la interfaz con una clase debería estar incluida en el archivo de cabecera. Aquella información que sólo será utilizada en forma interna en la clase y que no será necesaria para los clientes de la clase, debería ser incluida en el archivo fuente no publicado. Este es otra vez otro ejemplo del principio del mínimo privilegio.

16.10 C++ alienta a que los programas sean independientes de la puesta en práctica. Cuando se modifica la puesta en práctica de una clase utilizada mediante código independiente de la misma, dicho código no necesita ser cambiado, pero pudiera necesitar ser recopilado.

16.11 Conserve privados todos los miembros de datos de una clase. Proporcione funciones miembro públicas para definir los valores de los miembros de datos privados y para obtener los valores de los miembros de datos privados. Esta arquitectura ayuda a ocultar la puesta en práctica de una clase a la vista de sus clientes, lo que reduce errores y mejora la capacidad de modificación del programa.

16.12 Los diseñadores de clase utilizan miembros privados, protegidos y públicos para obligar a la idea de ocultamiento de información y al principio del mínimo privilegio.

16.13 Hacer privados los miembros de datos de una clase y públicas las funciones miembro de la clase, facilita la depuración, porque los problemas con las manipulaciones de datos se localizan ya sea en las funciones miembros de la clase o en los amigos de la misma

16.14 Las funciones miembro tienen tendencia a agruparse en varias categorías diferentes: funciones que leen y regresan el valor de miembros de datos privados, funciones que definen el valor de miembros de datos privados, funciones que ponen en práctica las características de la clase, y funciones que ejecutan varias tareas mecánicas para la clase, como la inicialización de objetos de clase, la asignación de objetos de clase, la conversión entre clases y tipos incorporados o entre clases y otras clases, y el manejo de memoria para objetos de clase.

16.15 Hacer los miembros de datos privados y controlando su acceso, especialmente el acceso de escritura, a aquellos miembros de datos vía funciones miembro, ayuda a asegurar la integridad de los datos.

16.16 Tener acceso a datos privados mediante las funciones miembro set y get no solamente protege los miembros de datos contra la recepción de valores inválidos, sino también aísla a los clientes de la clase de la representación de los miembros de datos. Entonces, si por alguna razón cambia la representación de los datos (típicamente para reducir la cantidad de almacenamiento requerido o para mejorar el rendimiento), sólo las funciones miembro necesitan modificarse no es necesario que los clientes cambien, en tanto la interfaz proporcionada por las funciones miembro se conserve igual. Los clientes pueden, sin embargo, requerir ser recompilados.

Ejercicios de autoevaluación

16.1 Llene cada uno de los siguientes espacios en blanco:

a) La palabra reservada _____ introduce una definición de estructura.

b) Se tiene acceso a los miembros de clase vía el operador _____ en conjunción con un objeto de clase o vía el operador _____ en conjunción con un apuntador a un objeto de clase.

c) Los miembros de una clase especificados como _____ son sólo accesibles a las funciones miembro de la clase y amigos de la clase.

d) Un _____ es una función miembro especial utilizada para inicializar los miembros de datos de una clase.

e) El acceso por omisión para los miembros de una clase es _____.

f) Una función _____ se utiliza para asignar valores a los miembros de dato privados de una clase.

g) _____ puede ser utilizada para asignar un objeto de una clase a otro objeto de la misma clase.

h) Las funciones miembro de una clase a menudo se hacen _____ y los miembros de datos de una clase se hacen por lo regular _____.

i) Una función _____ se utiliza para recuperar valores de datos privados de una clase.

j) El conjunto de funciones miembro públicas de una clase se conoce como la _____ de la clase.

k) La puesta en práctica de una clase se dice que está oculta de sus clientes o _____.

l) Las palabras reservadas _____ y _____ pueden ser utilizadas para introducir una definición de clase.

m) Los miembros de una clase especificados como _____ son accesibles en cualquier parte en que un objeto de la clase esté en alcance.

16.2 Encuentre el o los errores en cada una de las siguientes y explique cómo corregirlo.

a) Suponga que se declara el siguiente prototipo en la clase `Time`

```
void ~Time(int);
```

b) Lo siguiente es una definición parcial de la clase `Time`

```
class Time {
public:
   //function prototypes
private:
   int hour = 0;
   int minute = 0;
   int second = 0;
};
```

c) Suponga el siguiente prototipo que se declara en la clase `Employee`

```
int Employee(const char *, const char *);
```

Respuestas a los ejercicios de autoevaluación

16.1 a) `struct`. b) punto (`.`), flecha (`->`). c) `private`. d) constructor. e) `private`. f) set. g) copia a nivel de miembro por omisión. h) `private, public`. i) get. j) interfaz. k) encapsulado. l). `class, struct` m) `public`.

16.2 a) Error: no se permite que los destructores regresen valores o tomen argumentos.
Corrección: elimine el tipo de regreso `void` y el parámetro `int` de la declaración.

b) Error: los miembros no pueden ser inicializados de forma explícita en la definición de clase.
Corrección: elimine la inicialización explícita de la definición de clase e inicialice los miembros de datos en un constructor.

c) Error: los constructores no pueden regresar valores.
Corrección: elimine el tipo de regreso `int` de la declaración.

Ejercicios

16.3 ¿Cuál es el propósito del operador de resolución de alcance?

16.4 Compare los conceptos de `struct` y de `class` en C++.

16.5 Proporcione un constructor que sea capaz de utilizar la hora actual de la función `time()` —declarada en el encabezado `time.h`— de la biblioteca estándar de C para inicializar un objeto de la clase `Time`.

16.6 Cree una clase llamada `Complex` para ejecutar aritmética con números complejos. Escriba un programa manejador para probar su clase.

Los números complejos tienen la forma

```
realPart + imaginaryPart * i
```

donde i es

$$\sqrt{-1}$$

Utilice variables de punto flotante para representar los datos privados de la clase. Proporcione una función constructor que permita que se inicialice un objeto de esta clase cuando sea declarado. El constructor deberá contener valores por omisión, por si no se proporcionan inicializadores. Proporcione funciones miembros públicas para cada uno de los siguientes:

a) Suma de dos números `Complex`: las partes reales se suman juntas y las partes imaginarias se suman juntas también.
b) Resta de dos números `Complex`: la parte real del operando derecho se resta de la parte real del operador izquierdo y la parte imaginaria del operador derecho se resta de la parte imaginaria del operador izquierdo.
c) Imprimir números `Complex` en la forma `(a, b)` donde `a` sea la parte real y `b` la parte imaginaria.

16.7 Cree una clase llamada `RationalNumb er` (para ejecutar aritmética con fracciones). Escriba un programa manejador para probar su clase.

Utilice variables enteras para representar los datos privados de la clase el numerador y el denominador. Proporcione una función constructor que permita que se inicialice un objeto de esta clase cuando sea declarado. El constructor deberá contener valores por omisión, para el caso en que no se proporcionen inicializadores y deberá almacenar la fracción en forma simplificada (es decir, la fracción

$$\frac{2}{4}$$

deberá ser almacenada en el objeto como 1 en el numerador y 2 en el denominador). Proporcione funciones miembro públicas para cada una de las siguientes:

a) Suma de dos números `Rational`. El resultado deberá ser almacenado en forma simplificada.
b) Resta de dos números `Rational`. El resultado deberá ser almacenado en forma simplificada.
c) Multiplicación de dos números `Rational`. El resultado deberá ser almacenado en forma simplificada.
d) División de dos números `Rational`. El resultado deberá ser almacenado en forma simplificada.
e) Imprimir números `Rational` en la forma `a/b` donde `a` es el numerador y `b` el denominador.
f) Imprimir números `Rational` en formato de punto flotante.

16.8 Modifique la clase `Time` de la figura 16.10 para incluir una función miembro `tick` que incremente la hora almacenado en un objeto `Time` en un segundo. El objeto `Time` deberá conservarse siempre en un estado consistente. Escriba un programa manejador que pruebe la función miembro `tick` en un ciclo que imprima la hora en formato estándar durante cada iteración del ciclo para ilustrar que la función miembro `tick` funciona correctamente. Asegúrese de probar los casos siguientes:

a) Incrementar al siguiente minuto.
b) Incrementar a la hora siguiente.
c) Incrementar al día siguiente (es decir 11:59:59 PM a 12:00:00 AM).

16.9 Modifique la clase `Date` de la figura 16.12 para que lleve a cabo verificación de errores en los valores de inicializador correspondientes a los miembros de datos `month`, `day` y `year`. También proporcione una función miembro `nextDay` para incrementar los días en uno. El objeto `Date` deberá siempre conservarse en un estado consistente. Escriba un programa manejador que pruebe la función `nextDay` en un ciclo que imprima la fecha durante cada iteración del ciclo, a fin de ilustrar que la función `nextDay` funciona correctamente. Asegúrese de probar los casos siguientes:

a) Incrementar al siguiente mes.
b) Incrementar al siguiente año.

16.10 Combine la clase modificada `Time` del ejercicio 16.8 y la clase modificada `Date` del ejercicio 16.9 en una clase llamada `DateAndTime` (en el capítulo 19 analizaremos la herencia que nos permitirá llevar a cabo esta tarea rápidamente sin tener que modificar las definiciones existentes de clase). Modifique la función `tick` para llamar la función `nextDay` si la hora es incrementado al día siguiente. Modifique la función `printStandard` y `printMilitary` para extraer la fecha en adición a la hora. Escriba un programa manejador para probar la nueva clase `DateAndTime`. Pruebe específicamente incrementando la hora para pasar al día siguiente.

17

Clases: Parte II

Objetivos

- Ser capaz de crear y destruir dinámicamente objetos.
- Ser capaz de especificar objetos constantes y funciones miembro constantes.
- Comprender el objeto de las funciones amigo y de las clases amigo.
- Comprender el uso de los miembros de datos y de las funciones miembro estáticos.
- Comprender los varios tipos de clases contenedor.
- Ser capaz de desarrollar clases iteradoras que recorren los elementos de las clases contenedor.
- Comprender el uso del apuntador this.
- Ser capaz de crear y utilizar clases plantilla.

¿Pero qué, para servir a nuestros propios intereses, nos impide engañar a nuestros amigos?
Charles Churchill

En vez de esta absurda división de los sexos, deberían clasificar a las personas como estáticas y dinámicas.
Evelyn Waugh

Sobre todo lo siguiente: se veraz contigo mismo.
William Shakespeare
Hamlet

No tengas amigos que no sean tus iguales.
Confucius

Sinopsis

17.1 Introducción

En este capítulo continuamos nuestro estudio de las clases y de la abstracción de datos. Analizamos muchos temas más avanzados y colocamos los cimientos para el análisis de la homonimia de clases y de operadores del capítulo 18. El análisis de los capítulos 16 al 18 fomenta el uso de los objetos por parte de los programadores. Después, los capítulos 19 y 20 introducen herencia y polimorfismo las técnicas de la verdadera programación orientada a objetos.

17.2 Objetos constantes y funciones miembro const

Repetidamente hemos enfatizado el *principio del mínimo privilegio* como uno de los principios más fundamentales de una buena ingeniería de software. Veamos una forma en la cual este principio se aplica a los objetos.

Algunos objetos necesitan ser modificables y otros no. El programador puede utilizar la palabra reservada **const** para indicar que un objeto no es modificable, y que será un error cualquier intento de modificarlo. Por ejemplo,

```
const Time noon(12, 0, 0);
```


declara un objeto **const** de nombre **noon** de la clase **Time** y lo inicializa a las 12 de medio día.

Observación de ingeniería de software 17.1

*Declarar un objeto como **const** ayuda a que se cumpla el principio del mínimo privilegio. Cualquier intento accidental de modificar dicho objeto será detectado en tiempo de compilación, en vez de causar errores en tiempo de ejecución.*

Los compiladores C++ respetan de forma tan rígida las declaraciones **const**, que para objetos **const** no permiten de ninguna manera llamadas de función miembro (algunos compiladores sólo emiten una advertencia). Esto es duro, porque es probable que los clientes del objeto desearán utilizar varias funciones miembro "get" con dicho objeto. Para ello, el programador pudiera declarar funciones miembro **const**; sólo éstas pueden operar sobre objetos **const**. Naturalmente, las funciones miembro **const** no pueden modificar el objeto.

Error común de programación 17.1

*Definir como **const** una función miembro que modifique un miembro de datos de un objeto.*

Error común de programación 17.2

*Definir como **const** una función miembro que llama a una función miembro no **const**.*

Error común de programación 17.3

*Llamar a una función miembro no **const** en relación con un objeto **const**.*

Error común de programación 17.4

*Intentar modificar un objeto **const**.*

Una función es definida como **const** *tanto* en su declaración como en su definición, insertando la palabra reservada **const** después de la lista de parámetros de la función, y, en el caso de la definición de función, antes de la llave izquierda que inicia el cuerpo de la función. Por ejemplo, la función miembro

```
int getValue() const {return privateDataMember};
```

que solo regresa el valor de uno de los miembros de datos del objeto, es declarada como **const**. Si una función miembro **const** se define fuera de la definición de la clase, tanto la declaración como la definición de la función miembro deberán incluir **const**.

Observación de ingeniería de software 17.2

*Una función miembro **const** puede ser homónima en una versión no **const**. El compilador selecciona de forma automática la función miembro homónima basándose en si el objeto ha sido declarado **const** o no.*

Aquí se presenta un problema interesante tratándose de constructores y destructores, los cuales ciertamente modifican los objetos. Para constructores y destructores de objetos **const** no se requiere la declaración **const**. A un constructor debe permitírsele poder modificar un objeto de forma tal que el objeto pueda ser inicializado de manera correcta. Un destructor debe ser capaz de ejecutar sus trabajos de terminación, antes de que un objeto sea destruido.

El programa de la figura 17.1 ejemplifica un objeto constante de la clase **Time** e intenta modificar el objeto mediante las funciones miembro no constantes **setHour**, **setMinute** y **setSecond**. Las advertencias generadas por el compilador Borland C++ aparecen en la ventana de salida. El compilador fue ajustado de forma que no compile, si se produjese cualquier mensaje de advertencia.

Práctica sana de programación 17.1

Declare como **const** *todas las funciones miembro que se pretenda utilizar con objetos* **const**.

Un objeto **const** no puede ser modificado por asignación, por lo que deberá ser inicializado. Cuando un objeto miembro es **const**, un *inicializador de miembro* deberá ser utilizado para proporcionar al constructor con valores iniciales correspondientes al objeto. La figura 17.2 demuestra los usos de la sintaxis del inicializador de miembro para inicializar el miembro de datos **const** de nombre **increment** de la clase **Increment**. El constructor correspondiente a **Increment** se modifica como sigue:

```
// TIME5.H
// Declaration of the class Time.
// Member functions defined in TIME5.CPP

#ifndef TIME5_H
#define TIME5_H

class Time {
public:
   Time(int = 0, int = 0, int = 0);  // default constructor

   // set functions
   void setTime(int, int, int);  // set time
   void setHour(int);      // set hour
   void setMinute(int);    // set minute
   void setSecond(int);    // set second

   // get functions (normally declared const)
   int getHour() const;    // return hour
   int getMinute() const; // return minute
   int getSecond() const; // return second

   // print functions (normally declared const)
   void printMilitary() const;  // print military time
   void printStandard() const;  // print standard time
private:
   int hour;               // 0 - 23
   int minute;             // 0 - 59
   int second;             // 0 - 59
};

#endif
```

Fig. 17.1 Cómo utilizar una clase **Time** con objetos **const** y funciones miembro ***const*** (parte 1 de 4).


```
// TIME5.CPP
// Member function definitions for Time class.
#include "time5.h"
#include <iostream.h>

// Constructor function to initialize private data.
// Default values are 0 (see class definition).
Time::Time(int hr, int min, int sec)
{
   hour = (hr >= 0 && hr < 24) ? hr : 0;
   minute = (min >= 0 && min < 60) ? min : 0;
   second = (sec >= 0 && sec < 60) ? sec : 0;
}

// Set the values of hour, minute, and second.
void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}

// Set the hour value
void Time::setHour(int h)
   { hour = (h >= 0 && h < 24) ? h : 0; }

// Set the minute value
void Time::setMinute(int m)
   { minute = (m >= 0 && m < 60) ? m : 0; }

// Set the second value
void Time::setSecond(int s)
   { second = (s >= 0 && s < 60) ? s : 0; }

// Get the hour value
int Time::getHour() const { return hour; }

// Get the minute value
int Time::getMinute() const { return minute; }

// Get the second value
int Time::getSecond() const { return second; }

// Display military format time: HH:MM:SS
void Time::printMilitary() const
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}
```

Fig. 17.1 Cómo utilizar una clase **Time** con objetos **const** y funciones miembro **const** (parte 2 de 4).

```
// Display standard format time: HH:MM:SS AM (or PM)
void Time::printStandard() const
{
   cout << ((hour == 12) ? 12 : hour % 12) << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}
```

Fig. 17.1 Cómo utilizar una clase **Time** con objetos **const** y funciones miembro **const** (parte 3 de 4).

```
// FIG17_1.CPP
// Attempting to access a const object with
// non-const member functions.
#include <iostream.h>
#include "time5.h"

main()
{
   const Time t(19, 33, 52);    // constant object

   t.setHour(12);     // ERROR: non-const member function
   t.setMinute(20);   // ERROR: non-const member function
   t.setSecond(39);   // ERROR: non-const member function

   return 0;
}
```

```
Compiling FIG17_1.CPP:
Warning FIG17_1.CPP: Non-const function
   Time::setHour(int) called for const object
Warning FIG17_1.CPP: Non-const function
   Time::setMinute(int) called for const object
Warning FIG17_1.CPP: Non-const function
   Time::setSecond(int) called for const object
```

Fig. 17.1 Cómo utilizar una clase **Time** con objetos **const** y funciones miembro **const** (parte 4 de 4).

```
Increment::Increment(int c, int i)
   : increment(i)
{ count = c; }
```

La notación **: increment(i)** hace que **increment** sea inicializado al valor i. Si se requieren de varios inicializadores de miembro, simplemente inclúyalos después de los dos puntos en una lista separada por comas.


```
// FIG17_2.CPP
// Using a member initializer to initialize a
// constant of a built-in data type.

#include <iostream.h>

class Increment {
public:
   Increment(int c = 0, int i = 1);
   void addIncrement() { count += increment; }
   void print() const;
private:
   int count;
   const int increment;          // const data member
};

// Constructor for class Increment
Increment::Increment(int c, int i)
   : increment(i)    // Member initializer
{ count = c; }

// Print the data
void Increment::print() const
{
   cout << "count = " << count
        << ", increment = " << increment << endl;
}

main()
{
   Increment value(10, 5);

   cout << "Before incrementing: ";
   value.print();

   for (int j = 1; j <= 3; j++) {
      value.addIncrement();
      cout << "After increment " << j << ": ";
      value.print();
   }

   return 0;
}
```

```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5
```

Fig. 17.2 Cómo utilizar un inicializador de miembro para inicializar una constante de un tipo de datos incorporado.

En la figura 17.3 se ilustran los errores de compilador para un programa que intenta inicializar **increment** mediante un enunciado de asignación, en vez de mediante un inicializador de miembro.

Error común de programación 17.5

No proporcionar un inicializador de miembro para un objeto miembro **const**.

Observación de ingeniería de software 17.3

Tanto los objetos **const** *como las "variables"* **const** *necesitan ser inicializadas con sintaxis de inicializador miembro. Las asignaciones no son permitidas.*

17.3 Composición: clases como miembros de otras clases

Un objeto de clase **AlarmClock** necesita saber cuándo se supone debe sonar su alarma, por lo que, ¿por qué no incluir un objeto **Time** como miembro del objeto **AlarmClock**? Tal capacidad se llama *composición*. Una clase puede tener otras clases como miembros.

Observación de ingeniería de software 17.4

Una forma de reutilización del software es la composición, en la cual una clase tiene como miembros objetos de otras clases.

Cuando un objeto entra en alcance, su constructor es llamado automáticamente, por lo que es preciso especificar cómo se pasan argumentos a constructores de objetos miembro. Se construyen los objetos miembro, antes de que los objetos de clase que los incluyen sean construidos.

Observación de ingeniería de software 17.5

Si un objeto tiene varios objetos miembro, está indefinido el orden en el cual los objetos miembro serán construidos. No escriba código que dependa de que los constructores de objetos miembro ejecuten su trabajo en un orden específico.

En la figura 17.4 la clase **Employee** y la clase **Date** se utilizan para demostrar objetos como miembros de otros objetos. La clase **Employee** contiene los miembros de datos privados **lastName**, **firstName**, **birthDate**, y **hireDate**. Los miembros **birthDate** y **hireDate** son objetos de la clase **Date** que contiene los miembros de datos privados **month**, **day** y **year**. El programa produce un objeto **Employee**, e inicializa y muestra sus miembros de datos. Note la sintaxis del encabezado de función en la definición de constructor **Employee**:

```
Employee::Employee(char *fname, char *lname,
                   int bmonth, int bday, int byear,
                   int hmonth, int hday, int hyear)
   : birthDate(bmonth, bday, byear),
     hireDate(hmonth, hday, hyear)
```

El constructor toma ocho argumentos (**fname**, **lname**, **bmonth**, **bday**, **byear**, **hmonth**, **hday** y **hyear**). Los puntos en el encabezado separan los inicializadores de miembro de la lista de parámetros. Los inicializadores de miembro definen los argumentos **Employee** pasados a los constructores de los objetos miembro. Entonces, **bmonth**, **bday** y **byear** son pasados al constructor **birthDate**, y **hmonth**, **hday** y **hyear** son pasados al constructor **hireDate**. Varios inicializadores miembro son separados por comas.


```
// FIG17_3.CPP
// Attempting to initialize a constant of
// a built-in data type with an assignment.
#include <iostream.h>

class Increment {
public:
   Increment(int c = 0, int i = 1);
   void addIncrement() { count += increment; }
   void print() const;
private:
   int count;
   const int increment;
};

// Constructor for class Increment
Increment::Increment(int c, int i)
{              // Constant member 'increment' is not initialized
   count = c;
   increment = i;  // ERROR: Cannot modify a const object
}

// Print the data
void Increment::print() const
{
   cout << "count = " << count
        << ", increment = " << increment << '\n';
}

main()
{
   Increment value(10, 5);

   cout << "Before incrementing: ";
   value.print();

   for (int j = 1; j <= 3; j++) {
      value.addIncrement();
      cout << "After increment " << j << ": ";
      value.print();
   }

   return 0;
}
```

```
Compiling FIG17_3.CPP:
Warning FIG17_3.CPP 18: Constant member 'increment' is
   not initialized
Error FIG17_3.CPP 20: Cannot modify a const object
Warning FIG17_3.CPP 21: Parameter 'i' is never used
```

Fig. 17.3 Intento erróneo de inicializar una constante de un tipo de datos incorporado mediante asignación.

```
// DATE1.H
// Declaration of the Date class.
// Member functions defined in DATE1.CPP
#ifndef DATE1_H
#define DATE1_H

class Date {
public:
   Date(int = 1, int = 1, int = 1900);  // default constructor
   void print() const;  // print date in month/day/year format

private:
   int month;  // 1-12
   int day;    // 1-31 based on month
   int year;   // any year

   // utility function to test proper day for month and year
   int checkDay(int);
};

#endif
```

Fig. 17.4 Cómo utilizar los inicializadores objeto miembro (parte 1 de 5).

Un objeto miembro no necesita comenzar mediante un inicializador miembro. Si no se proporciona inicializador miembro, se llamará automáticamente al constructor por omisión del objeto miembro. Los valores, si es que hay alguno, que hayan sido definidos por el constructor por omisión, podrán ser pasados por alto mediante funciones set.

Error común de programación 17.6

No proporcionar un constructor por omisión para un objeto miembro, cuando no se proporciona inicializador miembro para dicho objeto miembro. Esto puede dar como resultado un objeto miembro no inicializado.

Sugerencia de rendimiento 17.1

Inicialice de forma explícita los objetos miembro mediante inicializadores miembro. Esto elimina la sobrecarga de una doble inicialización de objetos miembro una vez cuando se llame al constructor por omisión del objeto miembro, y una segunda vez cuando se utilicen las funciones set para inicializar dicho objeto miembro.

17.4 Funciones amigo y clases amigo

Una *función amigo* de una clase se define por fuera del alcance de dicha clase, pero aún así tiene el derecho de acceso a los miembros **private** (y como veremos en el capítulo 19, "Herencia", a los miembros **protected**) de la clase. Se puede declarar una función o toda una clase como un **friend** de otra clase.

Para declarar una función como un **friend** de una clase, en la definición de clase preceda el prototipo de función con la palabra reservada **friend**. Para declarar la clase **ClassTwo** como amigo de la clase **ClassOne**, prepare una declaración de la forma

```
friend ClassTwo;
```

como un miembro de la clase **ClassOne**.


```
// DATE1.CPP
// Member function definitions for Date class.

#include <iostream.h>
#include "date1.h"


// Constructor: Confirm proper value for month;
// call utility function checkDay to confirm proper
// value for day.
Date::Date(int mn, int dy, int yr)
{
   month = (mn > 0 && mn <= 12) ? mn : 1;  // validate
   year = yr;                              // could also check
   day = checkDay(dy);                     // validate

   cout << "Date object constructor for date ";
   print();
   cout << endl;
}

// Utility function to confirm proper day value
// based on month and year.
int Date::checkDay(int testDay)
{
   static int daysPerMonth[13] = {0, 31, 28, 31, 30,
                                  31, 30, 31, 31, 30,
                                  31, 30, 31};


   if (month != 2) {

      if (testDay > 0 && testDay <= daysPerMonth[month])
         return testDay;

   }
   else {  // February: Check for possible leap year
      int days = (year % 400 == 0 ||
                  (year % 4 == 0 && year % 100 != 0)) ? 29 : 28);

      if (testDay > 0 && testDay <= days)
         return testDay;
   }

   cout << "Day " << testDay << " invalid. Set to day 1.\n";
   return 1;  // leave object in consistent state if bad value
}


// Print Date object in form  month/day/year
void Date::print() const
   { cout << month << '/' << day << '/' << year; }
```

Fig. 17.4 Cómo utilizar inicializadores objeto miembro (parte 2 de 5).

```
// EMPLY1.H
// Declaration of the Employee class.
// Member functions defined in EMPLY1.CPP
#ifndef EMPLY1_H
#define EMPLY1_H

#include "date1.h"

class Employee {
public:
   Employee(char *, char *, int, int, int, int, int, int);
   void print() const;
private:
   char lastName[25];
   char firstName[25];
   Date birthDate;
   Date hireDate;
};

#endif
```

Fig. 17.4 Cómo utilizar inicializadores objeto miembro (parte 3 de 5).

```
// EMPLY1.CPP
// Member function definitions for Employee class.
#include <iostream.h>
#include <string.h>
#include "emply1.h"
#include "date1.h"

Employee::Employee(char *fname, char *lname,
                   int bmonth, int bday, int byear,
                   int hmonth, int hday, int hyear)
   : birthDate(bmonth, bday, byear),
     hireDate(hmonth, hday, hyear)
{
   strncpy(firstName, fname, 24);
   firstName[24] = '\0';
   strncpy(lastName, lname, 24);
   lastName[24] = '\0';
   cout << "Employee object constructor: "
        << firstName << ' ' << lastName << endl;
}

void Employee::print() const
{
   cout << lastName << ", " << firstName << "\nHired: ";
   hireDate.print();
   cout << "  Birthday: ";
   birthDate.print();
   cout << endl;
}
```

Fig. 17.4 Cómo usar inicializadores objeto miembro (parte 4 de 5).


```
// FIG17_4.CPP
// Demonstrating an object with a member object.
#include <iostream.h>
#include "emply1.h"

main()
{
   Employee e("Bob", "Jones", 7, 24, 49, 3, 12, 88);

   cout << endl;
   e.print();

   cout << "\nTest Date constructor with invalid values:\n";
   Date d(14, 35, 94);  // invalid Date values

   return 0;
}
```

```
Date object constructor for date 7/24/49
Date object constructor for date 3/12/88
Employee object constructor: Bob Jones

Jones, Bob
Hired: 3/12/88  Birthday: 7/24/49

Test Date constructor with invalid values:
Day 35 invalid. Set to day 1.
Date object constructor for date 1/1/94
```

Fig. 17.4 Cómo utilizar los inicializadores objeto miembro (parte 5 de 5).

Observación de ingeniería de software 17.6

Los conceptos de acceso de miembros correspondientes a `private`, `protected` *y* `public` *no tienen relación con las declaraciones de amistad, por lo que las declaraciones de amistad pueden ser colocadas en cualquier parte de la definición de clase.*

Práctica sana de programación 17.2

Coloque en la clase todas las definiciones de amistad en primer término, después del encabezado de clase, y no las anteceda con ningún especificador de acceso de miembros.

La amistad es concedida, y no tomada, es decir; para que la clase B sea un amigo de la clase A, la clase A debe declarar que la clase B es su amigo. También, la amistad no es ni simétrica ni transitiva, es decir, si la clase A es un amigo de la clase B, y la clase B es un amigo de la clase C, usted no puede inferir que la clase B sea un amigo de la clase A, que la clase C sea un amigo de la clase B o que la clase A sea un amigo de la clase C.

Observación de ingeniería de software 17.7

Algunas personas en la comunidad OOP (programación orientada a objetos) sienten que la "amistad" corrompe el ocultamiento de la información y debilita el valor del enfoque de diseño orientado a objetos.

El programa de la figura 17.5 demuestra la declaración y el uso de la función amigo **setX**, para definir el miembro de datos privado **x** de la clase **count**. Note que en la declaración de clase la declaración **friend** aparece primero (por regla convencional), inclusive antes que las funciones miembro públicas sean declaradas. El programa de la figura 17.6 demuestra los mensajes producidos por el compilador cuando es llamada la función no amigo **cannotSetX**, para modificar el miembro de datos privado **x**.

Es posible especificar funciones homónimas como amigos de una clase. Cada función homónima que se desea como un amigo, debe ser declarada en forma explícita en la definición de clase como amigo de la clase.

```
// FIG17_5.CPP
// Friends can access private members of a class.
#include <iostream.h>

// Modified Count class
class Count {
   friend void setX(Count &, int);  // friend declaration
public:
   Count() { x = 0; }               // constructor
   void print() const { cout << x << endl; }  // output
private:
   int x;  // data member
};

// Can modify private data of Count because
// setX is declared as a friend function of Count
void setX(Count &c, int val)
{
   c.x = val;  // legal: setX is a friend of Count
}

main()
{
   Count object;

   cout << "object.x after instantiation: ";
   object.print();
   cout << "object.x after call to setX friend function: ";
   setX(object, 8);  // set x with a friend
   object.print();

   return 0;
}
```

```
object.x after instantiation: 0
object.x after call to setX friend function: 8
```

Fig. 17.5 Los amigos pueden tener acceso a los miembros privados de una clase.


```
// FIG17_6.CPP
// Non-friend/non-member functions cannot access
// private data of a class.
#include <iostream.h>

// Modified Count class
class Count {
public:
   Count() { x = 0; }                    // constructor
   void print() const { cout << x << '\n'; }  // output
private:
   int x;  // data member
};

// Function tries to modify private data of Count,
// but cannot because it is not a friend of Count.
void cannotSetX(Count &c, int val)
{
   c.x = val;  // ERROR: 'Count::x' is not accessible
}

main()
{
   Count object;

   cannotSetX(object, 3); // cannotSetX is not a friend

   return 0;
}
```

```
Compiling FIG17_6.CPP:
Error FIG17_6.CPP 17: 'Count::x' is not accessible
Warning FIG17_6.CPP 18: Parameter 'c' is never used
Warning FIG17_6.CPP 18: Parameter 'val' is never used
```

Fig. 17.6 Funciones no amigas/no miembro no pueden tener acceso a los miembros privados de una clase.

17.5 Cómo utilizar el apuntador this

Cuando una función miembro referencia otro miembro de una clase en relación con un objeto específico de dicha clase, ¿Cómo C++ se asegura que es referenciado el objeto correcto? La respuesta es que cada objeto mantiene un apuntador a sí mismo llamado el *apuntador* ***this*** que es un argumento implícito en todas las referencias a miembros incluidos dentro de dicho objeto. El apuntador **this** puede también ser utilizado en forma explícita. Mediante el uso de la palabra reservada **this**, cada objeto puede determinar su propia dirección.

El apuntador **this** es utilizado de manera implícita para referenciar tanto los miembros de datos como las funciones miembro de un objeto. El tipo de este apuntador **this** depende del tipo del objeto y de si es declarada **const** la función miembro en la cual **this** es utilizado. En una función miembro no constante de la clase **Employee** el apuntador **this** tiene el tipo **Employee * const** (un apuntador constante a un objeto **Employee**). En una función miembro constante

de la clase **Employee** el apuntador **this** tiene el tipo **Employee const * const** (un apuntador constante a un objeto **Employee** constante).

Por ahora, presentamos un ejemplo sencillo de cómo utilizar de manera explícita el apuntador **this**; más adelante mostraremos algunos ejemplos substanciales y sutiles del uso de **this**. Cualquier función miembro tiene acceso al apuntador **this** al objeto para el cual está siendo invocado el miembro.

Sugerencia de rendimiento 17.2

Por razones de economía de almacenamiento, existe sólo una copia de cada función miembro por clase, y esta función miembro es invocada para todos los objetos de dicha clase. Por otra parte, cada objeto tiene su propia copia de los miembros de datos de la clase.

En la figura 17.7 se demuestra el uso explícito del apuntador **this** para permitir que una función miembro de la clase **Test** imprima los datos privados **x** de un objeto **Test**. El programa utiliza tanto el operador de flecha (->) sobre el apuntador **this**, como el operador de punto (.) sobre el apuntador **this** desreferenciado.

```
// FIG17_7.CPP
// Using the this pointer to refer to object members.
#include <iostream.h>

class Test {
public:
   Test(int = 0);
   void print() const;
private:
   int x;
};

Test::Test(int a) { x = a; }    // constructor

void Test::print() const
{
   cout << "        x = " << x
        << "\n  this->x = " << this->x
        << "\n(*this).x = " << (*this).x << '\n';
}

main()
{
   Test a(12);

   a.print();

   return 0;
}
```

```
        x = 12
  this-> = 12
(*this).x = 12
```

Fig. 17.7 Cómo utilizar el apuntador **this**.

Note los paréntesis que encierran a ***this** cuando se utiliza con el operador de selección de miembro punto (.). Los paréntesis son necesarios, porque el operador punto tiene una precedencia más alta que el operador *. Sin los paréntesis, la expresión

```
*this.x
```

sería evaluada como si tuviera los paréntesis como sigue:

```
*(this.x)
```

Esta expresión sería tratada por el compilador C++ como un error de sintaxis, porque el operador de selección de miembro no puede ser utilizado con un apuntador.

Error común de programación 17.7

Intentar utilizar el operador de selección de miembro (.) con un apuntador a un objeto (el operador de selección de miembro sólo puede ser utilizado con un objeto o con una referencia a un objeto).

Un uso interesante del apuntador **this**, es impedir que un objeto sea asignado a sí mismo. Como veremos en el capítulo 18, "Homonimia de operadores", la autoasignación puede causar errores serios cuando los objetos contengan apuntadores a almacenamientos asignados de forma dinámica mediante el operador **new**.

En la figura 17.8 se ilustra regresar una referencia a un objeto **Time** para permitir llamadas de función miembro de la clase **Time**, para concatenarlas. Cada una de las funciones miembro **Time** de nombre **setTime**, **setHour**, **setMinute** y **setSecond** regresan ***this** con un tipo de regreso **Time &**.

¿Porqué es válida la técnica de regresar ***this** como una referencia? El operador punto (.) se asocia de izquierda a derecha, por lo que la expresión

```
t.setHour(18).setMinute(30).setSecond(22);
```

primero evalúa **t.setHour(18)** y a continuación, regresa una referencia al objeto **t** como el valor de esta llamada de función. El resto de la expresión es entonces interpretado como

```
t.setMinute(30).setSecond(22);
```

La llamada **t.setMinute(30)** es ejecutada, regresando el equivalente de **t**. El resto de la expresión se interpreta como

```
t.setSecond(22);
```

Note que las llamadas

```
t.setTime(20, 20, 20).printStandard();
```

también utilizan la característica de concatenación. En esta expresión, estas llamadas deberán aparecer en este orden, porque **printStandard**, como está definida en la clase, no regresa una referencia a **t**. Colocar, en el enunciado anterior, la llamada a **printStandard** antes de la llamada a **setTime**, resultaría en un error de sintaxis.

```
// TIME6.H
// Declaration of class Time.
// Member functions defined in TIME6.CPP

#ifndef TIME6_H
#define TIME6_H

class Time {
public:
   Time(int = 0, int = 0, int = 0);  // default constructor

   // set functions
   Time &setTime(int, int, int);  // set hour, minute, second
   Time &setHour(int);     // set hour
   Time &setMinute(int);   // set minute
   Time &setSecond(int);   // set second

   // get functions (normally declared const)
   int getHour() const;    // return hour
   int getMinute() const; // return minute
   int getSecond() const; // return second

   // print functions (normally declared const)
   void printMilitary() const;  // print military time
   void printStandard() const;  // print standard time
private:
   int hour;               // 0 - 23
   int minute;             // 0 - 59
   int second;             // 0 - 59
};

#endif
```

Fig. 17.8 Cómo encadenar llamadas de función miembro (parte 1 de 4).

```
// TIME6.CPP
// Member function definitions for Time class.

#include "time6.h"
#include <iostream.h>

// Constructor function to initialize private data.
// Default values are 0 (see class definition).
Time::Time(int hr, int min, int sec)
{
   hour = (hr >= 0 && hr < 24) ? hr : 0;
   minute = (min >= 0 && min < 60) ? min : 0;
   second = (sec >= 0 && sec < 60) ? sec : 0;
}
```

Fig. 17.8 Cómo encadenar llamadas de función miembro (parte 2 de 4).


```
// Set the values of hour, minute, and second.
Time &Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
   return *this;   // enables chaining
}

// Set the hour value
Time &Time::setHour(int h)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   return *this;   // enables chaining
}

// Set the minute value
Time &Time::setMinute(int m)
{
   minute = (m >= 0 && m < 60) ? m : 0;
   return *this;   // enables chaining
}

// Set the second value
Time &Time::setSecond(int s)
{
   second = (s >= 0 && s < 60) ? s : 0;
   return *this;   // enables chaining
}

// Get the hour value
int Time::getHour() const { return hour; }

// Get the minute value
int Time::getMinute() const { return minute; }

// Get the second value
int Time::getSecond() const { return second; }

// Display military format time: HH:MM:SS
void Time::printMilitary() const
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}

// Display standard format time: HH:MM:SS AM (or PM)
void Time::printStandard() const
{
   cout << ((hour == 12) ? 12 : hour % 12) << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}
```

Fig. 17.8 Cómo encadenar llamadas de función miembro (parte 3 de 4).

```
// FIG17_8.CPP
// Chaining member function calls together
// with the this pointer
#include <iostream.h>
#include "time6.h"

main()
{
   Time t;

   t.setHour(18).setMinute(30).setSecond(22);
   cout << "Military time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();

   cout << "\n\nNew standard time: ";
   t.setTime(20, 20, 20).printStandard();
   cout << endl;

   return 0;
}
```

```
Military time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

Fig. 17.8 Cómo encadenar llamadas de función miembro (parte 4 de 4).

17.6 Asignación dinámica de memoria mediante los operadores new y delete

Los operadores ***new*** y ***delete*** ofrecen una mejor forma de efectuar la asignación dinámica de memoria, que mediante las llamadas de función **malloc** y **free** de C. Considere el código siguiente

```
TypeName *typeNamePtr;
```

En ANSI C, para crear dinámicamente un objeto del tipo **TypeName**, usted diría

```
typeNamePtr = malloc(sizeof(TypeName));
```

Esto requiere una llamada de función a **malloc** y una referencia explícita al operador **sizeof**. En versiones de C anteriores a ANSI C, también tendría que haber hecho una conversión explícita (cast) del apuntador regresado por **malloc**, utilizando el cast (**TypeName ***). La función **malloc** no tiene ningún procedimiento para inicializar el bloque de memoria asignado. En C++, simplemente usted escribe

```
typeNamePtr = new TypeName;
```

El operador **new** crea en forma automática un objeto del tamaño apropiado, llama el constructor para el objeto (si existe uno disponible) y regresa un apuntador del tipo correcto. Si **new** no puede

encontrar espacio, regresa un apuntador **0**. Para liberar espacio para este objeto en C++, utilice el operador **delete**, como sigue:

```
delete typeNamePtr;
```

C++ le permite incluir un *inicializador* para un objeto recién creado como en:

```
float *thingPtr = new float (3.14159);
```

el cual inicializa a **3.14159** un objeto **float** recién creado.

Se puede crear un arreglo y asignarlo a **int * chessBoardPtr** como sigue:

```
chessBoardPtr = new int[8][8];
```

Este arreglo puede ser borrado mediante el enunciado

```
delete [] chessBoardPtr;
```

Como veremos, utilizar **new** y **delete**, en vez de **malloc** y **free**, también ofrece otros beneficios. En particular, **new** de manera automática invoca al constructor, y **delete** automáticamente invoca al destructor de la clase.

Error común de programación 17.8

*Mezclar asignación dinámica de memoria del tipo **new** y **delete** con asignación dinámica de memoria del tipo **malloc** y **free**: el espacio creado por **malloc** no podrá ser liberado por **delete**; los objetos creados mediante **new** no podrán ser borrados por **free**.*

Práctica sana de programación 17.3

*A pesar de que los programas C++ pueden contener almacenamiento creado por **malloc** y borrados por **free**, y objetos creados por **new** y borrados por **delete**, lo mejor es utilizar solo **new** y **delete**.*

17.7 Miembros de clase estáticos

Por lo regular, cada objeto de una clase tiene su propia copia de todos los miembros de datos de la clase. Esto a veces resulta un desperdicio. En ciertos casos todos los miembros de una clase deberían compartir una copia de un miembro de datos particular. Para esta y otras razones se utiliza un miembro de datos estático. Un miembro de datos estático representa información "aplicable a toda la clase". La declaración de un miembro estático empieza con la palabra reservada **static.**

Sugerencia de rendimiento 17.3

Cuando una copia de los datos sea suficiente, utilice miembros de datos estáticos a fin de ahorrar almacenamiento.

Aunque los miembros de datos estáticos pudieran parecer variables globales, los miembros de datos estáticos tienen alcance de clase. Los miembros estáticos pueden ser públicos, privados o protegidos. Los miembros de datos estáticos deben ser inicializados en alcance de archivo. Los miembros de clase estáticos públicos son accesibles a través de cualquier objeto de dicha clase o mediante el operador de resolución de alcance binario, se puede tener acceso a ellos a través del nombre de la clase. Para tener acceso a los miembros de clase estáticos privados y protegidos se deben utilizar las funciones miembro públicas de la clase. Los miembros de clase estáticos existen aun cuando no existan objetos de dicha clase. Para tener acceso a un miembro de clase estático público cuando no existan objetos de dicha clase, sólo coloque el nombre de clase y el operador de resolución de alcance binario como prefijo al miembro de datos. Para tener acceso a un miembro

de clase estático privado o protegido, cuando no existan objetos de dicha clase, deberá ser incluida una función miembro estática pública y la función deberá ser llamada mediante el uso de un prefijo en su nombre con el nombre de la clase junto con el operador de resolución de alcance binario.

El programa de la figura 17.9 demuestra el uso del miembro de datos **static** privado y de la función miembro **static** pública. El miembro de datos **count** es inicializado a cero en alcance de archivo, mediante el enunciado

```
int Employee::count = 0;
```

El miembro de datos **count** lleva cuenta del número de objetos de la clase **Employee** que han sido producidos. Cuando existan objetos de la clase **Employee**, el miembro **count** puede ser referenciado a través de cualquier función miembro de un objeto **Employee** —en este ejemplo, **count** es referenciado tanto por el constructor como por el destructor. Cuando no existan objetos de la clase **Employee**, aun así el miembro **count** se puede referenciar, pero sólo mediante una llamada a la función miembro estática **getCount**, como sigue:

```
Employee::getCount()
```

En este ejemplo, se utiliza la función **getCount** para determinar el número de objetos **Employee** en la actualidad producidos. Note que cuando no exista ningún objeto producido, la llamada de función ya citada será emitida. Sin embargo, se podrá llamar a la función **getCount** a través de uno de los objeto cuando existan objetos producidos, como en

```
e1Ptr->getCount()
```

```
// EMPLOY1.H
// An employee class
#ifndef EMPLOY1_H
#define EMPLOY1_H

class Employee {
public:
   Employee(const char*, const char*);  // constructor
   ~Employee();                 // destructor
   char *getFirstName() const;  // return first name
   char *getLastName() const;   // return last name

   // static member function
   static int getCount();  // return # objects instantiated
private:
   char *firstName;
   char *lastName;

   // static data member
   static int count;  // number of objects instantiated
};

#endif
```

Fig. 17.9 Cómo utilizar un miembro de datos estático para llevar cuenta del número de objetos de una clase (parte 1 de 5).


```
// EMPLOY1.CPP
// Member functions definitions for class Employee
#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "employ1.h"

// Initialize the static data member
int Employee::count = 0;

// Define the static member function that
// returns the number of employee objects instantiated.
int Employee::getCount() { return count; }

// Constructor dynamically allocates space for the
// first and last name and uses strcpy to copy
// the first and last names into the object
Employee::Employee(const char *first, const char *last)
{
   firstName = new char[ strlen(first) + 1 ];
   assert(firstName != 0);   // ensure memory allocated
   strcpy(firstName, first);

   lastName = new char[ strlen(last) + 1 ];
   assert(lastName != 0);    // ensure memory allocated
   strcpy(lastName, last);

   ++count;  // increment static count of employees
   cout << "Employee constructor for " << firstName
        << ' ' << lastName << " called.\n";
}

// Destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   cout << "~Employee() called for " << firstName
        << ' ' << lastName << endl;
   delete [] firstName;  // recapture memory
   delete [] lastName;   // recapture memory
   --count;  // decrement static count of employees
}

// Return first name of employee
char *Employee::getFirstName() const
{
   char *tempPtr = new char[strlen(firstName) + 1];
   assert(tempPtr != 0);   // ensure memory allocated
   strcpy(tempPtr, firstName);
   return tempPtr;
}
```

Fig. 17.9 Cómo utilizar un miembro de datos estático para llevar cuenta del número de objetos de una clase (parte 2 de 5).

```
// Return last name of employee
char *Employee::getLastName() const
{
   char *tempPtr = new char[strlen(lastName) + 1];
   assert(tempPtr != 0);   // ensure memory allocated
   strcpy(tempPtr, lastName);
   return tempPtr;
}
```

Fig. 17.9 Cómo utilizar un miembro de datos estático para llevar cuenta del número de objetos de una clase (parte 3 de 5).

```
// FIG17_9.CPP
// Driver to test the employee class
#include <iostream.h>
#include "employ1.h"

main()
{
   cout << "Number of employees before instantiation is "
        << Employee::getCount() << endl;   // use class name

   Employee *e1Ptr = new Employee("Susan", "Baker");
   Employee *e2Ptr = new Employee("Robert", "Jones");

   cout << "Number of employees after instantiation is "
        << e1Ptr->getCount() << endl;

   cout << "\nEmployee 1: "
        << e1Ptr->getFirstName()
        << " " << e1Ptr->getLastName()
        << "\nEmployee 2: "
        << e2Ptr->getFirstName()
        << " " << e2Ptr->getLastName() << "\n\n";

   delete e1Ptr;   // recapture memory
   delete e2Ptr;   // recapture memory

   cout << "Number of employees after deletion is "
        << Employee::getCount() << endl;

   return 0;
}
```

Fig. 17.9 Cómo utilizar un miembro de datos estático para llevar cuenta del número de objetos de una clase (parte 4 de 5).

Una función miembro puede ser declarada **static** si no tiene acceso a miembros de clase no estáticos. A diferencia de las funciones miembro no estáticas, una función miembro estática no tiene un apuntador **this**, porque los miembros de datos estáticos y las funciones miembro estáticos existen en forma independiente a cualquier objeto de la clase.


```
Number of employees before instantiation is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after instantiation is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() calle for Susan Baker
~Employee() calle for Robert Jones
Number of employees after deletion is 0
```

Fig. 17.9 Cómo utilizar un miembro de datos estático para llevar cuenta del número de objetos de una clase (parte 5 de 5).

Error común de programación 17.9

Referirse al apuntador **this** *desde dentro de una función miembro estática.*

Error común de programación 17.10

Declarar una función miembro estática **const**.

Observación de ingeniería de software 17.8

Los miembros de datos estáticos y las funciones miembro estáticas existen y pueden ser utilizados, aún si no se han producido objetos de dicha clase.

17.8 Abstracción de datos y ocultamiento de información

Las clases ocultan por lo regular sus detalles de puesta en práctica a los clientes (es decir, usuarios) de las clases. Esto se conoce como ocultamiento de la información. Por consiguiente, por ejemplo, el programador puede crear un clase de pila y ocultar a sus usuarios la puesta en práctica de la pila. Mediante arreglos o listas enlazadas las pilas pueden ser puestas en práctica en forma fácil. Un cliente de una clase de pila no necesita saber cómo ha sido puesta en práctica la misma. Lo que en realidad interesa al usuario es que cuándo en la pila se coloquen elementos de datos que éstos serán recuperados en orden de últimas entradas, primeras salidas (LIFO). Este concepto se conoce como *abstracción de datos*, y las clases de C++ definen *tipos de datos abstractos (ADT,* por *abstract data types)*. Aunque los usuarios pudieran conocer los detalles de cómo se pone en práctica una clase, los usuarios no deben escribir código que dependa de dichos detalles. Esto significa que una clase particular (como una que ponga en práctica una pila, junto con sus operaciones **push** y **pop**), puede ser reemplazada por otra versión, sin afectar al resto del sistema, siempre y cuando la interfaz pública de dicha clase no cambie.

La tarea de un lenguaje de alto nivel es crear una presentación conveniente para uso de los programadores. No existe una forma aceptada de presentación y esta es una razón por la cual existen tantos lenguajes de programación. La programación orientada a objetos de C++, presenta sólo otra alternativa.

La mayor parte de los lenguajes de programación enfatizan acciones. En estos lenguajes, los datos existen en apoyo de acciones que los programas necesitan llevar a cabo. De todas formas, los datos son "menos interesantes" que las acciones. Los datos son "corrientes". Sólo existen unos

cuantos tipos de datos incorporados, y para los programadores resulta difícil crear sus propios tipos de datos.

En C++ y en el estilo de programación orientado a objetos, esta panorámica cambia. C++ aumenta la importancia de los datos. En C++ la actividad primordial es la creación de nuevos tipos de datos (es decir clases) y expresar las interacciones entre objetos de dichos tipos de datos.

Para ir en esa dirección, la comunidad de los lenguajes de programación necesitaba formalizar algunos conceptos relacionados con los datos. La formalización a la que nos estamos refiriendo es el concepto de tipo de datos abstracto (ADT por abstract data type). ADT recibe tanta atención, hoy en día, como tuvo durante las dos últimas décadas la programación estructurada. ADT no sustituye la programación estructurada. Más bien, provee una formalización adicional, que puede mejorar aún más el proceso de desarrollo de programas.

¿Qué es un tipo de datos abstracto? Veamos el tipo incorporado `int`. Lo que viene a la mente es el concepto de un entero en matemáticas, pero en una computadora `int` no es precisamente lo que es un entero en matemáticas. En particular, por lo regular los `int` de computadora son muy limitados en tamaño. Por ejemplo, en una máquina de 32 bits `int` pudiera estar limitado al rango desde -2 mil millones hasta +2 mil millones. Si el resultado de un cálculo cae fuera de este rango, ocurrirá un error y la máquina responderá de alguna manera dependiente de la máquina. Los enteros matemáticos no tienen este problema. Por lo tanto, el concepto de un `int` de computadora es solamente una aproximación al concepto de un entero del mundo real. Lo mismo es cierto tratándose de `float`.

Inclusive `char` es también una aproximación. A menudo los valores del tipo `char` son patrones de 8 bits, que no se parecen en nada a los caracteres que se supone deben representar como una `Z`, una `z`, un signo de dólares (`$`), un dígito (por ejemplo, `5`), y así en lo sucesivo. En la mayor parte de las computadoras los valores del tipo `char` están bastante limitados, en comparación con el rango de los caracteres del mundo real. El conjunto de caracteres ASCII de 7 bits proporciona 127 distintos valores de caracteres. Esto resulta en totalidad inadecuado para representar lenguajes como el japonés y el chino, que requieren de miles de caracteres.

El punto es que inclusive los tipos de datos incorporados, proporcionados en lenguajes de programación como C++, son sólo aproximaciones o modelos de conceptos y comportamientos del mundo real. Hasta este momento hemos aceptado como es a `int`, pero ahora el lector tiene que pensar con una nueva perspectiva. Los tipos como `int`, `float`, `char` y otros son todos ejemplos de *tipos de datos abstractos*. Son en esencia formas de representar conceptos del mundo real, a cierto nivel satisfactorio de precisión, dentro de un sistema de computación.

Un tipo de datos abstracto de hecho captura dos conceptos, es decir una representación de datos, así como aquellas operaciones permitidas sobre dichos datos. Por ejemplo, en C++, el concepto de `int` define la adición, substracción, multiplicación, división y operaciones de módulo, pero queda indefinida la división entre cero; y estas operaciones permitidas se llevan a cabo de una forma que resulta dependiente de los parámetros de la máquina, como son el tamaño de la palabra fija del sistema de computación subyacente. En C++, para poner en práctica tipos de datos abstractos, el programador utiliza las clases.

17.8.1 Ejemplo: tipo de datos abstracto de arreglo

En el capítulo 6, analizamos los arreglos. Un arreglo no es mucho más que un apuntador. Si el programador es cuidadoso, esta capacidad primitiva es aceptable para efectuar operaciones de arreglos. Existen muchas operaciones que sería bueno ejecutar con arreglos, pero que en C++ no están incorporadas. Con C++, el programador puede desarrollar un arreglo ADT, que es preferible a los arreglos "en bruto". La clase arreglo puede proveer muchas nuevas capacidades como

- Verificación del rango de los subíndices.
- Un rango arbitrario de subíndices.
- Asignación de arreglo.
- Comparación de arreglo.
- Entrada/salida de arreglo.
- Los arreglos conocen su tamaño.

La debilidad aquí es que estamos creando un tipo de datos personalizado, no estándar, que es poco probable que esté disponible precisamente de esa forma, en la mayor parte de las puestas en práctica de C++. El uso de la programación orientada a objetos y de C++ está aumentando con rapidez. Resulta crucial que la profesión de la programación propugne por una normalización y distribución a gran escala de bibliotecas de clases, para poder aprovechar todo el potencial de la orientación a objetos.

C++ tiene un pequeño conjunto de tipos incorporados. ADT amplía el lenguaje de programación base.

Observación de ingeniería de software 17.9

El programador puede crear nuevos tipos mediante el uso del mecanismo de clases. Estos nuevos tipos pueden ser diseñados para ser usados tan convenientemente como los tipos incorporados. Por lo tanto, C++ resulta un lenguaje extensible. A pesar de que mediante estos nuevos tipos el lenguaje es fácil de ampliar, el lenguaje base mismo no es modificable.

Los nuevos ADT creados en entornos C++ pueden ser propiedad de un individuo, de pequeños grupos o de empresas. Los ADT también pueden ser colocados en bibliotecas de clases estándar destinadas a una amplia difusión. Esto no necesariamente promueve la normalización, aunque de hecho es probable que emergerán estándares. El valor pleno de C++ únicamente será comprendido cuando bibliotecas de clase sustanciales y estándar resulten ampliamente disponibles. Un proceso institucional debe ser instaurado para alentar el desarrollo de bibliotecas estándar. En los Estados Unidos, a menudo dicha normalización ocurre a través de ANSI, el American National Standards Institute. ANSI está desarrollando actualmente una versión estándar de C++. Independiente de cómo aparezcan estas bibliotecas en última instancia, el lector que aprenda C++ y programación orientada a objetos, estará listo para rápidamente aprovechar los nuevos tipos de desarrollo de software orientado a componentes, que serán posibles con las bibliotecas de ADT.

17.8.2 Ejemplo: tipo de datos abstracto de cadena

Intencionalmente C++ es un lenguaje escueto, que proporciona a los programadores sólo aquellas capacidades fundamentales necesarias para elaborar un amplio rango de sistemas. Los diseñadores del lenguaje no deseaban crear sobrecargas de rendimiento. Su objetivo era que C++ fuese apropiado para la programación de sistemas, que demandan que los programas se ejecuten con eficiencia. Es cierto, entre los tipos incorporados de C++, hubiera sido posible incluir un tipo de cadena, pero en vez de ello los diseñadores optaron incluir un mecanismo mediante clases para la creación y puesta en práctica de dichos ADT. En el capítulo 18 desarrollaremos nuestro propio ADT para cadenas.

17.8.3 Ejemplo: tipo de datos abstracto de cola

Cada uno de nosotros de vez en cuando tiene que esperar en fila. Una fila de espera también se llama una *cola*. Esperamos en fila en la caja de salida de un supermercado, esperamos en fila para

obtener gasolina, esperamos en fila para subirnos al autobús, esperamos en fila en la autopista para pagar el peaje, y los estudiantes saben demasiado bien sobre tener que esperar en fila durante la época de registro para obtener las materias que desean. Los sistemas de cómputo utilizan de forma interna muchas filas de espera, y necesitamos escribir programas que simulen lo que las colas hacen y son.

Una cola es un bonito ejemplo de un tipo de datos abstracto. Una cola ofrece a sus clientes un comportamiento bien comprendido. Los clientes ponen cosas en una cola uno a la vez utilizando una operación *enqueue*, y los clientes obtienen estas cosas de regreso sobre demanda uno a la vez utilizando la operación *dequeue*. Conceptualmente, una cola puede hacerse muy larga. Una cola real, por lo regular, es finita. Los elementos de una cola son devueltos en un orden de *primeras entradas, primeras salidas (FIFO por first-in, first-out)*.

La cola oculta una representación de datos internos, que de alguna forma lleva control de los elementos actualmente esperando en fila, y ofrece un conjunto de operaciones a sus clientes, es decir *enqueue* y *dequeue*. Los clientes no tienen que preocuparse respecto a la puesta en práctica de la cola. Los clientes solo desean que la cola funcione "como anunciado". Cuando un cliente pone un nuevo elemento en cola, ésta deberá aceptar dicho elemento y colocarlo internamente en algún tipo de estructura de primeras entradas, primeras salidas. Cuando el cliente desee el siguiente elemento de la parte frontal de la cola, la cola deberá quitar el elemento de su representación interna y entregar el elemento al mundo exterior en un orden fijo, es decir, el elemento que ha estado por más tiempo en la cola, deberá ser el siguiente que, en la siguiente operación *dequeue*, será devuelto.

El ADT de la cola garantiza la integridad de su estructura interna de datos. Los clientes no pueden manipular en directo esta estructura de datos. Solo el ADT de la cola tiene acceso a sus datos internos. Los clientes pueden causar sólo operaciones permitidas, para ejecutarse sobre la representación de los datos; las operaciones no contempladas en la interfaz pública del ADT son rechazadas por el ADT de alguna forma apropiada. Esto podría significar la emisión de un mensaje de error, la terminación de la ejecución o sólo ignorar la solicitud de operación.

17.9 Clases contenedor e iteradores

Entre los tipos más populares de clases son las *clases contenedor* (también llamadas *clases de colección*), es decir, clases diseñadas para contener colecciones de objetos. Las clases contenedor proporcionan por lo común servicios como es inserción, borrado, búsqueda, clasificación, prueba de un elemento verificando membresía dentro de la clase, y otras similares. Los arreglos y las listas enlazadas son ejemplos de clases contenedor.

Es común asociar *objetos de iterador* o solo *iteradores* con la clase de colección. Un iterador es un objeto que regresa el siguiente elemento de una colección. Una vez que se ha escrito un iterador para una clase, obtener el siguiente elemento de la clase puede ser expresado en forma simple. Los iteradores se escriben como amigos de las clases, a través de las cuales hacen la iteración. De la misma forma que un libro se comparte entre varias personas y podría tener a la vez varios marcatextos, una clase contenedor puede tener varios iteradores operando sobre ella simultáneamente.

17.10 Clases plantilla

En el capítulo 15 analizamos funciones plantilla y cómo ayudan a la reutilización del software. Cerramos este capítulo con un análisis de las *clases plantilla*.

Es posible comprender que una pila es algo independiente del tipo de elementos que se colocan dentro de la pila. Pero cuando se trata de hecho programar una pila, se debe proporcionar un tipo

de datos. Esto crea una maravillosa oportunidad para la reutilización del software. Lo que necesitamos es un medio de describir el concepto genérico de pila y de producir clases que sean copias de esta clase genérica, pero de tipo específico. Esta es la capacidad proporcionada en C++ por las clases plantilla.

Observación de ingeniería de software 17.10

Las clases plantilla fomentan la reutilización del software.

Las clases plantilla se pusieron a disposición con el compilador de C++ de AT&T Versión 3, por lo que son una adición relativa reciente al lenguaje.

Las clases plantilla a menudo se conocen como *tipos parametrizados,* porque requieren de uno o más parámetros de tipo para especificar cómo personalizar la especificación de plantilla genérica.

El programador que desee utilizar clases plantilla, sólo escribe una definición de clase plantilla genérica. Cada vez que el programador necesite una nueva clase, el programador utiliza una notación concisa y simple, y el compilador escribe el código fuente para la clase que especifica el programador. Una clase plantilla de pila, por ejemplo podría entonces convertirse en base para crear muchas clases de pilas (como sería "una pila de `float`", "una pila de `int`", "una pila de `char`", etcétera) utilizadas en un programa.

Note la definición de la clase plantilla de pila en la figura 17.10. Se parece a una definición de clase convencional, excepto que está precedida por el encabezado

```
template <class T>
```

para indicar que se trata de una definición de clase plantilla que recibe un parámetro de tipo `T` indicando el tipo de la clase de pila que el programador desea crear. El tipo de elemento a almacenarse en esta pila se menciona sólo genéricamente como `T` a todo lo largo del encabezado de clase de pila y de las definiciones de función miembro.

```
// TSTACK1.H
// Simple template class Stack

#ifndef TSTACK1_H
#define TSTACK1_H

template <class T>
class Stack {
public:
   Stack(int = 10);       // Constructor with default size 10
   ~Stack() { delete [] stackPtr; } // Destructor
   int push(const T&);    // Push an element onto the stack
   int pop(T&);           // Pop an element off the stack
   int isEmpty() const { return top == -1; } // 1 if empty
   int isFull() const { return top == size - 1; } // 1 if full
private:
   int size;              // # of elements in the stack
   int top;               // location of the top element
   T *stackPtr;           // pointer to the stack
};
```

Fig. 17.10 Definición de la clase plantilla `Stack` (parte 1 de 4).

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

```
// Constructor with default size 10
template <class T>
Stack<T>::Stack(int s)
{
   size = s;
   top = -1;  // Empty stack
   stackPtr = new T[size];
}

// Push an element onto the stack
// return 1 if successful, 0 otherwise
template <class T>
int Stack<T>::push(const T &item)
{
   if (!isFull()) {
      stackPtr[++top] = item;
      return 1;  // push successful
   }
   return 0;     // push unsuccessful
}

// Pop an element off the stack
template <class T>
int Stack<T>::pop(T &popValue)
{
   if (!isEmpty()) {
      popValue = stackPtr[top--];
      return 1;  // pop successful
   }
   return 0;     // pop unsuccessful
}

#endif
```

Fig. 17.10 Definición de la clase plantilla **Stack** (parte 2 de 4).

Ahora veamos el manejador de prueba (la función **main**) correspondiente a la clase plantilla de pila. El manejador empieza produciendo el objeto **floatStack** del tamaño 5. Este objeto se declara ser de la clase **Stack<float>**. El compilador asocia el tipo **float** con el tipo parametrizado **T** en la plantilla para producir el código fuente para una clase de pila del tipo **float**.

El manejador a continuación inserta (push) los valores **float** 1.1, 2.2, 3.3, 4.4 y 5.5 sobre **floatStack**. El ciclo de inserción termina cuando el manejador intenta insertar un sexto valor sobre **floatStack** (mismo que ya está lleno porque fue creado con 5 elementos).

Ahora el manejador extrae (pop) los 5 valores de la pila (en orden LIFO). El manejador intenta extraer un sexto valor, pero **floatStack** está vacío, por lo que el ciclo de extracción termina.

A continuación, el manejador produce una pila de enteros con la declaración

```
Stack<int>    intStack;
```

Dado que no se especifica tamaño, el tamaño por omisión será de 10, tal y como se especifica en el constructor por omisión. Otra vez, el manejador cicla la inserción de valores sobre **intStack**, hasta que éste queda lleno, y a continuación cicla la extracción de valores de **intStack** hasta que queda vacío.


```
// FIG17_10.CPP
// Test driver for Stack template

#include <iostream.h>
#include "tstack1.h"

main()
{
   Stack<float> floatStack(5);
   float f = 1.1;
   cout << "Pushing elements onto floatStack\n";

   while (floatStack.push(f)) { // success (1 returned)
      cout << f << ' ';
      f += 1.1;
   }

   cout << "\nStack is full. Cannot push " << f
        << "\n\nPopping elements from floatStack" << endl;

   while (floatStack.pop(f))  // success (1 returned)
      cout << f << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;

   Stack<int> intStack;
   int i = 1;
   cout << "\nPushing elements onto intStack\n";

   while (intStack.push(i)) { // success (1 returned)
      cout << i << ' ';
      i++;
   }

   cout << "\nStack is full. Cannot push " << i
        << "\n\nPopping elements from intStack" << endl;

   while (intStack.pop(i))  // success (1 returned)
      cout << i << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;
   return 0;
}
```

Fig. 17.10 Manejador para la clase plantilla **Stack** (parte 3 de 4).

Las definiciones de función miembro por fuera del encabezado de clase plantilla cada una de ellas empieza con el encabezado

```
template <class T>
```

Entonces cada definición se parece a una definición convencional, a excepción que el tipo de elemento de pila siempre se lista genéricamente como del tipo parametrizado **T**. Como siempre,

```
Pushing elements onto floatStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from floatStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Fig. 17.10 Manejador para la clase plantilla `Stack` (parte 4 de 4).

se utiliza el operador de resolución de alcance binario para unir cada definición de función miembro al alcance correcto de la clase. En este caso, el nombre de clase es **`Stack<T>`**. Cuando **`floatStack`** es producido para ser del tipo **`Stack<float>`**, para representar la pila el constructor **`Stack`** crea un arreglo de elementos del tipo **`T`**. El tipo **`float`** sustituye a **`T`** y el enunciado

```
stackPtr = new T[size];
```

es generado por el compilador en la versión **`Stackat<float>`**, como

```
stackPtr = new float[size];
```

Los tipos parametrizados (es decir, las plantillas) pueden tomar más de un parámetro, como en

```
template <class T, class S, class R
```

Cada parámetro debe estar precedido por la palabra reservada **`class`**.

Resumen

- La palabra reservada **`const`** puede ser utilizada para especificar que un objeto no es modificable y que cualquier intento de modificarlo es un error.
- El compilador de C++ no permite llamadas de función miembro no **`const`** para un objeto **`const`**.
- Las funciones miembro **`const`** no pueden modificar un objeto.
- Una función se especifica como **`const`** tanto en su declaración como en su definición.
- Una función miembro **`const`** puede ser homónima con una versión no **`const`**. El compilador escogerá qué función miembro homónima utilizar en forma automática, basado en si el objeto ha sido declarado **`const`** o no.
- Un objeto **`const`** debe ser inicializado.

- Cuando una clase contenga objetos miembro **`const`**, los inicializadores miembros deben ser proporcionados al constructor.
- Las clases pueden estar compuestas de objetos de otras clases.
- Los objetos miembros son construidos antes de que sean construidos sus objetos de clase que los encierran.
- Si un inicializador miembro no es proporcionado para un objeto miembro, se llamará al constructor por omisión del objeto miembro.
- Una función amigo de una clase es una función definida por fuera de dicha clase y tiene derecho de acceso a miembros **`private`** y **`protected`** de la clase.
- Las declaraciones de amistad pueden ser colocadas en cualquier parte de la definición de clase.
- Cada función homónima que se pretende sea un amigo debe ser declarada de forma explícita como amigo de la clase.
- Cada objeto mantiene un apuntador a sí mismo, llamado apuntador **`this`**, que se convierte en un argumento implícito en todas las referencias a miembros dentro de dicho objeto. Este apuntador **`this`** es utilizado de forma implícita para referenciar tanto las funciones miembro como los miembros de datos de un objeto.
- Cada objeto puede determinar su propia dirección utilizando la palabra reservada **`this`**.
- El apuntador **`this`** pude ser utilizado explícitamente, pero se utiliza de manera implícita más a menudo.
- El operador **`new`** crea en forma automática un objeto del tamaño correcto, y regresa un apuntador del tipo correcto. En C++, para liberar el espacio correspondiente a este objeto, usted utiliza el operador **`delete`**.
- Un miembro de datos estático representa información "accesible a toda la clase". La declaración de un miembro estático empieza con la palabra reservada **`static`**.
- Los miembros de datos estáticos tienen alcance de clase.
- Los miembros estáticos de una clase son accesibles a través de un objeto de dicha clase, o pueden ser accesibles a través del nombre de la clase mediante el uso del operador de resolución de alcance.
- Una función miembro puede ser declarada estática, si no tiene acceso a miembros de clase no estáticos. A diferencia de las funciones miembro no estáticas, una función miembro estática no tiene apuntador **`this`**. Esto es debido a que los miembros de datos estáticos y las funciones miembro estáticas existen en forma independiente de cualquiera de los objetos de una clase.
- Las clases plantilla proporcionan una forma genérica de describir el concepto de una clase y de producir clases que son copias de esta clase genérica, pero son de tipo específico.
- Las clases plantilla a menudo se llaman tipos parametrizados, porque requieren de uno o más parámetros de tipo para especificar cómo personalizar la especificación de plantilla genérica.
- Las definiciones de clases plantilla empiezan con la línea

  ```
  template <class T>
  ```

  en la línea que antecede a la definición de clase. Puede existir más de un tipo parametrizado. Si es así, estarán separados por comas y cada tipo estará precedido por la palabra reservada **`class`**.

- Una clase plantilla se produce especificando el tipo de la clase que sigue al nombre de la clase, como sigue:

```
ClassName<type> objectName;
```

El compilador sustituirá `type` a todo lo largo de la definición de clase y de la definición de la función miembro correspondiente por el tipo parametrizado de la clase.

Terminología

operador de resolución de alcance binario (`::`)
llamadas de función de miembro encadenado
alcance de clase
composición
función miembro `const`
objeto `const`
constructor
clases contenedor
constructor por omisión
destructor por omisión
operador `delete`
operador `delete [ ]`
destructor
objetos dinámicos
extensibilidad
clase `friend`
función `friend`
iterador
operador de selección de miembro (`.`)
especificadores de acceso de miembro
inicializador de miembro
objeto miembro
constructor de objeto miembro
clase anidada
operador `new`
tipo parametrizado
operador selector de miembro de apuntador (`->`)
apuntador a miembro
principio de mínimo privilegio
miembro de datos estático
función miembro estática
clase plantilla
apuntador `this`

Errores comunes de programación

17.1 Definir como `const` una función miembro que modifique un miembro de datos de un objeto.
17.2 Definir como `const` una función miembro que llama a una función miembro no `const`.
17.3 Llamar a una función miembro no `const` en relación con un objeto `const`.
17.4 Intentar modificar un objeto `const`.
17.5 No proporcionar un inicializador de miembro para un objeto miembro `const`.
17.6 No proporcionar un constructor por omisión para un objeto miembro, cuando no se proporciona inicializador miembro para dicho objeto miembro. Esto puede dar como resultado un objeto miembro no inicializado.
17.7 Intentar utilizar el operador de selección de miembro (`.`) con un apuntador a un objeto (el operador de selección de miembro sólo puede ser utilizado con un objeto o con una referencia a un objeto).
17.8 Mezclar asignación dinámica de memoria del tipo `new` y `delete` con asignación dinámica de memoria del tipo `malloc` y `free`: el espacio creado por `malloc` no podrá ser liberado por `delete`; los objetos creados mediante `new` no podrán ser borrados por `free`.
17.9 Referirse al apuntador `this` desde dentro de una función miembro estática.
17.10 Declarar una función miembro estática `const`.

Prácticas sanas de programación

17.1 Declare como `const` todas las funciones miembro que se pretenda utilizar con objetos `const`.
17.2 Coloque en la clase todas las definiciones de amistad en primer término, después del encabezado de clase, y no las anteceda con ningún especificador de acceso de miembros.

17.3 A pesar de que los programas C++ pueden contener almacenamiento creado por `malloc` y borrados por `free`, y objetos creados por `new` y borrados por `delete`, lo mejor es utilizar solo `new` y `delete`.

Sugerencias de rendimiento

17.1 Inicialice de forma explícita los objetos miembro mediante inicializadores miembro. Esto elimina la sobrecarga de una doble inicialización de objetos miembro una vez cuando se llame al constructor por omisión del objeto miembro, y una segunda vez cuando se utilicen las funciones set para inicializar dicho objeto miembro.
17.2 Por razones de economía de almacenamiento, existe sólo una copia de cada función miembro por clase, y esta función miembro es invocada para todos los objetos de dicha clase. Por otra parte, cada objeto tiene su propia copia de los miembros de datos de la clase.
17.3 Cuando una copia de los datos sea suficiente, utilice miembros de datos estáticos a fin de ahorrar almacenamiento.

Observaciones de ingeniería de software

17.1 Declarar un objeto como `const` ayuda a que se cumpla el principio del mínimo privilegio. Cualquier intento accidental de modificar dicho objeto será detectado en tiempo de compilación, en vez de que causar errores en tiempo de ejecución.
17.2 Una función miembro `const` puede ser homónima en una versión no `const`. El compilador selecciona de forma automática la función miembro homónima basándose en el objeto que ha sido declarado `const` o no.
17.3 Tanto los objetos `const` como las "variables" `const` necesitan ser inicializadas con sintaxis de inicializador miembro. Las asignaciones no son permitidas.
17.4 Una forma de reutilización del software es la composición, en la cual una clase tiene como miembros objetos de otras clases.
17.5 Si un objeto tiene varios objetos miembro, está indefinido el orden en el cual los objetos miembro serán construidos. No escriba código que dependa de que los constructores de objetos miembro ejecuten su trabajo en un orden específico.
17.6 Los conceptos de acceso de miembros correspondientes a `private`, `protected` y `public` no tienen relación con las declaraciones de amistad, por lo que las declaraciones de amistad pueden ser colocadas en cualquier parte de la definición de clase.
17.7 Algunas personas en la comunidad OOP (Programación orientada a objetos) sienten que la "amistad" corrompe el ocultamiento de la información y debilita el valor del enfoque de diseño orientado a objetos.
17.8 Los miembros de datos estáticos y las funciones miembro estáticas existen y pueden ser utilizados, aun si no se han producido objetos de dicha clase.
17.9 El programador puede crear nuevos tipos mediante el uso del mecanismo de clases. Estos nuevos tipos pueden ser diseñados para ser usados tan conveniente como los tipos incorporados. Por lo tanto; C++ resulta un lenguaje extensible. A pesar de que mediante estos nuevos tipos el lenguaje es fácil de ampliar, el lenguaje base mismo no es modificable.
17.10 Las clases plantilla fomentan la reutilización del software.

Ejercicios de autoevaluación

17.1 Llene cada uno de los siguientes espacios en blanco:
a) Se utiliza sintaxis _____ para inicializar los miembros constantes de una clase.
b) En una clase debe declararse un _____ para tener acceso a los miembros de datos `private` de dicha clase.

c) El operador _____ asigna dinámicamente memoria para un objeto de un tipo especifico y regresa un _____ a dicho tipo.
d) Un objeto constante debe de ser _____; no puede ser modificado después de creado.
e) Un miembro de datos _____ representa información para todo el ámbito de la clase.
f) Todo objeto mantiene un apuntador a sí mismo llamado el apuntador _____.
g) _____ proporcionan un medio de describir el concepto genérico de una clase.
h) La palabra reservada _____ especifica que un objeto o una variable no es modificable.
i) Si no se proporciona un inicializador de miembro para un objeto miembro de una clase, se llama al _____ del objeto.
j) Una función miembro puede ser **static** si no tiene acceso a los miembros de clase_____.
k) Las funciones amigo pueden tener acceso a los miembros _____ y _____ de una clase.
l) Las clases plantilla se conocen a veces como tipos _____.
m) Los objetos miembro se construyen _____ que el objeto de clase que los contiene.
n) El operador _____ recupera memoria asignada previamente por **new**.

17.2 Encuentre el error en cada uno de los siguientes y explique cómo corregirlo.

a)
```
char *string;
string = new char[20];
free(string);
```

b)
```
class Example {
public:
   Example(int y = 10) { data = y; }
   int getIncrementedData() const { return ++data; }
   static int getCount()
   {
      cout << "Data is " << data << endl;
      return count;
   }
private:
   int data;
   static int getCount;
};
```

Respuestas a los ejercicios de autoevaluación

17.1 a) Inicializador de miembro. b) **friend** .c) **new**, apuntador. d) inicializado. e) **static**. f) **this**. g) clases plantilla. h) **const**. i) constructor por omisión. j) no **static**. k) **private, protected**. l) parametrizado. m) antes. n) **delete**.

17.2 a) Error: la memoria asignada dinámicamente por **new** es borrada por la función de biblioteca estándar de C **free**.
Corrección: utilice el operador **delete** de C++ para recuperar la memoria. La asignación dinámica de memoria de tipo C no debe de ser mezclada con los operadores de C++ **new** y **delete**.

b) Error: la definición de clase correspondiente a **Example** tiene dos errores. el primero ocurre en la función **getIncrementedData**. La función es declarada **const**, pero modifica el objeto.
Corrección: Para corregir el primer error, elimine la palabra reservada **const** de la definición de **getIncrementedData**.

Error: el segundo error ocurre en la función **getCount**. Esta función es declarada **static**, por lo que no se le permite acceso a cualquier miembro no **static** de la clase.

Corrección: para corregir el segundo error, elimine la línea de salida de la definición de **getCount**.

Ejercicios

17.3 Compare la asignación dinámica de memoria mediante los operadores de C++ **new** y **delete**, con la asignación dinámica de memoria utilizando las funciones de la biblioteca estándar de C **malloc** y **free**.

17.4 Explique el concepto de amistad en C++. Explique los aspectos negativos de la amistad, tal y como se describen en el texto.

17.5 ¿Puede una definición correcta de clase **Time** incluir ambos constructores siguientes?

```
Time (int h = 0, int m = 0, int s = 0);
Time();
```

17.6 ¿Qué ocurre cuando un tipo de regreso, inclusive el tipo **void**, es especificado para un constructor o destructor?

17.7 Cree una clase **Date** con las siguientes capacidades:

a) Extraer la fecha en formatos múltiples como

```
DDD YYYY
MM/DD/YY
June 14, 1992
```

b) Utilizar constructores homónimos para crear objetos **Date** inicializados con fechas de los formatos de la parte (a).
c) Crear un constructor **Date** que lea la fecha del sistema utilizando las funciones estándar de biblioteca del encabezado **time.h** y que defina los miembros **Date**.

En el capítulo 18, podremos crear operadores para probar la igualdad de dos fechas y para comparar fechas, a fin de determinar si una fecha es anterior a otra o posterior a otra.

17.8 Utilizando como guía el programa de procesamiento de lista de la figura 12.3, construya una clase **List**. La clase deberá proporcionar las mismas capacidades que el ejemplo. Escriba un programa manejador para probar por completo dicha clase.

17.9 Utilizando como guía el programa de procesamiento de colas de la figura 12.13, construya una clase **Queue**. La clase deberá proporcionar las mismas capacidades que el ejemplo. Escriba un programa manejador para probar completamente dicha clase.

17.10 Utilizando como punto de partida la clase **List** desarrollada en el ejercicio 17.8, cree una clase plantilla **List**, que sea capaz de producir objetos **List** que contengan muchos tipos de datos diferentes. Escriba un programa manejador para probar completamente la clase plantilla.

17.11 Utilizando como punto de partida la clase **Queue** desarrollada en el ejercicio 17.9, cree una clase plantilla **Queue** que sea capaz de producir objetos **Queue** que contengan muchos tipos de datos diferentes. Escriba un programa manejador para probar por completo la clase plantilla.

18

Homonimia de operadores

Objetivos

- Comprender cómo redefinir operadores para que funcionen con nuevos tipos.
- Comprender cómo convertir objetos de una clase a otra clase.
- Aprender cuándo efectuar, y cuándo no efectuar, la homonimia de operadores.
- Estudiar varias clases interesantes que utilizan operadores homónimos.

La diferencia entre construcción y creación es exactamente esto: que una cosa construida puede ser querida sólo después de construida; pero una cosa creada es querida antes de que exista.
Gilbert Keith Chesterton
Prefacio a Dickens, Pickwick Papers

El dado ha sido lanzado.
Julius Caesar

En realidad jamás nuestro doctor operaba, a menos de que fuese necesario. El era precisamente así. Si no necesitaba el dinero, no ponía una mano sobre usted.
Herb Shriner

Sinopsis

18.1 Introducción

En los capítulos 16 y 17, presentamos los fundamentos de las clases C++ y el concepto de tipos de datos abstractos (ADT). Las manipulaciones sobre los objetos de clase (es decir, sobre ejemplos de ADT) se llevaron a cabo enviando mensajes a los objetos (bajo forma de llamadas de funciones miembro). Para ciertos tipos de clases esta notación de llamada de función resulta engorrosa, en especial para clases matemáticas. Sería agradable, para estos tipos de clases, utilizar el extenso repertorio de operadores incorporados de C++ concebidos para especificar manipulaciones de objetos. En este capítulo, mostramos cómo habilitar a los operadores de C++ para funcionar con objetos de clase. Este proceso se conoce como homonimia de operador. Es un proceso directo y natural extender C++ con estas nuevas capacidades.

En C++ el operador `<<` es utilizado para varios fines como operador de inserción de flujos y como operador de desplazamiento a la izquierda. Este es un ejemplo de *homonimia de operador*. Similar, `>>` también tiene su homónimo; es utilizado tanto como operador de extracción de flujo como operador de desplazamiento a la derecha. En la biblioteca de clases de C++, ambos operadores tienen homónimos. El lenguaje de C++ mismo hace la homonimia de + y de —. Estos operadores funcionan en forma diferente dependiendo del contexto en aritmética de enteros, aritmética de punto flotante y aritmética de apuntadores.

En general, C++ le permite al programador hacer homónimos de la mayor parte de los operadores, haciéndolos sensibles al contexto dentro del cual son utilizados. El compilador generará el código apropiado, basándose en la forma en la cual el operador sea utilizado. Algunos operadores se hacen homónimos en forma frecuente, en especial el operador de asignación y varios

operadores aritméticos, como + y -. La tarea llevada a cabo por los operadores homónimos, también pueden ser ejecutada mediante llamadas de función explícitas, pero la notación de operador es por lo general más legible.

Analizaremos cuándo utilizar homonimia de operadores y cuando no utilizarla. Mostraremos cómo hacer la homonimia de operadores, y presentaremos muchos programas completos utilizando operadores homónimos.

18.2 Fundamentos de la homonimia de operadores

La programación en C++ es un proceso sensible a los tipos y enfocado a los tipos. Los programadores pueden utilizar tipos incorporados y pueden definir nuevos tipos. Los tipos incorporados pueden ser utilizados aprovechando la extensa colección de operadores de C++. Los operadores proporcionan a los programadores una forma de notación concisa para expresar manipulaciones de objetos de tipos incorporados.

Los programadores pueden también utilizar operadores con tipos definidos por usuario. Aunque C++ no permite que se creen operadores nuevos, sí permite que se haga la homonimia de algunos operadores existentes, de tal forma que estos operadores sean utilizados con objetos de clase, teniendo los operadores un significado apropiado a los nuevos tipos. Esta es una de las características más poderosas de C++. La homonimia de operadores contribuye a la extensibilidad de C++, lo que seguro es uno de los atributos más llamativos de este lenguaje.

Práctica sana de programación 18.1

Utilice la homonimia de operadores cuando ésta haga más claro un programa si efectúa las mismas operaciones mediante llamadas de función explícitas.

Práctica sana de programación 18.2

Evite un uso excesivo o inconsistente de la homonimia de operadores, ya que ello puede hacer que un programa resulte críptico o difícil de leer.

A pesar de que la homonimia de operadores pudiera sonar como una capacidad exótica, la mayor parte de los programadores utilizan de forma implícita operadores homónimos en forma regular. Por ejemplo, el operador más (+) opera de forma bien diferente sobre enteros, flotantes y dobles. Pero, con todo, más (+) funciona bien con variables del tipo `int`, `float`, `double` y varios otros tipos incorporados, porque en el lenguaje C++ mismo se hace la homonimia del operador más (+).

Se hace la homonimia de operadores escribiendo una definición de función (con encabezado y cuerpo) como usted lo haría normalmente, excepto que ahora el nombre de la función se convierte en la palabra clave `operator`, seguida por el símbolo correspondiente al operador homónimo. Por ejemplo, para hacer el homónimo del operador de adición se utilizaría el nombre de función `operator+`.

Para utilizar un operador sobre objetos de clase, dicho operador *deberá* ser un homónimo con dos excepciones. El operador de asignación (=) puede ser utilizado sin homónimo con cualquier clase. Cuando no se incluye un operador de asignación homónimo, el comportamiento por omisión es una *copia a nivel de miembro* de los miembros de datos de la clase. Pronto veremos que dicha copia a nivel de miembro por omisión es peligrosa, tratándose de clases con miembros que apunten a almacenamiento dinámicamente asignado; para dichas clases por lo regular haremos la homonimia del operador de asignación. El operador de dirección (&) también puede ser utilizado sin homonimia con objetos de cualquier clase; sólo devuelve la dirección del objeto en memoria. También se puede hacer la homonimia de este operador.

La homonimia es lo más apropiado para clases matemáticas. A menudo éstas requieren que un conjunto substancial de operadores sean homónimos, para asegurar paralelismo con la forma en que estas clases matemáticas son manejadas en el mundo real. Por ejemplo, para una clase de números complejos sería poco usual hacer la homonimia de sólo la adición, dado que con números complejos también son muy utilizados otros operadores aritméticos.

C++ es un lenguaje rico en operadores. Los programadores de C++ comprenden el significado y el contexto de cada operador. Por lo tanto, cuando en relación con nuevas clases se trata de hacer la homonimia de operadores, los programadores probablemente harán selecciones razonables.

El punto de la homonimia de operadores es proporcionar las mismas expresiones concisas para tipos definidos por usuario, que las que C++ proporciona con su rica colección de operadores, en relación con tipos incorporados. La homonimia de operadores, sin embargo, no es automática; para llevar a cabo las operaciones deseadas el programador deberá escribir funciones de homonimia de operadores. Algunas veces estas funciones se realizan mejor como funciones miembro; algunas veces se realizan mejor como funciones amigo; y en ocasiones pueden ser hechas como funciones no miembro, y no amigo.

Un uso equivocado de la homonimia sería hacer la homonimia del operador + para llevar a cabo operaciones de tipo substracción o hacer la homonimia del operador / para ejecutar operaciones de tipo multiplicación. Un uso como éste de la homonimia puede hacer un programa confuso.

Práctica sana de programación 18.3

Haga la homonimia de operadores para llevar a cabo la misma función o funciones muy similares sobre objetos de clase que dichos operadores ejecutan sobre objetos de tipos incorporados.

Práctica sana de programación 18.4

Antes de escribir programas en C++ utilizando operadores homónimos, consulte los manuales de C++ correspondientes a su compilador, para informarse de las varias restricciones y requisitos únicos a operadores particulares.

18.3 Restricciones sobre la homonimia de operadores

Se puede hacer la homonimia de la mayor parte de los operadores de C++. Estos se muestran en la figura 18.1. En la figura 18.2 se muestran los operadores que no pueden tener homónimos.

La precedencia de un operador no puede ser modificada por la homonimia. Esto puede llevar a situaciones incómodas en las cuales un operador tiene un homónimo, pero de forma tal, que su precedencia fija es inapropiada. Sin embargo, se pueden utilizar paréntesis dentro de una expresión, a fin de obligar el orden de evaluación de operadores homónimos.

Operadores que pueden tener homónimos

+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete

Fig. 18.1 Operadores que pueden tener homónimos.

Operadores que no pueden tener homónimos

.	.*	::	?:	sizeof

Fig. 18.2 Operadores que no pueden tener homónimos.

La asociatividad de un operador tampoco puede ser modificada por la homonimia. Para hacer la homonimia de un operador no pueden utilizarse argumentos por omisión.

No es posible modificar el número de operandos que toma un operador: los operadores unarios homónimos se conservan como unarios; los operadores binarios homónimos se conservan como binarios. El único operador ternario de C++, (?:), no puede tener homónimo. Los operadores &, *, + y - cada uno de ellos tienen versiones unaria y binaria; cada una de estas versiones unaria y binaria pueden tener homónimos en forma separada.

No es posible crear operadores nuevos; sólo se puede hacer la homonimia de operadores existentes. Esto impide que el programador utilice notaciones populares como el operador **, utilizado en BASIC para la exponenciación.

Error común de programación 18.1

Intentar crear operadores nuevos.

El significado del funcionamiento de un operador sobre objetos de tipos incorporados no puede ser modificado por la homonimia de operadores. El programador no puede, por ejemplo, cambiar cómo + añade dos enteros. La homonimia de operador funciona sólo con objetos de tipos definidos por el usuario o por una combinación de un objeto de tipo definido por el usuario y un objeto de tipo incorporado.

Error común de programación 18.2

Intentar modificar cómo funciona un operador con objetos de tipos incorporados.

Observación de ingeniería de software 18.1

En una función operador, por lo menos un argumento debe ser un objeto de clase o una referencia a un objeto de clase. Esto impedirá que los programadores modifiquen cómo funcionan los operadores sobre objetos de tipos incorporados.

Al hacer la homonimia de (), [], ->, o = la función de homonimia de operador deberá ser declarada como un miembro de clase. Para los demás operadores, las funciones de homonimia de operador pueden ser amigo.

La homonimia de un operador de asignación y de un operador de adición, para permitir enunciados como

```
object2 = object2 + object1
```

no implica que el operador += quede también de forma automática homónimo para permitir enunciados como

```
object2 += object1;
```

Sin embargo; este comportamiento puede ser conseguido haciendo la homonimia explícita del operador += correspondiente a dicha clase.

Error común de programación 18.3

Suponer que hacer la homonimia de un operador (como +) hace automáticamente la homonimia de sus operadores relacionados (como +=). Los operadores sólo pueden tener homónimos en forma explícita; la homonimia implícita no existe.

18.4 Funciones operador como miembros de clase en comparación con funciones amigo

Las funciones operador pueden ser funciones miembro o funciones no miembro; las funciones no miembro por lo regular son funciones amigo. Las funciones miembro utilizan el apuntador **this** en forma implícita, para obtener uno de sus argumentos de objeto de clase. En una llamada de función no miembro, este argumento de clase deberá estar listado en forma explícita.

Independiente que una función de operador esté puesta en práctica como función miembro o como función no miembro, aun así en expresiones, el operador será utilizado de la misma forma. Por lo tanto, ¿cuál de las puestas en práctica es la mejor?

Cuando una función operador es puesta en práctica como función miembro, el operando más a la izquierda (o único) deberá ser un objeto de clase (o una referencia a un objeto de clase) de la clase del operador. Si el operando izquierdo tiene que ser un objeto de clase diferente o de tipo incorporado, esta función de operador deberá ser puesta en práctica como función no miembro, exactamente como lo hicimos al hacer la homonimia de << y >> como operadores de inserción y de extracción de flujo, respectivamente. Una función operador puesta en práctica como una función no miembro necesita ser un amigo, si dicha función debe tener acceso directo a miembros privados o protegidos de dicha clase.

El operador homónimo << debe tener un operando izquierdo del tipo **ostream &** (como **cout** en la expresión **cout << classObject**), por lo que deberá ser función no miembro. De igual forma, el operador homónimo >> deberá tener un operando izquierdo del tipo **istream &** (como **cin** en la expresión **cin >> classObject**), de forma tal, que también él deberá de ser una función miembro. También, cada una de estas funciones de operador homónimas requiere tener acceso a los miembros de datos privados del objeto de clase que se está extrayendo o introduciendo, por lo que, por razones de rendimiento, esas funciones operador homónimas por lo regular se hacen funciones amigo de la clase.

Sugerencia de rendimiento 18.1

Es posible hacer la homonimia de un operador como función no miembro, y no amigo, pero dicha función con la necesidad de tener acceso a datos privados o protegidos de una clase requeriría utilizar las funciones set o get proporcionadas en la interfaz pública de la clase. La sobrecarga de llamar a estas funciones causaría bajo rendimiento.

Las funciones miembro operador son llamadas sólo cuando el operando izquierdo de un operador binario es de forma específica un objeto de dicha clase o cuando el único operando de un operador unario es un objeto de dicha clase.

Otra razón por la cual uno podría elegir una función no miembro para hacer la homonimia de un operador, sería para permitir que el operador fuera conmutativo. Por ejemplo, suponga que tenemos un objeto, **number**, del tipo **long int** y un objeto **bigInteger1**, de la clase **HugeInteger** (una clase en la cual los enteros pueden ser de manera arbitraria extensos, en vez de quedar limitados por el tamaño de palabra de máquina del hardware subyacente; en los

ejercicios del capítulo se desarrolla la clase **HugeInteger**). El operador de adición (+) produce un objeto temporal **HugeInteger** como la suma de un **HugeInteger** y de un **long int** (como en la expresión **bigInteger1 + number**), o como la suma de un **long int** y un **HugeInteger** (como en la expresión **number + bigInteger1**). Por lo tanto, necesitamos que el operador de adición sea conmutativo (exactamente como normalmente es). El problema estriba en que, si debe hacer la homonimia del operador como función miembro, el objeto de clase debe aparecer a la izquierda de la adición. Por lo tanto, hacemos la homonimia del operador como amigo, para permitir que **HugeInteger** aparezca a la derecha de la adición. La función **operator+**, que se ocupa de **HugeInteger** a la izquierda, aún puede ser una función miembro.

18.5 Cómo hacer la homonimia de operadores de inserción y de extracción de flujo

C++ tiene la capacidad de introducir y de extraer los tipos de datos estándar utilizando los operadores de extracción de flujo >> y de inserción de flujo <<. Se hace la homonimia de estos operadores (incluidos en las bibliotecas de clase proporcionadas con los compiladores C++) para procesar cada tipo de datos estándar, incluyendo cadenas y direcciones de memoria. Los operadores de inserción y de extracción de flujo también pueden tener homónimos para ejecutar entradas y salidas para tipos definidos por usuario. En la figura 18.3 se demuestra la homonimia de los operadores de extracción y de inserción de flujo para manejar datos de una clase de número de teléfono definida por usuario llamada **PhoneNumber**. Este programa supone que los números telefónicos están introducidos de forma correcta. Dejamos para los ejercicios la inclusión de la verificación de errores.

La función operador de extracción de flujo (**operator>>**) toma como argumentos una referencia **istream** (**input**) y una referencia (**num**) a un tipo definido por usuario (**PhoneNumber**), y devuelve una referencia **istream**. En la figura 18.3 se utiliza la función de operador **operator>>** para introducir números telefónicos de la forma

```
(800) 555-1212
```

en objetos de la clase **PhoneNumber**. Cuando el compilador ve en **main** la expresión

```
cin >> phone
```

genera la llamada de función

```
operator>>(cin, phone);
```

Cuando esta llamada se ejecuta, el parámetro **input** se convierte en un seudónimo para **cin** y el parámetro **num** se convierte en un seudónimo para **phone**. La función operador utiliza la función miembro **istream**, de nombre **getline**, para leer como cadenas las tres porciones del número telefónico a los miembros **areaCode**, **exchange** y **line** del objeto referenciado **PhoneNumber** (**num** en la función operador y **phone** en **main**). La función **getline** explica en detalle en el capítulo 21. Los caracteres paréntesis, espacio y guión son pasados por alto, llamando a la función miembro **istream**, de nombre **ignore**, que descarta el número especificado de caracteres del flujo de entrada (un carácter por omisión). La función **operator>>** devuelve la referencia **istream** de nombre **input** (es decir, **cin**). Esto permite que se *concatenen* operaciones de entrada sobre objetos **PhoneNumber** con operaciones de entrada sobre otros objetos **PhoneNumber** o sobre objetos de otros tipos de datos. Por ejemplo, se podrían introducir dos objetos **PhoneNumber** como sigue:

```
cin >> phone1 >> phone2;
```

```
// FIG18_3.CPP
// Overloading the stream-insertion and
// stream-extraction operators.
#include <iostream.h>

class PhoneNumber {
   friend ostream &operator<<(ostream &, const PhoneNumber &);
   friend istream &operator>>(istream &, PhoneNumber &);
private:
   char areaCode[4];  // 3-digit area code and null
   char exchange[4];  // 3-digit exchange and null
   char line[5];      // 4-digit line and null
};

// Overloaded stream insertion operator (cannot be
// a member function).
ostream &operator<<(ostream &output, const PhoneNumber &num)
{
   output << "(" << num.areaCode << ") "
          << num.exchange << "-" << num.line;

   return output;      // enables cout << a << b << c;
}

// overloaded stream extraction operator
istream &operator>>(istream &input, PhoneNumber &num)
{
   input.ignore();                    // skip (
   input.getline(num.areaCode, 4); // input area code
   input.ignore(2);                   // skip ) and space
   input.getline(num.exchange, 4); // input exchange
   input.ignore();                    // skip dash (-)
   input.getline(num.line, 5);     // input line

   return input;       // enables cin >> a >> b >> c;
}

main()
{
   PhoneNumber phone; // create object phone

   cout << "Enter a phone number in the "
        << "form (123) 456-7890:\n";

   // cin >> phone invokes operator>> function by
   // issuing the call operator>>(cin, phone).
   cin >> phone;

   // cout << phone invokes operator<< function by
   // issuing the call operator<<(cout, phone).
   cout << "The phone number entered was:\n"
        << phone << endl;
   return 0;
}
```

Fig. 18.3 Operadores de inserción y de extracción de flujo definidos por usuario (parte 1 de 2).


```
Enter a phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was:
(800) 555-1212
```

Fig. 18.3 Operadores de inserción y de extracción de flujo definidos por usuario (parte 2 de 2).

Primero, se ejecutaría la expresión **cin >> phone1** mediante la llamada

```
operator>>(cin, phone1);
```

Esta llamada a continuación regresaría **cin** como el valor de **cin >> phone1**, por lo que la porción restante de la expresión se interpretaría solo como **cin >> phone2**.

El operador de inserción de flujo toma como argumentos una referencia **ostream** (**output**) y una referencia (**num**) a un tipo definido por usuario (**PhoneNumber**), y devuelve una referencia **ostream**. La función **operator<<** muestra objetos del tipo **PhoneNumber**.

Cuando en **main** el compilador ve la expresión

```
cout << phone
```

generará la llamada de función

```
operator<<(cout, phone);
```

La función **operator<<** muestra las porciones del número telefónico como cadenas, porque están almacenadas en formato de cadena (la función miembro **istream** de nombre **getline** almacena un carácter nulo después de que termina su entrada).

Note que las funciones **operator>>** y **operator<<** se declaran en **classPhoneNumber** como funciones no miembro, amigo. Estos operadores deben ser no miembro, porque en cada caso el objeto de la clase **PhoneNumber** aparece como el operando derecho del operador; a fin de hacer la homonimia del operador como función miembro —el operando de clase debe aparecer a la izquierda. Los operadores de entrada y de salida homónimos deben ser declarados como amigo, si han de tener acceso directo a los miembros de clase no públicos.

Observación de ingeniería de software 18.2

Se pueden añadir nuevas capacidades de entrada/salida a C++ correspondientes a tipos definidos por usuario, sin modificar las declaraciones o los miembros de datos privados correspondientes a la clase **ostream** *o a la clase* **istream**. *Esto promueve la extensibilidad del lenguaje de programación C++ uno de los aspectos más atractivos de C++.*

18.6 Homonimia de operadores unarios

Se puede hacer la homonimia de un operador unario para una clase como función miembro no estática sin argumentos, o como función no miembro con un argumento; dicho argumento debe ser un objeto de la clase o una referencia a un objeto de la clase.

Más adelante en este capítulo haremos la homonimia del operador unario ! para probar si una cadena está vacía. Al hacer la homonimia de un operador unario como !, en forma de función miembro no estática sin argumentos, si **s** es un objeto de clase, cuando el compilador vea la expresión **!s** generará la llamada **s.operator! ()**. El operando **s** es el objeto de clase s para el cual se está invocando la función miembro **operator!**:

```
class String {
public:
   int operator!() const;
   ...
};
```

Existan dos formas distintas en que se puede hacer la homonimia de un operador unario como ! en forma de función no miembro con un argumento —con un argumento que es un objeto o con un argumento que es una referencia a un objeto. Si **s** es un objeto de clase, entonces **!s** se tratará como si se hubiera escrito la llamada **operator! (s)**.

```
class String {
   friend int operator!(const String &);
   ...
};
```

Práctica sana de programación 18.5

Al hacer la homonimia de operadores unarios, es preferible hacer la funciones operador miembros de clase en vez de funciones amigo no miembro. Esta es una solución más limpia. Las funciones amigo y las clases amigo deberán evitarse, salvo que sean en lo absoluto necesarias. La utilización de amigos viola el encapsulado de una clase.

18.7 Homonimia de operadores binarios

Se puede hacer la homonimia de un operador binario como función miembro no estática con un argumento o como, función no miembro con dos argumentos (uno de dichos argumentos debiendo ser un objeto de clase o una referencia a un objeto de clase).

Más adelante en este capítulo, haremos la homonimia de **+=** para indicar concatenación de dos objetos de cadena. Al hacer la homonimia del operador binario **+=** como una función miembro no estática de una clase **String** con un argumento, si **y** y **z** son objetos de clase, entonces **y += z** será tratado como si se hubiera escrito **y.operator+=(z)**.

```
class String {
public:
   String &operator+=(const String &);
   ...
};
```

Se puede hacer la homonimia del operador binario **+=** como función no miembro con dos argumentos uno de los cuales debe ser un objeto de clase o una referencia a un objeto de clase. Si **y** y **z** son objetos de clase, entonces **y += z** será tratado como si dentro del programa la llamada **operator+=(y, z)** hubiera sido escrita


```
class String {
   friend String &operator+=(String &, const String &);
   ...
};
```

18.8 Estudio de caso: una clase Array

En C y en C++ la notación de arreglo es solo una alternativa a los apuntadores, por lo que los arreglos tienen mucho potencial para errores. Por ejemplo, un programa con facilidad puede "salirse" de un arreglo, porque C y C++ no verifican si los subíndices caen fuera del rango de un arreglo. Los arreglos del tamaño *n* deberá numerar sus elementos 0, ..., *n* - 1; no se permiten rangos con subíndices distintos. Un arreglo completo no puede ser introducido o extraído de una vez; cada elemento del arreglo debe ser leído o escrito en forma individual. No se pueden comparar dos arreglos mediante operadores de igualdad u operadores relacionales. Cuando se pasa un arreglo a una función de uso general, diseñada para manejar arreglos de cualquier tamaño, como argumento adicional deberá pasarse el tamaño del arreglo. Un arreglo no puede ser asignado a otro arreglo utilizando el operador u operadores de asignación. Estas y otras capacidades ciertamente parecerían como "naturales" para el manejo de arreglos, pero C y C++ no tienen estas capacidades. Sin embargo, C++ sí proporciona los medios para poner en práctica estas capacidades para arreglos, mediante los mecanismos de la homonimia de operadores.

En este ejemplo, desarrollamos una clase de arreglo que lleva a cabo verificación de rangos, con el fin de asegurarse que los subíndices se conservan dentro de los límites del arreglo. La clase permite que, mediante el operador de asignación, un objeto de arreglo sea asignado a otro. Los objetos de la clase de arreglo de forma automática conocen su tamaño, por lo que al pasar el arreglo a una función dicho tamaño no necesita ser pasado como argumento. Mediante los operadores de extracción y de inserción de flujo, respectivamente, es posible introducir o extraer arreglos completos. Se pueden llevar a cabo comparaciones de arreglos utilizando los operadores de igualdad == y !=. Nuestra clase de arreglo utiliza un miembro estático para llevar control del número de objetos de arreglo que en el programa han sido producidos. Este ejemplo aumentará su aprecio a la abstracción de datos. Es probable que desee sugerir varias mejoras a esta clase de arreglo. El desarrollo de clases es una actividad interesante e intelectual fascinante.

El programa de la figura 18.4 demuestra la clase **Array** y sus operadores homónimos. Primero recorramos el programa manejador en **main**. A continuación veamos la definición de clase y cada una de las funciones miembro de la clase y las definiciones de las funciones amigo. El programa produce dos objetos de clase **Array integers1**, con siete elementos, e **integers2**, con un tamaño por omisión de diez elementos (valor por omisión especificado por el constructor de **Array**). El miembro de datos estático **arrayCount** de la clase **Array** contiene el número de objetos **Array** producidos durante la ejecución del programa. La función miembro estática **getArrayCount** devuelve este valor. La función miembro **getSize** devuelve el tamaño del arreglo **integers1**. El programa extrae el tamaño del arreglo **integers1**, y a continuación extrae el arreglo mismo, utilizando el operador de inserción de flujo homónimo, para confirmar que los elementos del arreglo fueron inicializados de forma correcta por el constructor. A continuación, el tamaño del arreglo **integers2**, y después se extrae el arreglo mismo, utilizando el operador de inserción de flujo homónimo.

Al usuario se le pide que introduzca 16 enteros. Para leer dichos valores a ambos arreglos se utiliza el operador de extracción de flujo homónimo, mediante el enunciado

```
cin >> integers1 >> integers2;
```

Los primeros siete valores se almacenan en **integers1** y los restantes en **integers2**. Para confirmar que la entrada fue llevada a cabo correctamente, los dos arreglos son extraídos mediante el operador de inserción de flujo.

A continuación el programa prueba el operador de desigualdad homónimo evaluando la condición.

```
integers1 != integers2
```

y el programa informa que en efecto los arreglos no son iguales.

El programa produce un tercer arreglo, llamado **integers3**, y lo inicializa con el arreglo **integers1**. El programa extrae el tamaño del arreglo **integers3** y a continuación extrae el arreglo mismo, utilizando el operador de inserción de flujo homónimo, para confirmar que los elementos del arreglo fueron inicializados de forma correcta por el constructor.

Después el programa prueba el operador de asignación homónimo (=) con la expresión **integers1=integers2**. Ambos arreglos a continuación son impresos, para confirmar que la asignación fue correcta. Es interesante notar que **integers1** originalmente contenía 7 enteros y para contener una copia de los 10 elementos de **integers2** necesitaba ser redimensionado. Como veremos, el operador de asignación homónimo ejecuta este redimensionamiento de forma transparente para el llamador del operador.

A continuación el programa usa el operador de igualdad homónimo (==) para ver si después de la asignación los objetos **integers1=integers2** son de hecho idénticos.

Entonces el programa utiliza el operador de subíndice homónimo para hacer la referencia al elemento **integers1[5]** un elemento dentro del rango de **integers1**. Este nombre con subíndice se utiliza después como valor a la derecha (*rvalue*) para imprimir el valor en **integers1[5]**, y entonces el nombre se utiliza como valor a la izquierda (*lvalue*) en el lado izquierdo del enunciado de asignación, para recibir un nuevo valor, 1000, correspondiente a **integers1[5]**. Note que el **operator[]** provee la referencia y asigna el valor.

El programa a continuación intenta asignar le valor 1000 a **integers1[15]** un elemento fuera de rango. El programa detecta este error y termina en forma anormal.

De forma interesante, el operador de subíndice de arreglo **[]** no está restringido para uso en arreglos; puede ser utilizado para seleccionar elementos de otros tipos de clases contenedor, como son listas enlazadas, cadenas, diccionarios y demás. También, ya no es necesario que los subíndices sean enteros; se pueden utilizar caracteres o cadenas, por ejemplo.

Ahora que hemos visto cómo funciona este programa, vayamos recorriendo el encabezado de clase y las definiciones de funciones miembro. Las líneas

```
int *ptr;                    // pointer to first element of array
int size;                    // size of the array
static int arrayCount;       // # of Arrays instantiated
```

representa los miembros de datos privados de la clase. El arreglo está formado de un apuntador, **ptr** (al tipo apropiado, en este caso **int**), un miembro **size**, que indica el número de elementos en el arreglo, y un miembro estático **arrayCount**, que indica el número de objetos de arreglo que han sido producidos.

Las líneas

```
friend ostream &operator<<(ostream &, const Array &);
friend istream &operator>>(istream &, Array &);
```

declaran el operador de inserción de flujo homónimo y el operador de extracción de flujo homónimo como amigo de la clase **Array**. Cuando el compilador ve una expresión como


```
cout << arrayObject
```

invoca la función **operator<<** mediante la generación de la llamada

```
operator<<(cout, arrayObject)
```

Cuando el compilador ve una expresión como

```
cin >> arrayObject
```

invoca a la función **operator>>** mediante la generación de la llamada

```
operator>>(cin, arrayObject)
```

Notamos otra vez que esas funciones operador no pueden ser miembros de la clase **Array**, porque el objeto **Array** siempre es mencionado del lado derecho de los operadores de inserción y de extracción de flujo. La función **operator<<** imprime el número de elementos indicados por **size** en el arreglo almacenado en **ptr**. La función **operator>>** introduce directamente al arreglo apuntado por **ptr**. Cada una de estas funciones operador devuelven una referencia apropiada, para permitir llamadas concatenadas.

La línea

```
Array(int arraySize = 10)            // default constructor
```

declara al constructor por omisión correspondiente a la clase y especifica que el tamaño del arreglo se preestablece en 10 elementos. Cuando el compilador ve una declaración como

```
Array integers1(7);
```

o su forma equivalente

```
Array integers1 = 7;
```

invoca al constructor por omisión. La función miembro constructor por omisión **Array** incrementa **arrayCount**, copia el argumento en el miembro de datos **size**, utiliza a **new** para obtener el espacio para contener la representación interna de este arreglo y asigna el apuntador regresado por **new** al miembro de datos **ptr**, utiliza **assert** para probar que **new** tuvo éxito, y a continuación utiliza un ciclo **for** para inicializar a cero todos los elementos del arreglo. Sería perfectamente razonable tener una clase **Array** que no inicializara sus miembros si, por ejemplo, dichos miembros tuvieran que ser leídos en algún momento posterior.

La línea

```
Array(const Array &);                          // copy constructor
```

es un *constructor de copia*. Inicializa un objeto **Array** haciendo una copia de un objeto existente **Array**. Dicha copia deberá ser efectuada con cuidado, a fin de evitar el error de que ambos objetos **Array** apunten al mismo almacenamiento asignado dinámicamente, lo que sería exactamente el mismo problema que ocurriría mediante la copia a nivel de miembro por omisión. Los constructores de copia son invocados siempre que se requiera una copia de un objeto, como es tratándose de llamada por valor, al devolver un objeto de una función llamada, o al inicializar un objeto como una copia de otro objeto de la misma clase. En una declaración se llama al constructor de copia cuando un objeto de la clase **Array** es producido e inicializado con otro objeto de la clase **Array**, como ocurre en la declaración siguiente:

```
Array integers3(integers1);
```

o en la declaración equivalente

```
Array integers3 = integers1);
```

La función miembro constructor de copia **Array** incrementa **arrayCount**, copia **size** del arreglo utilizado para la inicialización en el miembro de datos **size**, utiliza **new** para obtener el espacio para contener la representación interna de este arreglo y asigna el apuntador regresado por **new** al miembro de datos **ptr**, utiliza **assert** para probar que **new** tuvo éxito, y después utiliza un ciclo **for** para copiar todos los elementos del arreglo inicializador a este arreglo. Es importante notar que si el constructor de copia simplemente hubiera copiado **ptr** del objeto fuente al objeto destino **ptr**, entonces ambos objetos hubieran señalado al mismo almacenamiento asignado dinámicamente. El primer destructor a ejecutarse hubiera entonces borrado el almacenamiento dinámicamente asignado, y el **ptr** del otro objeto hubiera quedado sin definición, una situación que seguramente causaría un error serio en tiempo de ejecución.

Práctica sana de programación 18.6

Un destructor, el operador de asignación, y un constructor de copia para una clase por lo general se proporcionan en grupo.

La línea

```
~Array();                                   // destructor
```

declara al destructor de la clase. Cuando se termine la vida de un objeto de la clase **Array**, el destructor será invocado en forma automática. El destructor decrementa **arrayCount** y a continuación utiliza **delete** para recuperar el almacenamiento dinámico creado por **new** en el constructor.

La línea

```
int getSize() const;                        // return size
```

declara a una función que lee el tamaño del arreglo.

La línea

```
const Array &operator=(const Array &);      // assign arrays
```

declara la función operador de asignación homónimo para la clase. Cuando el compilador ve una expresión como

```
integers1 = integers2;
```

invoca a la función **operator=** mediante la generación de la llamada

```
integers1.operator=(integers2)
```

La función miembro **operator=** prueba buscando autoasignación. Si se está intentando hacer una autoasignación, la asignación es pasada por alto (es decir, el objeto ya es él mismo). Si no se trata de una autoasignación, entonces la función miembro utiliza **delete** para recuperar el espacio originalmente asignado en el arreglo destino, copia **size** del arreglo fuente a **size** del arreglo destino, utiliza **new** para asignar dicha cantidad de espacio para el arreglo destino y coloca

el apuntador devuelto por **new** en el miembro **ptr** del arreglo, utiliza **assert** para verificar que **new** tuvo éxito, y después utiliza un ciclo **for** para copiar los elementos del arreglo, correspondientes al arreglo fuente, al arreglo destino. Independiente de que se trate de una autoasignación o no, a continuación la función miembro devuelve el objeto actual (es decir, ***this**) como referencia constante; esto habilita las asignaciones **Array** concatenadas como **x = y = z**. Si hubiera sido omitida la prueba de autoasignación, la función miembro hubiera empezado borrando el espacio del arreglo destino. Dado que en una autoasignación esto es también el arreglo fuente, el arreglo hubiera sido destruido.

Error común de programación 18.4

No probar la existencia y evitar la autoasignación cuando se hace la homonimia del operador de asignación de una clase que incluya un apuntador a almacenamiento dinámico.

Error común de programación 18.5

No proporcionar un operador de asignación homónimo y un constructor de copia para una clase, cuando los objetos de dicha clase contengan apuntadores a almacenamiento dinámicamente asignado.

Observación de ingeniería de software 18.3

Es posible evitar que un objeto de clase sea asignado a otro. Esto se lleva a cabo definiendo el operador de asignación como miembro privado de la clase.

La línea

```
int operator==(const Array &) const: // compare equal
```

declara al operador de igualdad homónimo (==) para la clase. Cuando el compilador ve la expresión

```
integers1 == integers2
```

en **main**, invoca la función miembro **operator==** mediante la generación de la llamada

```
integers1.operator==(integers2)
```

La función miembro **operator==** de inmediato devuelve 0 (falso) si son diferentes los miembros **size** de los arreglos. De lo contrario, la función miembro compara cada par de elementos. Si todos son iguales, devuelve 1 (verdadero). El primer par de elementos que sea distinto hará que se devuelva de inmediato 0.

La línea

```
int operator!=(const Array &) const; // compare !equal
```

declara el operador de desigualdad homónimo (!=) para la clase. Invocará a la función miembro **operator!=** y operará en forma similar a la función de operador de igualdad homónimo.

La línea

```
int &operator[](int);                       //subscript operator
```

declara al operador de subíndice homónimo correspondiente a la clase. Cuando el compilador ve la expresión

```
integers1[5]
```

en **main**, el compilador invoca la función miembro homónima **operator[]** generando la llamada

```
integers1.operator[](5)
```

La función miembro **operator[]** prueba si el subíndice está dentro de rango, y si no es así, el programa termina en forma anormal. Si el subíndice está dentro de rango, devuelve el elemento apropiado del arreglo como referencia, de manera que pueda ser utilizado como valor a la izquierda (*lvalue*) (por ejemplo, en el lado izquierdo de un enunciado de asignación).

La línea

```
static int getArrayCount();              // return count of Arrays
```

declara la función miembro estática **getArrayCount** que devuelve el valor del miembro de datos estático **arrayCount**, aún si no existen objetos de la clase **Array**.

```
// ARRAY1.H
// Simple class Array (for integers)
#ifndef ARRAY1_H
#define ARRAY1_H

#include <iostream.h>

class Array {
   friend ostream &operator<<(ostream &, const Array &);
   friend istream &operator>>(istream &, Array &);
public:
   Array(int = 10);                          // default constructor
   Array(const Array &);                     // copy constructor
   ~Array();                                 // destructor
   int getSize() const;                      // return size
   const Array &operator=(const Array &);    // assign arrays
   int operator==(const Array &) const;      // compare equal
   int operator!=(const Array &) const;      // compare !equal
   int &operator[](int);                     // subscript operator
   static int getArrayCount();               // return count of
                                             // arrays instantiated
private:
   int *ptr; // pointer to first element of array
   int size; // size of the array
   static int arrayCount;  // # of Arrays instantiated
};

#endif
```

Fig. 18.4 Definición de la clase **Array** (parte 1 de 7).


```
// ARRAY1.CPP
// Member function definitions for class Array
#include <iostream.h>
#include <stdlib.h>
#include <assert.h>
#include "array1.h"

// Initialize static data member at file scope
int Array::arrayCount = 0;    // no objects yet

// Return the number of Array objects instantiated
int Array::getArrayCount() { return arrayCount; }

// Default constructor for class Array
Array::Array(int arraySize)
{
   ++arrayCount;                    // count one more object
   size = arraySize;                // default size is 10
   ptr = new int[size];             // create space for array
   assert(ptr != 0);  // terminate if memory not allocated

   for (int i = 0; i < size; i++)
      ptr[i] = 0;                   // initialize array
}

// Copy constructor for class Array
Array::Array(const Array &init)
{
   ++arrayCount;                    // count one more object
   size = init.size;                // size this object
   ptr = new int[size];             // create space for array
   assert(ptr != 0);  // terminate if memory not allocated

   for (int i = 0; i < size; i++)
      ptr[i] = init.ptr[i];         // copy init into object
}

// Destructor for class Array
Array::~Array()
{
   --arrayCount;                    // one fewer objects
   delete [] ptr;                   // reclaim space for array
}

// Get the size of the array
int Array::getSize() const { return size; }

// Overloaded subscript operator
int &Array::operator[](int subscript)
{
   // check for subscript out of range error
   assert(0 <= subscript && subscript < size);

   return ptr[subscript];   // reference return creates lvalue
}
```

Fig. 18.4 Definiciones de funciones miembro para la clase **Array** (parte 2 de 7).

```
// Determine if two arrays are equal and
// return 1 if true, 0 if false.
int Array::operator==(const Array &right) const
{
   if (size != right.size)
      return 0;     // arrays of different sizes

   for (int i = 0; i < size; i++)
      if (ptr[i] != right.ptr[i])
         return 0; // arrays are not equal

   return 1;        // arrays are equal
}

// Determine if two arrays are not equal and
// return 1 if true, 0 if false.
int Array::operator!=(const Array &right) const
{
   if (size != right.size)
      return 1;     // arrays of different sizes

   for (int i = 0; i < size; i++)
      if (ptr[i] != right.ptr[i])
         return 1;       // arrays are not equal

   return 0;        // arrays are equal
}

// Overloaded assignment operator
const Array &Array::operator=(const Array &right)
{
   if (&right != this) {     // check for self-assignment
      delete [] ptr;         // reclaim space
      size = right.size;     // resize this object
      ptr = new int[size];   // create space for array copy
      assert(ptr != 0);      // terminate if memory not allocated

      for (int i = 0; i < size; i++)
         ptr[i] = right.ptr[i];  // copy array into object
   }

   return *this;    // enables x = y = z;
}

// Overloaded input operator for class Array;
// inputs values for entire array.
istream &operator>>(istream &input, Array &a)
{
   for (int i = 0; i < a.size; i++)
      input >> a.ptr[i];

   return input;    // enables cin >> x >> y;
}
```

Fig. 18.4 Definiciones de funciones miembro para la clase **Array** (parte 3 de 7).


```
// Overloaded output operator for class Array
ostream &operator<<(ostream &output, const Array &a)
{
   for (int i = 0; i < a.size; i++) {
      output << a.ptr[i] << ' ';

      if ((i + 1) % 10 == 0)
         output << endl;
   }

   if (i % 10 != 0)
      output << endl;

   return output;    // enables cout << x << y;
}
```

Fig. 18.4 Definiciones de función miembro para la clase **Array** (parte 4 de 7.

```
// FIG18_4.CPP
// Driver for simple class Array
#include <iostream.h>
#include "array1.h"

main()
{
   // no objects yet
   cout << "# of arrays instantiated = "
        << Array::getArrayCount() << '\n';

   // create two arrays and print Array count
   Array integers1(7), integers2;
   cout << "# of arrays instantiated = "
        << Array::getArrayCount();

   // print integers1 size and contents
   cout << "\n\nSize of array integers1 is "
        << integers1.getSize()
        << "\nArray after initialization:\n"
        << integers1;

   // print integers2 size and contents
   cout << "\nSize of array integers2 is "
        << integers2.getSize()
        << "\nArray after initialization:\n"
        << integers2;

   // input and print integers1 and integers2
   cout << "\nInput 17 integers:\n";
   cin >> integers1 >> integers2;
   cout << "After input, the arrays contain:\n"
        << "integers1: " << integers1
        << "integers2: " << integers2;
```

Fig. 18.4 Manejador para la clase **Array** (parte 5 de 7).

```
   // create array integers3 using integers1 as an
   // initializer; print size and contents
   Array integers3(integers1);

   cout << "\nSize of array integers3 is "
        << integers3.getSize()
        << "\nArray after initialization:\n"
        << integers3;

   // use overloaded inequality (!=) operator
   cout << "\nEvaluating: integers1 != integers2\n";
   if (integers1 != integers2)
      cout << "They are not equal\n\n";

   // use overloaded assignment (=) operator
   cout << "Assigning integers2 to integers1:\n";
   integers1 = integers2;
   cout << "integers1: " << integers1
        << "integers2: " << integers2;

   // use overloaded equality (==) operator
   cout << "\nEvaluating: integers1 == integers2\n";
   if (integers1 == integers2)
      cout << "They are equal\n\n";

   // use overloaded subscript operator to create rvalue
   cout << "integers1[5] is " << integers1[5] << endl;

   // use overloaded subscript operator to create lvalue
   cout << "Assigning 1000 to integers1[5]\n";
   integers1[5] = 1000;
   cout << "integers1: " << integers1;

   // attempt to use out of range subscript
   cout << "\nAttempt to assign 1000 to integers1[15]\n";
   integers1[15] = 1000;  // ERROR: out of range

   return 0;
}
```

Fig. 18.4 Manejador para la clase **Array** (parte 6 de 7).

18.9 Conversión entre tipos

La mayor parte de los programas procesan información en una variedad de tipos. Algunas veces todas las operaciones "se mantienen dentro de un tipo". Por ejemplo, añadir un entero a un entero produce un entero (siempre y cuando el resultando no sea tan grande que no pueda ser representado como un entero). Pero, a menudo es necesario convertir datos de un tipo a datos de otro tipo. Esto puede ocurrir en asignaciones, en cálculos, al pasar valores a funciones, y al devolver valores de funciones. El compilador sabe cómo llevar a cabo conversiones entre tipos incorporados. Los programadores pueden obligar a conversiones entre tipos incorporados mediante las conversiones explícitas casting.


```
# of arrays instantiated = 0
# of arrays instantiated = 2

Size of array integers1 is 7
Array after initialization:
0 0 0 0 0 0 0

Size of array integers2 is 10
Array after initialization:
0 0 0 0 0 0 0 0 0 0

Input 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
After input, the arrays contain:
integers1: 1 2 3 4 5 6 7
integers2: 8 9 10 11 12 13 14 15 16 17

Size of array integers3 is 7
Array after initialization:
1 2 3 4 5 6 7

Evaluating: integers1 != integers2
They are not equal

Assigning integers2 to integers1:
integers1: 8 9 10 11 12 13 14 15 16 17
integers2: 8 9 10 11 12 13 14 15 16 17

Evaluating: integers1 == integers2
They are equal

integers1[5] is 13
Assigning 1000 to integers1[5]
integers1: 8 9 10 11 12 13 14 15 16 17

Attempt to assign 1000 to integers1[15]
Assertion failed: 0 <= subscript && subscript < size,
   file ARRAY1.CPP, line 98
Abnormal program termination
```

Fig. 18.4 Salida del manejador para la clase **Array** (parte 7 de 7).

Pero ¿qué pasa en relación con los tipos definidos por usuario? El compilador no puede saber en forma automática cómo convertir entre tipos definidos por usuario y tipos incorporados. El programador deberá especificar en forma explícita cómo deberán ocurrir dichas conversiones. Estas conversiones pueden ser ejecutadas mediante *constructores de conversión*, es decir, constructores de un solo argumento, que sólo conviertan objetos de otros tipos en objetos de una clase particular. Más adelante en este capítulo utilizaremos un constructor de conversión para convertir las cadenas **char** * normales en objetos **String** de C++.

Un *operador de conversión* (también conocido como *operador de conversión explicita cast*) puede ser utilizado para convertir un objeto de una clase en un objeto de otra clase o en un objeto

de un tipo incorporado. Este operador de conversión debe ser una función miembro no estática; este tipo de operador de conversión no puede ser una función amigo.

El prototipo de función

```
operator char *() const;
```

declara una función operador de conversión explicita cast homónima para crear un objeto temporal **char*** partiendo de un objeto de un tipo definido por usuario. Una función operador de conversión explicita cast homónima no especifica un tipo de regreso —el tipo de regreso es el tipo al cual se está convirtiendo el objeto. Si **s** es un objeto de clase, cuando el compilador vea la expresión **(char *) s** el compilador generará la llamada **s.operator char * ()**. El operando **s** es el objeto de clase **s** para el cual se invoca la función miembro **operator char ***.

Las funciones operador de conversión explícita cast homónimas pueden ser definidas para la conversión de objetos de tipos definidos por usuario a tipos incorporados o a objetos de otros tipos definidos por usuario. Los prototipos de función

```
operator int() const;
operator otherClass() const;
```

declaran funciones operador de conversión explícita cast homónimas para la conversión de un objeto de un tipo definido por usuario a un entero o para convertir un objeto de un tipo definido por usuario a un objeto de un tipo definido por usuario **otherClass**.

Una de las características maravillosas de los operadores de conversión explícita (cast) y de los constructores de conversión es que, cuando es necesario, el compilador puede llamar estas funciones en forma automática para crear objetos temporales. Por ejemplo, si en un programa aparece un objeto **s** de una clase **String** definida por usuario, en una posición donde se espera un **char *** normal, tal como

```
cout << s;
```

el compilador llama la función de operador de conversión explícita (cast) homónima **operator char *** para convertir el objeto en un **char *** y utiliza el **char *** resultante dentro de la expresión.

18.10 Estudio de caso: una clase String

Como un ejercicio de recapitulación a nuestro estudio de la homonimia construiremos una clase que maneje la creación y manipulación de cadenas. Ni C ni C++ tienen un tipo de datos de cadenas incorporado. Pero C++ nos permite añadir como clase a nuestro propio tipo de cadenas, y —mediante los mecanismos de la homonimia— proporciona un conjunto de operadores para manejar convenientemente las cadenas. Las varias partes de la figura 18.5 incluyen un encabezado de clase, definiciones de funciones miembros, y un programa manejador para probar nuestra nueva clase **String**.

Primero, presentaremos el encabezado para la clase **String**. Analizamos los datos privados utilizados para representar objetos **String**. A continuación, recorremos la interfaz pública de la clase, analizando cada uno de los servicios que la misma proporciona.

A continuación, recorreremos el programa manejador en **main**. Analizaremos el estilo de codificación al cual "aspiramos", es decir, los tipos de expresiones operador intensivo que nos gustaría tener la capacidad de escribir, con objetos de nuestra nueva clase **String** y con el conjunto de operadores homónimos de la clase.

A continuación, analizamos las definiciones de funciones miembro correspondientes a la clase **String**. Para cada uno de los operadores homónimos, mostramos el código en el programa manejador que invoca a la función operador homónima, y explicamos cómo funciona la función operador homónima.

Empezamos con la representación interna de un **String**. Las líneas

```
private:
   char *sPtr;                    // pointer to start of string
   int length;                    // string length
```

declaran los miembros de datos privados de la clase. Un objeto **String** tiene un apuntador a su almacenamiento dinámico asignado representando la cadena de caracteres, y tiene un campo de longitud que representa el número de caracteres en la cadena, excluyendo el carácter nulo al final de la cadena de caracteres.

Ahora recorramos el archivo de cabecera de la clase **String** de la figura 18.5. Las líneas

```
friend ostream &operator<<(ostream &, const String &);
friend istream &operator>>(istream &, String &);
```

declaran la función operador de inserción de flujo homónima **operator<<** y la función operador de extracción de flujo homónima **operator>>** como amigo de la clase. La puesta en práctica de las funciones anteriores es simple.

La línea

```
String(const char * = ""); // conversion constructor
```

declara un *constructor de conversión*. Este constructor toma un argumento **char *** (que por omisión es la cadena vacía) y produce un objeto **String** que incluye la misma cadena de caracteres. Cualquier constructor de un argumento puede ser considerado como si fuera un constructor de conversión. Como veremos, estos constructores son útiles cuando estamos haciendo asignaciones de cadenas de caracteres a objetos **String**. El constructor de conversión convierte la cadena convencional en un objeto **String**, que a continuación es asignado al objeto destino **String**. La disponibilidad de este constructor de conversión significa que para asignar en forma específica las cadenas de caracteres a los objetos **String** no es necesario proveer un operador de asignación homónimo. El compilador invoca en forma automática el constructor de conversión, a fin de crear un objeto temporal **String** que contenga la cadena de caracteres. Entonces, se invoca el operador de asignación homónimo para asignar el objeto temporal **String** a otro objeto **String**. Note que cuando C++ utiliza de esta forma un constructor de conversión, sólo puede aplicar un constructor para intentar hacer coincidir las necesidades del operador de asignación homónimo. No es posible hacer coincidir las necesidades del operador homónimo llevando a cabo una serie de conversiones implícitas, definidas por el usuario. El constructor de conversión **String** podría ser invocado en una declaración como **String s1 ("happy")**. La función miembro constructor de conversión calcula la longitud de la cadena de caracteres y asigna este cálculo al miembro de datos privado **length**, utiliza **new** para asignar suficiente espacio al miembro de datos privado **sPtr**, utiliza **assert** para probar que **new** tuvo éxito, y si así fue, utiliza **strcpy** para copiar la cadena de caracteres al objeto.

La línea

```
String(const String &);              // copy constructor
```

es un constructor de copia. Inicializa un objeto **String** haciendo una copia de un objeto existente **String**. Esta copia deberá ser efectuada con cuidado para evitar el problema de tener ambos objetos **String** señalando al mismo almacenamiento asignado dinámicamente, exactamente el problema que ocurriría tratándose de una copia a nivel de miembro por omisión. El constructor de copia opera de forma similar al constructor de conversión, excepto que simplemente copia el miembro **length** del objeto fuente **String** al objeto destino **String**. Note que el constructor de copia genera un nuevo espacio para la cadena de caracteres interna del objeto destino. Si solo copiara **sPtr** en el objeto fuente al **sPtr** del objeto destino, entonces ambos objetos señalarían al mismo almacenamiento dinámico asignado. El primer destructor que se ejecutase borraría entonces el almacenamiento dinámicamente asignado y el **sPtr** del otro objeto quedaría entonces sin definir, una situación que probablemente causaría un error serio en tiempo de ejecución.

La línea

```
~String();                                  // destructor
```

declara al destructor correspondiente a la clase **String**. El destructor utiliza **delete** para reclamar el almacenamiento dinámico obtenido por **new** en los constructores para proveer el espacio correspondiente a la cadena de caracteres.

La línea

```
const String &operator=(const String &);    // assignment
```

declara al operador de asignación homónimo. Cuando el compilador ve una expresión como **string1 = string2**, genera la llamada de función

```
string1.operator=(string2);
```

La función operador de asignación homónimo **operator=** prueba por si existe autoasignación, exactamente de la misma forma que lo hizo el constructor de copia. Si se trata de una autoasignación, la función sólo regresa, dado que el objeto ya es él mismo. Si esta prueba se omitiera, la función de inmediato borraría el espacio en el objeto destino y se perdería la cadena de caracteres. Suponiendo que no se trata de una autoasignación, la función no borra el espacio, copia el campo de longitud del objeto fuente al objeto destino, genera un nuevo espacio para el objeto destino, utiliza **assert** para determinar si **new** tuvo éxito, y entonces utiliza **strcpy** para copiar la cadena de caracteres del objeto fuente al objeto destino. Independiente de que se trate o no de una autoasignación, se devuelve *** this** para permitir asignaciones concatenadas.

La línea

```
String &operator+=(const String &);          // concatenation
```

declara al operador de concatenación de cadenas homónimo. Cuando el compilador ve la expresión **s1 += s2** en **main**, genera la llamada de función **s1.operator+=(s2)**. La función **operator+=** crea un apuntador temporal para contener el apuntador de cadena de caracteres del objeto actual, en tanto pueda ser borrada la memoria de la cadena de caracteres, calcula la longitud combinada de la cadena concatenada, utiliza **new** para reservar espacio para la cadena, utiliza **assert** para probar si **new** tuvo éxito, utiliza **strcpy** para copiar la cadena original al nuevo espacio asignado, utiliza **strcat** para concatenar la cadena de caracteres del objeto fuente al nuevo espacio asignado, utiliza **delete** para recuperar el espacio ocupado por la cadena de

caracteres original de este objeto, y devuelve *** this** como un **String &** para permitir la concatenación de los operadores **+=**.

¿Será necesario tener un segundo operador de concatenación homónimo para poder hacer la concatenación de un **String** y de un **char ***? No, el constructor de conversión **const char *** convierte una cadena convencional en un objeto **String**, que a continuación coincide con el operador de concatenación homónimo. C++ puede llevar a cabo estas conversiones en un nivel de profundidad, para facilitar una coincidencia. C++ también puede, antes de llevar a cabo la conversión entre un tipo incorporado y una clase, ejecutar una conversión implícita definida por compilador entre tipos incorporados.

Sugerencia de rendimiento 18.2

Usar el operador de concatenación homónimo += que toma un solo argumento del tipo **const char *** *da resultados más eficaces que primero tener que efectuar la conversión implícita y a continuación, la concatenación. Las conversiones implícitas requieren de menos código y causan menos errores.*

La línea

```
int operator!() const;                      // is String empty?
```

declara el operador de negación homónimo. Este operador se usa por lo común con las clases de cadenas, para probar si una cadena está vacía. Por ejemplo, cuando el compilador ve la expresión **!string1**, genera la llamada de función

```
string1.operator!()
```

Esta función sencillamente devuelve el resultado de probar si **length** es igual a cero.

Las líneas

```
int operator==(const String &) const;    // test s1 == s2
int operator!=(const String &) const;    // test s1 != s2
int operator<(const String &) const;     // test s1 < s2
int operator>(const String &) const;     // test s1 > s2
int operator>=(const String &) const;    // test s1 >= s2
int operator<=(const String &) const;    // test s1 <= s2
```

declaran los operadores de igualdad homónimos y los operadores relacionales homónimos, correspondientes a la clase **String**. Todos ellos son similares, por lo que analizaremos un ejemplo, simplemente la homonimia del operador **>=**. Cuando el compilador ve la expresión **string1 >= string2**, generará la llamada de función

```
string1.operator>=(string2)
```

misma que devolverá **1** (verdadero) si **string1** es mayor que o igual a **string2**. Cada uno de estos operadores utiliza **strcmp** para comparar las cadenas de caracteres en los objetos **String**.

La línea

```
char &operator[](int);                      // return char reference
```

declara el operador de subíndice homónimo. Cuando el compilador ve una expresión como **string1[0]**, generará la llamada de función **string1.operator[](0)**. La función **ope-**

rator[] primero utiliza **assert** para llevar a cabo una verificación de rango sobre el subíndice; si el subíndice está fuera de rango, el programa imprimirá un mensaje de error y terminará en forma anormal. Si el subíndice está dentro de rango, el operador **operator[]** devuelve el carácter apropiado como un **char &** del objeto **String**; este **char &** puede ser utilizado como un valor a la izquierda (*lvalue*) para modificar el carácter designado del objeto **String**.

La línea

```
String &operator() (int, int);    // return a substring
```

declara el *operador de llamada de función homónimo*. En las clases de cadenas, es común hacer la homonimia de este operador para seleccionar una subcadena de un objeto **String**. Los dos parámetros enteros especifican la posición inicial y la longitud de la subcadena que se está seleccionando de **String**. Si la posición inicial está fuera de rango o si es negativa la longitud de la subcadena, se generará un mensaje de error. Por regla convencional, si la longitud de la subcadena es 0, entonces la subcadena será seleccionada hasta el final del objeto **String**. Por

```
// STRING2.H
// Definition of a String class
#ifndef STRING1_H
#define STRING1_H

#include <iostream.h>

class String {
   friend ostream &operator<<(ostream &, const String &);
   friend istream &operator>>(istream &, String &);
public:
   String(const char * = ""); // conversion constructor
   String(const String &);    // copy constructor
   ~String();                 // destructor
   const String &operator=(const String &);  // assignment
   String &operator+=(const String &);    // concatenation
   int operator!() const;                 // is String empty?
   int operator==(const String &) const;  // test s1 == s2
   int operator!=(const String &) const;  // test s1 != s2
   int operator<(const String &)  const;  // test s1 < s2
   int operator>(const String &)  const;  // test s1 > s2
   int operator>=(const String &) const;  // test s1 >= s2
   int operator<=(const String &) const;  // test s1 <= s2
   char &operator[](int);         // return char reference
   String &operator()(int, int); // return a substring
   int getLength() const;        // return string length
private:
   char *sPtr;                   // pointer to start of string
   int length;                   // string length
};

#endif
```

Fig. 18.5 Definición de una clase **String** básica (parte 1 de 8).

ejemplo, suponga que **string1** es un objeto **String** que contiene la cadena de caracteres **"AEIOU"**. Cuando el compilador ve la expresión **string1(2, 2)**, genera la llamada de función **string1.operator() (2, 2)**. Cuando esta llamada se ejecuta, se produce un objeto temporal **String**, que contiene la cadena **"IO"** y devuelve una referencia a este objeto. Hacer la homonimia del operador de llamada de función **()** resulta poderosa porque las funciones pueden tomar listas de parámetros arbitrariamente largas y complejas. Por lo tanto, podemos utilizar esta capacidad para muchos fines interesantes. Otro uso del operador de llamada de función es una notación alterna de subscripción de los arreglos: en vez de utilizar la notación de dobles corchetes de C, que resultan torpes tratándose de dobles arreglos, como en **a[b][c]**, por ejemplo, algunos programadores prefieren hacer la homonimia del operador de llamada de función para permitir la notación **a(b, c)**. El operador de llamada de función homónimo solamente puede ser una función miembro no estática. Este operador se utiliza sólo cuando "el nombre de función" es un objeto de la clase **String**.

La línea

```
int getLength() const;            // return string length
```

declara una función que devuelve la longitud del objeto **String**. Note que esta función obtiene la longitud, regresando el valor de los datos privados de la clase **String**.

Llegado a este punto, el lector deberá recorrer el código en **main**, examinar la ventana de salida, y verificar el uso de cada operador homónimo.

```
// STRING2.CPP
// Member function definitions for class String

#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "string2.h"

// Conversion constructor: Convert char * to String
String::String(const char *s)
{
   cout << "Conversion constructor: " << s << endl;
   length = strlen(s);              // compute length
   sPtr = new char[length + 1]; // allocate storage
   assert(sPtr != 0);  // terminate if memory not allocated
   strcpy(sPtr, s);                 // copy literal to object
}

// Copy constructor
String::String(const String &copy)
{
   cout << "Copy constructor: " << copy.sPtr << endl;
   length = copy.length;            // copy length
   sPtr = new char[length + 1]; // allocate storage
   assert(sPtr != 0);               // ensure memory allocated
   strcpy(sPtr, copy.sPtr);         // copy string
}
```

Fig. 18.5 Definiciones de funciones miembro para la clase **String** (parte 2 de 8).

```
// Destructor
String::~String()
{
   cout << "Destructor: " << sPtr << endl;
   delete [] sPtr;               // reclaim string
}

// Overloaded = operator; avoids self assignment
const String &String::operator=(const String &right)
{
   cout << "operator= called" << endl;

   if (&right != this) {            // avoid self assignment
      delete [] sPtr;               // prevents memory leak
      length = right.length;        // new String length
      sPtr = new char[length + 1]; // allocate memory
      assert(sPtr != 0);            // ensure memory allocated
      strcpy(sPtr, right.sPtr);     // copy string
   }
   else
      cout << "Attempted assignment of a String to itself\n";

   return *this;   // enables concatenated assignments
}

// Concatenate right operand to this object and
// store in this object.
String &String::operator+=(const String &right)
{
   char *tempPtr = sPtr;         // hold to be able to delete
   length += right.length;       // new String length
   sPtr = new char[length + 1]; // create space
   assert(sPtr != 0);   // terminate if memory not allocated
   strcpy(sPtr, tempPtr);        // left part of new String
   strcat(sPtr, right.sPtr);     // right part of new String
   delete [] tempPtr;            // reclaim old space
   return *this;                 // enables concatenated calls
}

// Is this String empty?
int String::operator!() const { return length == 0; }

// Is this String equal to right String?
int String::operator==(const String &right) const
{ return strcmp(sPtr, right.sPtr) == 0; }

// Is this String not equal to right String?
int String::operator!=(const String &right) const
{ return strcmp(sPtr, right.sPtr) != 0; }

// Is this String less than right String?
int String::operator<(const String &right) const
{ return strcmp(sPtr, right.sPtr) < 0; }
```

Fig. 18.5 Definiciones de funciones miembro para la clase **String** (parte 3 de 8).


```
// Is this String greater than right String?
int String::operator>(const String &right) const
{ return strcmp(sPtr, right.sPtr) > 0; }

// Is this String greater than or equal to right String?
int String::operator>=(const String &right) const
{ return strcmp(sPtr, right.sPtr) >= 0; }

// Is this String less than or equal to right String?
int String::operator<=(const String &right) const
{ return strcmp(sPtr, right.sPtr) <= 0; }

// Return a reference to a character in a String.
char &String::operator[](int subscript)
{
   // First test for subscript out of range
   assert(subscript >= 0 && subscript < length);

   return sPtr[subscript];  // creates lvalue
}

// Return a substring beginning at index and
// of length subLength as a reference to a String object.
String &String::operator()(int index, int subLength)
{
   // ensure index is in range and substring length >= 0
   assert(index >= 0 && index < length && subLength >= 0);

   String *subPtr = new String;   // empty String
   assert(subPtr != 0);      // ensure new String allocated

   // determine length of substring
   int size;

   if ((subLength == 0) || (index + subLength > length))
      size = length - index + 1;
   else
      size = subLength + 1;

   // allocate memory for substring
   delete subPtr->sPtr;
   subPtr->length = size;
   subPtr->sPtr = new char[size];
   assert(subPtr->sPtr != 0);  // ensure space allocated

   // copy substring into new String
   for (int i = index, j = 0; i < index + size - 1; i++, j++)
      subPtr->sPtr[j] = sPtr[i];

   subPtr->sPtr[j] = '\0';  // terminate the new String
   return *subPtr;          // return new String
}
```

Fig. 18.5 Definiciones de funciones miembro para la clase **String** (parte 4 de 8).

```
// Return string length
int String::getLength() const { return length; }

// Overloaded output operator
ostream &operator<<(ostream &output, const String &s)
{
   output << s.sPtr;
   return output;    // enables concatenation
}

// Overloaded input operator
istream &operator>>(istream &input, String &s)
{
   static char temp[100];    // buffer to store input

   input >> temp;
   s = temp;          // use String class assignment operator
   return input;      // enables concatenation
}
```

Fig. 18.5 Definiciones de funciones miembro para la clase **String** (parte 5 de 8).

```
// FIG18_5.CPP
// Driver for class String
#include <iostream.h>
#include "string2.h"

main()
{
   String s1("happy"), s2(" birthday"), s3;

   // test overloaded equality and relational operators
   cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
        << "\"; s3 is empty\n"
        << "The results of comparing s2 and s1:"
        << "\ns2 == s1 yields " << (s2 == s1)
        << "\ns2 != s1 yields " << (s2 != s1)
        << "\ns2 >  s1 yields " << (s2 > s1)
        << "\ns2 <  s1 yields " << (s2 < s1)
        << "\ns2 >= s1 yields " << (s2 >= s1)
        << "\ns2 <= s1 yields " << (s2 <= s1) << "\n\n";

   // test overloaded String empty (!) operator
   cout << "Testing !s3:\n";
   if (!s3) {
      cout << "s3 is empty; assigning s1 to s3;\n";
      s3 = s1;                // test overloaded assignment
      cout << "s3 is \"" << s3 << "\"\n\n";
   }
```

Fig. 18.5 Manejador para probar la clase **String** (parte 6 de 8).


```
   // test overloaded String concatenation operator
   cout << "s1 += s2 yields s1 = ";
   s1 += s2;                  // test overloaded concatenation
   cout << s1 << "\n\n";

   // test conversion constructor
   cout << "s1 += \" to you\" yields\n";
   s1 += " to you";           // test conversion constructor
   cout << "s1 = " << s1 << "\n\n";

   // test overloaded function call operator () for substring
   cout << "The substring of s1 starting at\n"
        << "location 0 for 14 characters, s1(0, 14), is: "
        << s1(0, 14) << "\n\n";

   // test substring "to-end-of-String" option
   cout << "The substring of s1 starting at\n"
        << "location 15, s1(15, 0), is: "
        << s1(15, 0) << "\n\n";  // 0 means "to end of string"

   // test copy constructor
   String *s4Ptr = new String(s1);
   cout << "*s4Ptr = " << *s4Ptr << "\n\n";

   // test assignment (=) operator with self-assignment
   cout << "assigning *s4Ptr to *s4Ptr\n";
   *s4Ptr = *s4Ptr;           // test overloaded self-assignment
   cout << "*s4Ptr = " << *s4Ptr << endl;

   // test destructor
   delete s4Ptr;

   // test using subscript operator to create lvalue
   s1[0] = 'H';
   s1[6] = 'B';
   cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
        << s1 << "\n\n";

   // test subscript out of range
   cout << "Attempt to assign 'd' to s1[30] yields:\n";
   s1[30] = 'd';      // ERROR: subscript out of range

   return 0;
}
```

Fig. 18.5 Manejador para probar la clase **String** (parte 7 de 8).

18.11 Homonimia de ++ y --

Todos los operadores de incremento y de decremento —preincremento, postincremento, predecremento y postdecremento pueden tener homónimos. Veremos cómo el compilador puede conocer cual es la versión prefija o postfija del operador de incremento o de decremento que está siendo utilizada.

```
Conversion constructor: happy
Conversion constructor: birthday
Conversion constructor:
s1 is "happy"; s2 is "birthday"; s3 is empty
The results of comparing s2 and s1:
s2 == s1 yields 0
s2 != s1 yields 1
s2 >  s1 yields 0
s2 <  s1 yields 1
s2 >= s1 yields 0
s2 <= s1 yields 1

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += "to you" yields
Conversion constructor: to you
Destructor: to you
s1 = happy birthday to you

Conversion constructor:
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is: happy birthday

Conversion constructor:
The substring of s1 starting at
location 15, s1(15, 0), is: to you

Copy constructor: happy birthday to you
*s4Ptr = happy birthday to you

ssigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:
Assertion failed: subscript >= 0 && subscript < length,
   file STRING2.CPP, line 98
Abnormal program termination
```

Fig. 18.5 Salida del manejador para probar la clase **String** (parte 8 de 8).

Para hacer la homonimia del operador incremental para permitir el uso tanto de preincremento como de postincremento, cada función de operador homónimo debe tener una firma distinta, de tal forma que el compilador sea capaz de determinar cuál es la versión de ++ que se desea. Se hace la homonimia de las versiones prefijas de la misma manera que cualquier operador unario prefijo.

Suponga, por ejemplo, que en el objeto **d1** de **Date** deseamos añadir 1 al día. Cuando el compilador ve la expresión preincremental

```
++d1
```

el compilador genera la llamada de función miembro

```
d1.operator++()
```

cuyo prototipo sería

```
Date operator++();
```

Si el preincremento es puesto en práctica como una función no miembro, cuando el compilador vea la expresión

```
++d1
```

el compilador generará la llamada de función

```
operator++(d1)
```

cuyo prototipo debería ser declarado en la clase **Date** como

```
friend Date operator++(Date &);
```

Hacer la homonimia del operador de postincremento resulta un pequeño desafío, porque el compilador debe ser capaz de distinguir las firmas de las funciones de operador de preincremento y postincremento homónimas. La regla convencional que ha sido adoptada en C++, es que cuando el compilador vea la expresión de postincremento

```
d1++
```

generará la llamada de función miembro

```
d1.operator++(0)
```

cuyo prototipo sería

```
Date operator++(int)
```

Aquí el 0 es un valor estricto de relleno, con el propósito que sea distinguible la lista de argumentos de **operator++** utilizada para postincrementar, de la lista de argumentos de **operator++** utilizada para preincrementar.

Si el postincremento es puesto en práctica como función no miembro, cuando el compilador vea la expresión

```
d1++
```

el compilador generará la llamada de función

```
operator++(d1, 0)
```

cuyo prototipo sería

```
friend Date operator++(Date &, int);
```

Otra vez, el argumento `0 int` es utilizado por el compilador para que la lista de argumentos de **operator++** utilizado para postincrementar, resulte distinguible de la lista de argumentos para preincrementar. Todo lo que hemos dicho en esta sección para la homonimia de operadores de preincremento y postincremento es aplicable a la homonimia de los operadores de predecremento y posdecremento. En la sección siguiente, examinaremos una clase **Date**, que utiliza operadores de preincremento y postincremento homónimos.

18.12 Estudio de caso: una clase Date

En la figura 18.6 se ilustra una clase **Date**. La clase utiliza operadores de preincremento y de postincremento homónimos, para añadir 1 al día en un objeto **Date**.

La clase proporciona las siguientes funciones miembro: un operador de inserción de flujo homónimo, un constructor por omisión, una función **setDate**, una función operador de preincremento homónima, una función operador de postincremento homónima, y un operador de asignación de adición homónimo (+=).

El programa manejador en **main** crea los objetos de fecha **d1** que se inicializan a January 1, 1990; **d2**, que se inicializan a December 27, 1992; y **d3** que se inicializan a una fecha inválida. El constructor **Date** llama a **setDate** para validar el mes, día y año específico. Si el mes es inválido, se define como 1. Un año inválido se define como 1900. Un día inválido se define como 1.

El programa manejador extrae cada uno de los objetos construidos **Date** utilizando el operador de inserción de flujo homónimo. El operador homónimo += es utilizado para añadir 7 días a **d2**. A continuación la función **setDate** es usada para definir **d3** a February 28, 1992. Después, un nuevo objeto **Date**, **d4**, se define a March 18, 1969. Entonces mediante el operador de preincremento homónimo, **d4** se incrementa en 1. La fecha es impresa, antes y después del preincremento, para confirmar que se hizo correctamente. Por ultimo, **d4** es incrementado utilizando el operador de postincremento homónimo. La fecha es impresa antes y después del postincremento, a fin de confirmar que funcionó correctamente.

La homonimia del operador de preincremento es directa. Para efectuar el incremento real el programa llama a la función de utilería **helpIncrement**. Esta función debe ocuparse de los desbordamientos que ocurren cuando intentamos aumentarle un día a un mes que ya está en su valor máximo. Estos excedentes requieren que el mes se incremente. Si el mes es ya el mes 12, el año también deberá ser incrementado.

El operador de preincremento homónimo devuelve una *copia* incrementada del objeto real. Esto ocurre porque el objeto real, *** this**, es regresado como un **Date**. Esto de hecho invoca al constructor de copia.

Resulta un poco más complejo hacer la homonimia del operador de postincremento. A fin de emular el efecto de postincremento, debemos regresar una copia del objeto **Date** sin incrementar. Al introducir el **operator++**, guardamos en **temp** el objeto actual (*** this**). A continuación llamamos a **helpIncrement** para incrementar el objeto. Por último regresamos la copia sin incrementar del objeto en **temp**.


```
// DATE1.H
// Definition of class Date
#ifndef DATE1_H
#define DATE1_H
#include <iostream.h>

class Date {
   friend ostream &operator<<(ostream &, const Date &);
public:
   Date(int m = 1, int d = 1, int y = 1900);  // def. constructor
   void setDate(int, int, int);  // set the date
   Date operator++();            // preincrement
   Date operator++(int);         // postincrement
   Date &operator+=(int);        // add days, modify object
private:
   int month;
   int day;
   int year;
   static int days[];      // array of days per month
   void helpIncrement();   // utility function
};

#endif
```

Fig. 18.6 Definición de la clase **Date** (parte 1 de 6).

```
// DATE1.CPP
// Member function definitions for Date class
#include <iostream.h>
#include "date1.h"

// Initialize static member at file scope;
// one class-wide copy.
int Date::days[] = {0, 31, 28, 31, 30, 31, 30,
                    31, 31, 30, 31, 30, 31};

// Date constructor
Date::Date(int m, int d, int y) { setDate(m, d, y); }

// Set the date
void Date::setDate(int mm, int dd, int yy)
{
   month = (mm >= 1 && mm <= 12) ? mm : 1;
   year = (yy >= 1900 && yy <= 2100) ? yy : 1900;

   // test for a leap year
   if (month == 2 && year % 4 == 0 && (year % 400 == 0 ||
                                       year % 100 != 0))
      day = (dd >= 1 && dd <= 29) ? dd : 1;
   else
      day = (dd >= 1 && dd <= days[month]) ? dd : 1;
}
```

Fig.18.6 Definición de función miembro para la clase **Date** (parte 2 de 6).

```
// Preincrement operator overloaded as a member function.

Date Date::operator++()
{
   helpIncrement();
   return *this;  // value return; not a reference return
}

// Postincrement operator overloaded as a member function.
// Note that the dummy integer parameter does not have a
// parameter name.

Date Date::operator++(int)
{
   Date temp = *this;
   helpIncrement();

   // return non-incremented, saved, temporary object
   return temp;   // value return; not a reference return
}

// Add a specific number of days to a date

Date &Date::operator+=(int additionalDays)
{
   for (int i = 1; i <= additionalDays; i++)
      helpIncrement();

   return *this;    // enables concatenation
}

// Function to help increment the date

void Date::helpIncrement()
{
   // test for a leap year first
   if (month == 2 && day < 29 && year % 4 == 0 &&
       (year % 400 == 0 || year % 100 != 0))
      ++day;
   else if (day < Date::days[month])  // not last day of month
      ++day;
   else if (month < 12) {  // last day of month < December
      ++month;
      day = 1;
   }
   else {                  // December 31
      month = 1;
      day = 1;
      ++year;
   }
}
```

Fig. 18.6 Definición de función miembro para la clase **Date** (parte 3 de 6).


```
// Overloaded output operator
ostream &operator<<(ostream &output, const Date &d)
{
   static char *monthName[13] = {"", "January",
      "February", "March", "April", "May", "June",
      "July", "August", "September", "October",
      "November", "December"};

   output << monthName[d.month] << ' '
          << d.day << ", " << d.year;

   return output;   // enables concatenation
}
```

Fig. 18.6 Definición de función miembro para la clase **Date** (parte 4 de 6).

```
// FIG18_6.CPP
// Driver for class Date
#include <iostream.h>
#include "date1.h"

main()
{
   Date d1, d2(12, 27, 1992), d3(0, 99, 8045);
   cout << "d1 is " << d1
        << "\nd2 is " << d2
        << "\nd3 is " << d3;

   cout << "\n\nd2 += 7 is " << (d2 += 7) << "\n\n";

   d3.setDate(2, 28, 1992);
   cout << "  d3 is " << d3;
   cout << "\n++d3 is " << ++d3 << "\n\n";

   Date d4(3, 18, 1969);

   cout << "Testing the preincrement operator:\n"
        << "  d4 is " << d4
        << "\n++d4 is " << ++d4
        << "\n  d4 is " << d4;

   cout << "\n\nTesting the postincrement operator:\n"
        << "  d4 is " << d4
        << "\nd4++ is " << d4++
        << "\n  d4 is " << d4 << endl;

   return 0;
}
```

Fig. 18.6 Manejador para la clase **Date** (parte 5 de 6).

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992

Testing the preincrement operator:
  d4 is March 18, 1969
++d4 is March 19, 1969
  d4 is March 19, 1969

Testing the postincrement operator:
  d4 is March 19, 1969
d4++ is March 19, 1969
  d4 is March 20, 1969
```

Fig. 18.6 Salida correspondiente al manejador de la clase **Date** (parte 6 de 6).

Resumen

- En C++ el operador << es utilizado para varios fines, como operador de inserción de flujo y como operador de desplazamiento a la izquierda. Esto es un ejemplo de homonimia de operadores. De igual forma, >> también tiene homónimo; se utiliza tanto como operador de extracción de flujo, como operador de desplazamiento a la derecha.
- En forma más general, C++ le permite al programador hacer la homonimia de la mayor parte de los operadores, para que sean sensibles al contexto en el cual están siendo utilizados. El compilador genera el código apropiado basándose en la forma en la cual cada operador es utilizado.
- La homonimia de operadores contribuye a la extensibilidad de C++.
- Se lleva a cabo la homonimia de operadores al escribir una definición de función (incluyendo un encabezado y un cuerpo). El nombre de la función se convierte en la palabra reservada **operator**, seguida por el símbolo correspondiente al operador homónimo.
- Para utilizar un operador sobre objetos de clase, dicho operador *debe* ser un operador homónimo con dos excepciones. El operador de asignación (=) puede ser utilizado con cualquier clase para llevar a cabo una copia a nivel de miembro por omisión, sin necesidad de tener homónimo. El operador de dirección (&) también puede ser utilizado sobre objetos de cualquier clase sin tener homónimo; sólo devuelve la dirección en la memoria del objeto.
- Al hacer la homonimia de operadores la idea subyacente es disponer de las mismas expresiones concisas para tipos definidos por usuario que C++ proporciona con su rica colección de operadores para tipos incorporados.
- La mayor parte de los operadores de C++ pueden tener homónimos.

- La homonimia de un operador no puede modificar su precedencia y asociatividad.
- En un operador homónimo no se pueden utilizar argumentos por omisión.
- No es posible modificar el número de operandos que un operador toma: los operadores unarios homónimos se conservan como operadores unarios; los operadores binarios homónimos se conservan como operadores binarios.
- No es posible crear símbolos para operadores nuevos; sólo se puede hacer la homonimia de operadores existentes.
- El significado de cómo opera un operador sobre objetos de tipos incorporados no puede ser modificado mediante la homonimia.
- Al hacer la homonimia de `()`, `[]`, `->`, o `=`, la función operador homónima debe ser declarada como miembro de clase.
- Las funciones operador pueden ser funciones miembro o funciones no miembro.
- Cuando una función operador es puesta en práctica como función miembro, el operando más a la izquierda debe ser un objeto de clase (o una referencia a un objeto de clase) de la clase del operador.
- Si el operando a la izquierda tiene que ser un objeto de clase distinta, esta función operador debe ser puesta en práctica como una función no miembro.
- Las funciones miembro de operador son llamadas sólo cuando el operando izquierdo de un operador binario es en específico un objeto de dicha clase o cuando el único operando de un operador unario es un objeto de dicha clase.
- Se podría escoger una función no miembro para hacer la homonimia de un operador, a fin de permitir que el operador sea conmutativo.
- Se puede hacer la homonimia de un operador unario como función miembro no estática sin argumentos o como función no miembro con un argumento; dicho argumento debe ser un objeto de un tipo definido por usuario o una referencia a un objeto de un tipo definido por usuario.
- Se puede hacer la homonimia de un operador binario como función miembro no estática con un argumento, o como función no miembro con dos argumentos (uno de los cuales debe ser un objeto de clase o una referencia a un objeto de clase).
- El operador de subíndice de arreglo `[]` no está restringido en su uso sólo con arreglos; puede ser utilizado para seleccionar elementos de otros tipos de clases contenedor como son listas enlazadas, cadenas, diccionarios y demás. Además, ya no es necesario que los subíndices sean enteros, también pueden ser utilizados caracteres o cadenas, por ejemplo.
- Un constructor de copia se utiliza para inicializar un objeto con otro objeto de la misma clase. Los constructores de copia también son invocados siempre que se requiera de la copia de un objeto, como es el caso en la llamada por valor o al regresar un valor de una función llamada.
- El compilador no sabe en forma automática cómo convertir entre tipos definidos por usuario y tipos incorporados. El programador debe especificar en forma explícita cómo ocurrirán dichas conversiones. Estas conversiones pueden ser llevadas a cabo mediante constructores de conversión (es decir, constructores de un argumento) que sólo convierten objetos de otros tipos en objetos de una clase particular.
- Un operador de conversión (u operador cast) puede ser utilizado para convertir un objeto de una clase a un objeto de otra o a un objeto de un tipo incorporado. Dicho operador de conversión

debe ser una función miembro no estática; este tipo de operador de conversión no puede ser una función amigo.

- Un constructor de conversión es un constructor de un argumento, que es utilizado para convertir el argumento en el objeto de la clase del constructor. El compilador puede llamar de forma implícita a un constructor de este tipo.
- El operador de asignación es el operador homónimo más frecuente. Por lo regular se utiliza para asignar un objeto a otro objeto de la misma clase, pero mediante el uso de los constructores de conversión; también puede ser utilizado para asignaciones entre clases.
- Si no se define un operador de asignación homónimo, la asignación se permitirá, pero será por omisión a una copia a nivel de miembro de cada miembro de datos. En algunos casos esto es aceptable. En el caso de objetos que contengan apuntadores a almacenamiento dinámico asignado, la copia a nivel de miembro dará como resultado dos objetos distintos apuntando al mismo almacenamiento asignado dinámicamente. Cuando se llame el destructor correspondiente a cualquiera de estos objetos, se liberará el almacenamiento dinámico asignado. Si el otro objeto entonces quiere referirse a dicho almacenamiento, el resultado quedará indefinido.
- Para hacer la homonimia del operador de incremento y permitir el uso tanto de preincremento como de postincremento, cada función de operador homónima debe tener una signatura distinta o diferente, de tal forma, que el compilador pueda ser capaz de determinar qué versión de `++` debe utilizar. Las versiones prefijas se hacen homónimas de igual forma que cualquier operador unario prefijo. El proporcionar una firma única a la función de operador de postincremento se consigue incluyendo un segundo argumento que debe ser del tipo `int`. De hecho, el usuario no proporciona un valor para este argumento entero especial. Aparece ahí simplemente para ayudar al compilador a distinguir entre versiones prefijas y postfijas de los operadores de incremento y de decremento.

Terminología

tipo incorporado
clase `Array`
clase `Date`
clase `String`
llamadas de función concatenadas
resolución de conversión de ambigüedad
constructor de conversión
función de conversión
operador de conversión
conversiones entre clases de tipos diferentes
conversiones entre tipos y clases incorporados
constructor de copia
conversiones explícitas de tipo (mediante cast)
operador homónimo de función amigo
homonimia de función
conversiones implícitas de tipo
operador homónimo de función miembro
operadores no sujetos de homonimia
palabra reservada `operator`
homonimia de operadores
operadores como funciones
operadores capaces de homonimia
operador homónimo `==`
operador homónimo `!=`
operador homónimo `<`
operador homónimo `<=`
operador homónimo `>`
operador homónimo `>=`
operador de asignación homónimo (`=`)
operador homónimo `[]`
hacer la homonimia
homonimia de un operador binario
homonimia de un operador unario
homonimia de un operador postfijo
homonimia de un operador prefijo
conversión definida por usuario
tipo definido por usuario

Errores comunes de programación

18.1 Intentar crear operadores nuevos.
18.2 Intentar modificar cómo funciona un operador con objetos de tipos incorporados.
18.3 Suponer que hacer la homonimia de un operador (como +) hace de forma automática la homonimia de sus operadores relacionados (como +=). Los operadores sólo pueden tener homónimos en forma explícita; la homonimia implícita no existe.
18.4 No probar la existencia y evitar la autoasignación cuando se hace la homonimia del operador de asignación de una clase que incluya un apuntador a almacenamiento dinámico.
18.5 No proporcionar un operador de asignación homónimo y un constructor de copia para una clase, cuando los objetos de dicha clase contengan apuntadores a almacenamiento dinámico asignado.

Prácticas sanas de programación

18.1 Utilice la homonimia de operadores cuando ésta haga más claro un programa si efectúa las mismas operaciones mediante llamadas de función explícitas.
18.2 Evite un uso excesivo o inconsistente de la homonimia de operadores, ya que ello puede hacer que un programa resulte críptico o difícil de leer.
18.3 Haga la homonimia de operadores para llevar a cabo la misma función o funciones bastante similares sobre objetos de clase que dichos operadores ejecutan sobre objetos de tipos incorporados.
18.4 Antes de escribir programas en C++ utilizando operadores homónimos, consulte los manuales de C++ correspondientes a su compilador, para informarse de las varias restricciones y requisitos únicos a operadores particulares.
18.5 Al hacer la homonimia de operadores unarios, es preferible hacer la funciones operador miembros de clase en vez de funciones amigo no miembro. Esta es una solución más limpia. Las funciones amigo y las clases amigo deberán evitarse, salvo que sean en lo absoluto necesarias. La utilización de amigos viola el encapsulado de una clase.
18.6 Un destructor, el operador de asignación, y un constructor de copia para una clase por lo general se proporcionan en grupo.

Sugerencias de rendimiento

18.1 Es posible hacer la homonimia de un operador como función no miembro, y no amigo, pero dicha función con la necesidad de tener acceso a datos privados o protegidos de una clase requeriría utilizar las funciones set o get proporcionadas en la interfaz pública de la clase. La sobrecarga de llamar a estas funciones causaría bajo rendimiento.
18.2 Usar el operador de concatenación homónimo `+=` que toma un solo argumento del tipo `const char *` da resultados más eficaces que primero tener que efectuar la conversión implícita y a continuación la concatenación. Las conversiones implícitas requieren de menos código y causan menos errores.

Observaciones de ingeniería de software

18.1 En una función operador, por lo menos un argumento debe ser un objeto de clase o una referencia a un objeto de clase. Esto impedirá que los programadores modifiquen cómo funcionan los operadores sobre objetos de tipos incorporados.
18.2 Se pueden añadir nuevas capacidades de entrada/salida a C++ correspondientes a tipos definidos por usuario, sin modificar las declaraciones o los miembros de datos privados correspondientes a la clase `ostream` o a la clase `istream`. Esto promueve la extensibilidad del lenguaje de programación C++ —uno de los aspectos más atractivos de C++.

18.3 Es posible evitar que un objeto de clase sea asignado a otro. Esto se lleva a cabo definiendo el operador de asignación como miembro privado de la clase.

Ejercicios de autoevaluación

18.1 Llene cada uno de los siguientes espacios en blanco:

a) Suponga que `a` y `b` son variables enteras y que formamos la suma `a + b`. Ahora suponga que `c` y `d` son variables de punto flotante y que formamos la suma `c + d`. Los dos operadores `+` están siendo utilizados aquí con claridad para fines distintos. Esto es un ejemplo de _____.

b) La palabra reservada _____ introduce una definición de función de operador homónimo.

c) Para utilizar operadores sobre objetos de clase, deben tener homónimos a excepción de los operadores _____ y _____.

d) La _____ y _____ de un operador no pueden ser modificadas mediante la homonimia.

18.2 Explique los varios significados de los operadores `<<` y `>>` en C++.

18.3 ¿En que contexto pudiera ser utilizado el nombre `operator/` en C++?.

18.4 (Verdadero/falso). En C ++ solo los operadores existentes pueden tener homónimos.

18.5 ¿Cómo se compara la precedencia de un operador homónimo en C++ con la precedencia del operador original?.

Respuestas a los ejercicios de autoevaluación

18.1 a) homonimia de operador. b) `operator`. c) asignación (=), dirección(`&`). d) precedencia, asociatividad.

18.2 El operador `>>` es, dependiendo del contexto, a la vez operador de desplazamiento a la derecha y operador de extracción de flujo. El operador `<<` es a la vez operador de desplazamiento a la izquierda y operador de inserción de flujo, dependiendo del contexto.

18.3 Para la homonimia de operadores: sería el nombre de una función que proporcionaría una nueva versión del operador `/`.

18.4 Verdadero.

18.5 Idéntico.

Ejercicios

18.6 De tantos ejemplos como pueda de la homonimia de operadores implícita en C. De tantos ejemplos como pueda de la homonimia de operadores implícita en C++. De un ejemplo razonable de una situación en la cual pudiera usted desear hacer la homonimia explícita de un operador en C++.

18.7 Los operadores C++ que no pueden tener homónimos son _____, _____, _____, _____ y _____.

18.8 La concatenación de cadenas requiere de dos operandos —las dos cadenas que han de concatenarse. En el texto mostramos cómo poner en práctica un operador de concatenación homónimo, que concatena el segundo objeto `String` a la derecha del primer objeto `String`, modificando así el primer objeto `String`. En algunas aplicaciones, es deseable producir un objeto concatenado `String` sin modificar los dos argumentos `String`. Diseñe un `operator+` para permitir operaciones como

```
string1 = string2 + string3;
```

18.9 *(Ejercicio de homonimia de operadores final.)* Para poder valorizar el cuidado que debe ponerse en la selección de operadores para su homonimia, liste cada uno de los operadores que se puedan hacer

homónimos en C++ y por cada uno de ellos liste un significado posible (o varios, si es apropiado) para cada una de las distintas clases que ha estudiado en este curso. Sugerimos que pruebe:

a) Arreglos
b) Pilas
c) Cadenas

Una vez hecho lo anterior, comente sobre qué operadores parecen tener significado para una amplia variedad de clases. ¿Qué operadores parecen tener poco valor para homonimia? ¿Qué operadores parecen ambiguos?

18.10 Ahora ejecute el proceso descrito en el programa anterior pero a la inversa. Liste cada uno de los operadores capaces de homonimia en C++. Para cada uno de ellos, liste lo que usted sienta es quizás la "operación óptima" para la cual dicho operador debería ser utilizado. Si existen varias operaciones excelentes, lístelas todas.

18.11 *(Proyecto).* C++ es un lenguaje en evolución, y continuamente se están desarrollando nuevos lenguajes. ¿Qué operadores adicionales recomendaría usted añadir a C++ o a un lenguaje similar a C++, que apoyasen tanto a la programación procedural como a la programación orientada a objetos? Escriba una cuidadosa justificación. Pudiera inclusive considerar enviar sus sugerencias al Comité ANSI C++.

18.12 Un buen ejemplo de la homonimia del operador de llamada de función `()` es permitir la forma más común de dobles subíndices de arreglo. En vez de decir

```
chessBoard[row][column]
```

en relación con un arreglo de objetos, haga la homonimia del operador de llamada de función para que pueda utilizarse la forma alterna

```
chessBoard(row, column)
```

18.13 Haga la homonimia del operador de subíndice para regresar un miembro dado en una lista enlazada.

18.14 Haga la homonimia del operador de subíndice para regresar el elemento más grande de una colección, el segundo más grande, el tercero más grande, etcétera.

18.15 Considere la clase `Complex` mostrada en la figura 18.7. La clase permite operaciones de los *números* llamados *complejos*. Estos son números de forma `realPart + imaginaryPart * i`, donde *i* tiene el valor:

$$\sqrt{-1}$$

a) Modifique la clase, para permitir la entrada y salida de números complejos, mediante los operadores homónimos `>>` y `<<`, respectivamente (deberá eliminar la función de impresión de la clase).

b) Haga la homonimia del operador de multiplicación, para permitir la multiplicación de números complejos, como en álgebra.

c) Haga la homonimia de los operadores `==` y `!=`, para permitir comparaciones de números complejos.

18.16 Una máquina con enteros de 32 bits puede representar enteros en el rango de aproximadamente -2 mil millones hasta +2 mil millones. Esta restricción de tamaño fijo rara vez causa problemas. Pero existen muchas aplicaciones en las cuales nos gustaría ser capaces de utilizar un rango mucho más amplio de enteros. Para esto fue construido C++, es decir para crear nuevos tipos poderosos de datos. Considere la clase `HugeInt` de la figura 18.8. Estudie cuidadosamente esta clase, y después

a) Describa con precisión cómo funciona.

b) ¿Qué restricciones tiene la clase?

c) Modifique la clase para que pueda procesar enteros arbitrariamente grandes. (Sugerencia: utilice una lista enlazada para representar a `HugeInt`.)

d) Haga la homonimia del operador de multiplicación `*`.

e) Haga la homonimia del operador de división `/`.

f) Haga la homonimia de todos los operadores relacionales y de igualdad.

```
// COMPLEX1.H
// Definition of class Complex
#ifndef COMPLEX1_H
#define COMPLEX1_H

class Complex {
public:
   Complex(double = 0.0, double = 0.0);       // constructor
   Complex operator+(const Complex &) const; // addition
   Complex operator-(const Complex &) const; // subtraction
   Complex &operator=(const Complex &);       // assignment
   void print() const;                        // output
private:
   double real;      // real part
   double imaginary; // imaginary part
};

#endif
```

Fig. 18.7 Definición de la clase **Complex** (parte 3 de 5).

```
// COMPLEX1.CPP
// Member function definitions for class Complex
#include <iostream.h>
#include "complex1.h"

// Constructor
Complex::Complex(double r, double i)
{
   real = r;
   imaginary = i;
}

// Overloaded addition operator
Complex Complex::operator+(const Complex &operand2) const
{
   Complex sum;
   sum.real = real + operand2.real;
   sum.imaginary = imaginary + operand2.imaginary;
   return sum;
}

// Overloaded subtraction operator
Complex Complex::operator-(const Complex &operand2) const
{
   Complex diff;
   diff.real = real - operand2.real;
   diff.imaginary = imaginary - operand2.imaginary;
   return diff;
}
```

Fig. 18.7 Definiciones de función miembro para la clase **Complex** (parte 2 de 5).


```
// Overloaded = operator
Complex& Complex::operator=(const Complex &right)
{
   real = right.real;
   imaginary = right.imaginary;
   return *this;   // enables concatenation
}

// Display a Complex object in the form: (a, b)
void Complex::print() const
   { cout << '(' << real << ", " << imaginary << ')'; }
```

Fig. 18.7 Definiciones de función miembro para la clase **Complex** (parte 3 de 5).

```
// FIG18_7.CPP
// Driver for class Complex
#include <iostream.h>
#include "complex1.h"

main()
{
   Complex x, y(4.3, 8.2), z(3.3, 1.1);

   cout << "x: ";
   x.print();
   cout << "\ny: ";
   y.print();
   cout << "\nz: ";
   z.print();

   x = y + z;
   cout << "\n\nx = y + z:\n";
   x.print();
   cout << " = ";
   y.print();
   cout << " + ";
   z.print();

   x = y - z;
   cout << "\n\nx = y - z:\n";
   x.print();
   cout << " = ";
   y.print();
   cout << " - ";
   z.print();
   cout << '\n';

   return 0;
}
```

Fig. 18.7 Manejador para la clase **Complex** (parte 4 de 5).

```
x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)

x = y + z:
(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

x = y - z:
(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)
```

Fig. 18.7 Salida del manejador para la clase **Complex** (parte 5 de 5).

```
// HUGEINT1.H
// Definition of the HugeInt class
#ifndef HUGEINT1_H
#define HUGEINT1_H

#include <iostream.h>

class HugeInt {
   friend ostream &operator<<(ostream &, HugeInt &);
public:
   HugeInt(long = 0);             // conversion constructor
   HugeInt(const char *);         // conversion constructor
   HugeInt operator+(HugeInt &); // addition
private:
   short integer[30];
};

#endif
```

Fig. 18.8 Clase de grandes enteros definida por usuario (parte 1 de 4).

```
// HUGEINT1.CPP
// Member and friend function definitions for class HugeInt
#include <string.h>
#include "hugeint1.h"

// Conversion constructor
HugeInt::HugeInt(long val)
{
   for (int i = 0; i <= 29; i++)
      integer[i] = 0;    // initialize array to zero

   for (i = 29; val != 0 && i >= 0; i--) {
      integer[i] = val % 10;
      val /= 10;
   }
}
```

Fig. 18.8 Clase de grandes enteros definida por usuario (parte 2 de 4).


```
HugeInt::HugeInt(const char *string)
{
   int i, j;

   for (i = 0; i <= 29; i++)
      integer[i] = 0;

   for (i = 30 - strlen(string), j = 0; i <= 29; i++, j++)
      integer[i] = string[j] - '0';
}

// Addition
HugeInt HugeInt::operator+(HugeInt &op2)
{
   HugeInt temp;
   int carry = 0;

   for (int i = 29; i >= 0; i--) {
      temp.integer[i] = integer[i] + op2.integer[i] + carry;

      if (temp.integer[i] > 9) {
         temp.integer[i] %= 10;
         carry = 1;
      }
      else
         carry = 0;
   }

   return temp;
}

ostream& operator<<(ostream &output, HugeInt &num)
{
   for (int i = 0; (num.integer[i] == 0) && (i <= 29); i++)
      ; // skip leading zeros

   if (i == 30)
      output << 0;
   else
      for ( ; i <= 29; i++)
         output << num.integer[i];

   return output;
}
```

Fig. 18.8 Clase de grandes enteros definida por usuario (parte 3 de 4).

18.17 Cree una clase **rationalNumber** (fracciones) con las siguientes capacidades:

a) Cree un constructor que impida la existencia de un denominador 0 en una fracción, que simplifique las fracciones que no estén en forma simplificada, y que evite denominadores negativos.

b) Haga la homonimia de los operadores de suma, resta, multiplicación y división para esta clase.

c) Haga la homonimia de los operadores relacionales y de igualdad para esta clase.

```
// FIG18_8.CPP
// Test driver for HugeInt class
#include <iostream.h>
#include "hugeint1.h"

main()
{
   HugeInt n1(7654321), n2(7891234),
           n3("99999999999999999999999999999"),
           n4("1"), n5;

   cout << "n1 is " << n1 << "\nn2 is " << n2
        << "\nn3 is " << n3 << "\nn4 is " << n4
        << "\nn5 is " << n5 << "\n\n";

   n5 = n1 + n2;
   cout << n1 << " + " << n2 << " = " << n5 << "\n\n";

   cout << n3 << " + " << n4 << "\n= " << (n3 + n4) << "\n\n";

   n5 = n1 + 9;
   cout << n1 << " + " << 9 << " = " << n5 << "\n\n";

   n5 = n2 + "10000";
   cout << n2 << " + " << "10000" << " = " << n5 << "\n\n";

   return 0;
}
```

```
n1 is 7654321
n2 is 7891234
n3 is 99999999999999999999999999999
n4 is 1
n5 is 0

7654321 + 7891234 = 15545555

99999999999999999999999999999 + 1
= 100000000000000000000000000000

7654321 + 9 = 7654330

7891234 + 10000 = 7901234
```

Fig. 18.8 Clase de grandes enteros definida por usuario (parte 4 de 4).

18.18 El operador **sizeof** en un objeto **String** sólo devuelve el tamaño de los miembros de datos del objeto **String**; no incluye la longitud del espacio **String** asignado por **new**. Haga la homonimia del operador **sizeof** para que regrese el tamaño del objeto **String**, incluyendo la memoria dinámicamente asignada.

18.19 Estudie las funciones de biblioteca de manejo de cadenas de C y ponga en práctica cada una de las funciones como parte de la clase **String**. A continuación utilice estas funciones para llevar a cabo manipulaciones de texto.

18.20 Desarrolle la clase **Polynomial**. La representación interna de un **Polynomial** es una lista de términos enlazados. Cada término tiene un coeficiente y un exponente. El término

$$2x^4$$

tiene un coeficiente de 2 y un exponente de 4. Desarrolle una clase completa que contenga las funciones destructor y constructor apropiadas así como las funciones *set* y *get*. La clase también deberá proporcionar las siguientes capacidades de operadores homónimos:

a) Haga la homonimia del operador de suma (+) para poder sumar dos **Polynomial**.
b) Haga la homonimia del operador de resta (-) para substraer dos **Polynomials**.
c) Haga la homonimia del operador de asignación, para asignar un **Polynomial** a otro.
d) Haga la homonimia del operador de multiplicación (*) para multiplicar dos **Polynomials**.
e) Haga la homonimia del operador de asignación de adición (+=), del operador de asignación de substracción (-=) y del operador de asignación de multiplicación (*=).

19

Herencia

Objetivos

- Ser capaz de crear nuevas clases heredando de clases existentes.
- Comprender como la herencia fomenta la reutilización del software.
- Comprender los conceptos de clases base y clases derivadas.
- Ser capaz de utilizar herencia múltiple para derivar una clase a partir de varias clases base.

No diga que conoce por completo a otro, hasta que con él haya dividido una herencia.
Johann Kasper Lavater

Este método es para definir como el número de una clase la clase de todas las clases similares a la clase dada.
Bertrand Russell

Un mazo de cartas fue diseñado como la más pura de las jerarquías, siendo cada una de las cartas superior a todas las que están debajo de ella, y lacayo de las que están por encima.
Ely Culbertson

Aunque es muy bueno heredar una biblioteca, mejor aún es reunirla.
Augustine Birrell

Ahorre autoridad base de libros de terceros.
William Shakespeare
Love's Labour's Lost

Sinopsis

19.1 Introducción

En éste y en el capítulo siguiente analizamos dos de las capacidades de mayor importancia que proporciona la programación orientada a objetos *herencia* y *polimorfismo*. La herencia es una forma de *reutilización del software*, en la cual se crean clases nuevas a partir de clases existentes, mediante la absorción de sus atributos y comportamientos, y embelleciendo éstos con las capacidades que las clases nuevas requieren. La reutilización del software ahorra tiempo en el desarrollo de programas, fomenta la reutilización de software de alta calidad, probado y depurado, y reduce problemas en un sistema después de convertido en funcional. Estas son excitantes posibilidades. El polimorfismo nos permite escribir programas en forma general, a fin de procesar una amplia variedad de clases relacionadas tanto existentes como aún sin especificar. La herencia y el polimorfismo son técnicas efectivas para enfrentarse con la complejidad del software.

Al crear una clase nueva, en vez de escribir en su totalidad miembros de datos y funciones miembro nuevos, el programador puede determinar que la clase nueva debe *heredar* los miembros de datos y las funciones miembro provenientes de una *clase base* ya definida. La clase nueva se

conoce como *clase derivada*. Cada clase derivada misma se convierte en candidato a clase base, para alguna clase derivada futura. Mediante la *herencia única*, una clase es derivada de una única clase base. Con la *herencia múltiple* una clase derivada hereda de múltiples clases base (posiblemente no relacionadas). A menudo una clase derivada añade miembros de datos y funciones miembro propias, por lo que en general una clase derivada es más grande que su clase base. Una clase derivada es más específica que su clase base y representa un grupo más pequeño de objetos. Con la herencia única, la clase derivada se inicia en esencia de la misma forma que la clase base. La fuerza real de la herencia proviene de la capacidad de definir en la clase derivada, adiciones, reemplazos o refinamientos de las características heredadas de la clase base.

Cada objeto de una clase derivada también es un objeto de la clase base de dicha clase derivada. Sin embargo; la inversa no es verdad, los objetos de la clase base no son objetos de la clase derivada de dicha clase base. Nos aprovecharemos de esta relación "un objeto de clase derivada es un objeto de clase base" para ejecutar algunas manipulaciones interesantes. Por ejemplo, podemos unir una gran variedad de objetos distintos relacionados mediante la herencia en una lista enlazada de objetos de clase base. Esto permite que una variedad de objetos se procesen de forma general. Como veremos en éste y en el siguiente capítulo, esto es un impulso clave a la programación orientada a objetos.

En este capítulo añadimos una nueva forma de control de acceso de miembros, es decir el acceso protegido. Para el acceso a los miembros de clase base protegidos, las clases derivadas y sus amigos reciben un trato favorecido en relación con otras funciones.

La experiencia en la elaboración de sistemas de software indica que grandes porciones de código se refieren a casos especiales muy relacionados. En dichos sistemas se hace difícil visualizar la "imagen general" porque el diseñador y el programador se han visto preocupados por los casos especiales. La programación orientada a objetos proporciona varias formas de "ver el bosque a través de los árboles" —un proceso conocido a veces como *abstracción*.

Si un programa está lleno de casos especiales muy relacionados, entonces es común ver enunciados `switch`, que distinguen entre casos especiales, que proporcionan la lógica de proceso necesaria para manejar cada caso en forma individual. En el capítulo 20, mostraremos cómo utilizar la herencia y el polimorfismo para reemplazar la lógica `switch` por una lógica más simple.

Distinguimos entre *relaciones "es un"* y *relaciones "tiene un"*. "Es un" es herencia. En una relación "es un", un objeto de un tipo de clase derivada también puede ser tratado como un objeto del tipo de clase base. "Tiene un" es composición (véase la figura 17.4). En una relación "tiene un" un objeto de clase tiene como miembros uno o más objetos de otras clases.

Una clase derivada no puede tener acceso a los miembros privados de su clase base; permitirlo violaría el encapsulado de la clase base. Una clase derivada puede, sin embargo, tener acceso a los miembros públicos y protegidos de su clase base. Los miembros de clase base que no deban ser accesibles a una clase derivada mediante la herencia, en la clase base se declararán privados. Una clase derivada puede tener acceso a los miembros privados de la clase base mediante funciones de acceso, provistas en la interfaz pública de la clase base. Un problema, tratándose de la herencia, es que una clase derivada puede heredar funciones miembro públicas que no requiera, y que expresamente no tendría. Cuando un miembro de clase base no es apropiado para una clase derivada, dicho miembro puede ser redefinido en la clase derivada con una puesta en práctica apropiada.

De lo más excitante resulta la idea que las clases nuevas pueden heredar de *bibliotecas de clase* existentes. Las organizaciones desarrollan sus propias bibliotecas de clase y pueden aprovechar de otras bibliotecas disponibles en todo el mundo. La perspectiva es que, en algún momento, el software será elaborado a partir de *componentes reutilizables estandarizados*, de la

misma forma que, hoy día, a menudo es construido el hardware. Esto ayudará a estar a la altura del reto de desarrollar el software más y más poderoso que en el futuro necesitaremos.

19.2 Clases base y clases derivadas

A menudo un objeto de clase en realidad también "es un" objeto de otra clase. Un rectángulo *es un* cuadrilátero (como lo es un cuadrado, un paralelogramo y un trapezoide). Entonces, la clase **Rectangle** puede decirse que *hereda* de la clase **Quadrilateral**. En este contexto, la clase **Quadrilateral** se conoce como *clase base* y la clase **Rectangle** se conoce como *clase derivada*. Un rectángulo *es un* tipo específico de un cuadrilátero, pero sería incorrecto decir que un cuadrilátero *es un* rectángulo. En la figura 19.1 se muestran varios ejemplos sencillos de herencia.

Otros lenguajes de programación orientados a objetos, como Smalltalk, utilizan diferentes terminologías: en la herencia, la clase base se conoce como *superclase* y la clase derivada como *subclase*. Dado que por lo general la herencia produce clases derivadas de tamaño mayor que las clases base, los términos superclase y subclase parecen inapropiados; evitaremos utilizar estos términos.

La herencia forma estructuras jerárquicas de apariencia arborescente. Una clase base existe en una relación jerárquica con sus clases derivadas. Ciertamente una clase puede existir por sí misma, pero es cuando una clase es utilizada mediante el mecanismo de la herencia, que ésta se convierte en clase base, proveyendo atributos y comportamientos a otras clases, o la clase se convierte en una clase derivada, que hereda dichos atributos y comportamientos.

Desarrollemos una jerarquía simple de herencia. Una comunidad universitaria típica está formada por miles de personas que son miembros de la misma. Estas personas son los empleados y los estudiantes. Los empleados son o miembros de la facultad o personal de oficinas. Los miembros de la facultad son o administradores (rectores o coordinadores de departamentos) o la facultad de profesores. Esto da como resultado la jerarquía de herencia que se muestra en la figura 19.2.

Clase base	Clases derivadas
Student	CommuterStudent
	ResidentStudent
Shape	Circle
	Triangle
	Rectangle
Loan	CarLoan
	HomeImprovementLoan
	MortgageLoan
Employee	FacultyMember
	StaffMember
Account	CheckingAccount
	SavingsAccount

Fig. 19.1 Algunos ejemplos simples de herencia.

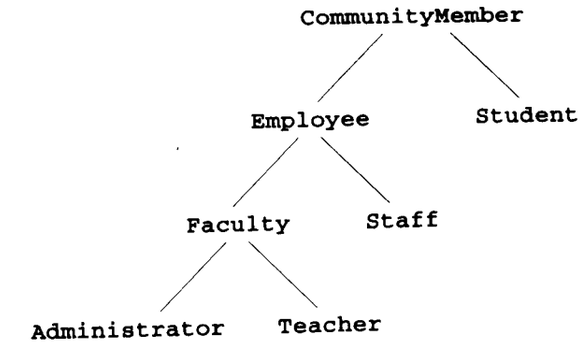


Fig. 19.2 Una jerarquía de herencia para los miembros de una comunidad universitaria.

Otra jerarquía de herencia sustancial es la jerarquía **Shape** de la figura 19.3. Un comentario común entre aquellos alumnos que estudian por primera vez la programación orientada a objetos, es que en el mundo real existen ejemplos abundantes de jerarquías. Es simple que a estos alumnos jamás se les ha pedido que categoricen el mundo real de esta forma, por lo que deben hacer algunos ajustes en su forma de pensar.

Para especificar que la clase **CommissionWorker** se deriva de la clase **Employee**, típicamente la clase **CommissionWorker** sería definida como sigue:

```
class CommissionWorker : public Employee {
   ...
};
```

Esto se conoce como *herencia pública* y es el tipo que más probablemente utilice el lector. También analizaremos *herencia privada* y *herencia protegida*. En el caso de la herencia pública, los miembros públicos y protegidos de la clase base son heredados como miembros públicos y protegidos de la clase derivada, respectivamente.

Es posible tratar en forma similar a los objetos de la clase base y a los objetos de la clase derivada. El estado común es expresado en los atributos y comportamientos de la clase base. Los objetos de cualquier clase derivada pública originados de una clase base común, pueden todos ellos ser tratados como objetos de dicha clase base.

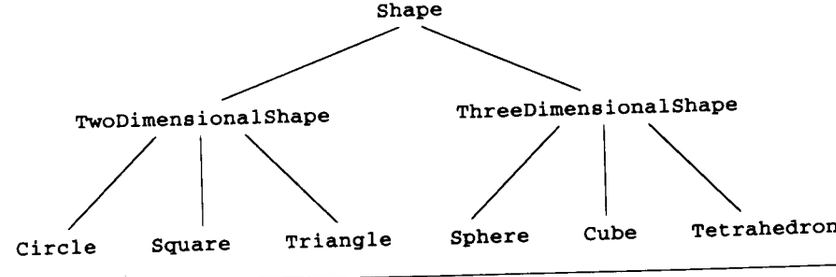


Fig. 19.3 Una porción de la jerarquía de clase **Shape**.

Veremos muchos ejemplos en los cuales aprovecharemos esta relación con una facilidad de programación no disponible en lenguajes no orientados a los objetos como es C.

19.3 Miembros protegidos

Los miembros públicos de una clase base son accesibles a todas las funciones en el programa. Los miembros privados de clase base son accesibles sólo para las funciones miembro y los amigos de la clase base.

El acceso protegido sirve como nivel intermedio de protección entre el acceso público y el acceso privado. Los miembros protegidos de clase base son accesibles sólo por miembros y amigos de la clase base, y por miembros y amigos de las clases derivadas.

Los miembros de clases derivadas pueden hacer referencia a miembros públicos y protegidos de la clase base sólo utilizando los nombres de los miembros. No es necesario utilizar el operador de resolución de alcance. Se supone el objeto actual.

19.4 Cómo hacer la conversión explícita (cast) de apuntadores de clase base a apuntadores de clase derivada

Un objeto de clase derivada pública también puede ser tratado como objeto de su clase base correspondiente. Esto permite algunas manipulaciones interesantes. Por ejemplo, a pesar del hecho que objetos de una variedad de clases derivadas de clase base particular pudieran ser muy distintos unos de los otros, aún así, podemos crear una lista enlazada de los mismos otra vez, siempre y cuando los tratemos como objetos de clase base. Pero lo inverso no es cierto: un objeto de clase base no es también automáticamente un objeto de clase derivada.

Error común de programación 19.1

Puede causar errores tratar un objeto de clase base como si fuera un objeto de clase derivada.

El programa podrá, sin embargo; utilizar un especificador de conversión explícita (cast) para convertir uno de clase base a un apuntador de clase derivada. Pero tenga cuidado si dicho apuntador debe ser desreferenciado, entonces primero deberá hacer que señale a un objeto del tipo de clase derivada.

Error común de programación 19.2

Efectuar la conversión explícita de un apuntador de clase base que señala a un objeto de clase base a un apuntador de clase derivada y a continuación, hacer referencia a miembros de clase derivada que no existen en dicho objeto.

Nuestro primer ejemplo aparece en la figura 19.4. partes 1 hasta la 5. Las partes 1 y 2 muestran la definición de clase **Point** y las definiciones de funciones miembro **Point**. Las partes 3 y 4 muestran la definición de clase **Circle** y la definición de función miembro **Circle**. La parte 5 muestra un programa manejador, en el cual demostramos la asignación de apuntadores de clase derivada a apuntadores de clase base, y la conversión explícita (cast) de apuntadores de base clase a apuntadores de clase derivada.

Examinemos primero la definición de clase **Point** (figura 19.4, parte 1). La interfaz pública a **Point** tiene las funciones miembro **setPoint, getX**, y **getY**. Los miembros de datos **x** y **y** de **Point** se definen como **protected**. Esto impide que los usuarios de los objetos **Point** tengan acceso directo a los datos, pero permite que las clases que se deriven de **Point** tengan acceso directo a los miembros de datos heredados. Si los datos fueran definidos como **private**, para tener acceso a los datos tendrían que ser utilizadas las funciones miembro públicas de **Point**.

Note que la función de operador de inserción de flujo homónima **Point** (figura 19.4 parte 2), es capaz de hacer referencia directa a las variables **x** y **y** porque éstas son miembros protegidos de la clase **Point**. Note también que es necesario hacer referencia a **x** y a **y** mediante objetos como en **p.x** y **p.y**. Esto es debido a que la función de operador de inserción de flujo homónima no es una función miembro de la clase **Circle**, sino que es un amigo de la clase.

```
// POINT.H
// Definition of class Point
#ifndef POINT_H
#define POINT_H

class Point {
   friend ostream &operator<<(ostream &, const Point &);
public:
   Point(float = 0, float = 0);       // default constructor
   void setPoint(float, float);       // set coordinates
   float getX() const { return x; }   // get x coordinate
   float getY() const { return y; }   // get y coordinate
protected:          // accessible by derived classes
   float x, y;      // x and y coordinates of the Point
};

#endif
```

Fig. 19.4 Definición de la clase **Point** (parte 1 de 5).

```
// POINT.CPP
// Member functions for class Point
#include <iostream.h>
#include "point.h"

// Constructor for class Point
Point::Point(float a, float b)
{
   x = a;
   y = b;
}

// Set x and y coordinates of Point
void Point::setPoint(float a, float b)
{
   x = a;
   y = b;
}

// Output Point (with overloaded stream insertion operator)
ostream &operator<<(ostream &output, const Point &p)
{
   output << '[' << p.x << ", " << p.y << ']';

   return output;   // enables concatenated calls
}
```

Fig. 19.4 Definiciones de función miembro para la clase **Point** (parte 2 de 5).

```
// CIRCLE.H
// Definition of class Circle
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream.h>
#include <iomanip.h>
#include "point.h"

class Circle : public Point {  // Circle inherits from Point
   friend ostream &operator<<(ostream &, const Circle &);
public:
   // default constructor
   Circle(float r = 0.0, float x = 0, float y = 0);

   void setRadius(float);      // set radius
   float getRadius() const;    // return radius
   float area() const;         // calculate area
protected:
   float radius;
};

#endif
```

Fig. 19.4 Definición de la clase **Circle** (parte 3 de 5).

```
// CIRCLE.CPP
// Member function definitions for class Circle
#include "circle.h"

// Constructor for Circle calls constructor for Point
// with a member initializer then initializes radius.
Circle::Circle(float r, float a, float b)
   : Point(a, b)       // call base-class constructor
{ radius = r; }

// Set radius of Circle
void Circle::setRadius(float r) { radius = r; }

// Get radius of Circle
float Circle::getRadius() const { return radius; }

// Calculate area of Circle
float Circle::area() const
{ return 3.14159 * radius * radius; }

// Output a Circle in the form:
// Center = [x, y]; Radius = #.##
ostream &operator<<(ostream &output, const Circle &c)
{
   output << "Center = [" << c.x << ", " << c.y
          << "]; Radius = " << setiosflags(ios::showpoint)
          << setprecision(2) << c.radius;

   return output;    // enables concatenated calls
}
```

Fig. 19.4 Definición de función miembro para la clase **Circle** (parte 4 de 5).


```
// FIG19_4.CPP
// Casting base-class pointers to derived-class pointers

#include <iostream.h>
#include <iomanip.h>
#include "point.h"
#include "circle.h"

main()
{
   Point *pointPtr, p(3.5, 5.3);
   Circle *circlePtr, c(2.7, 1.2, 8.9);

   cout << "Point p: " << p << "\nCircle c: " << c << endl;

   // Treat a Circle as a Circle (with some casting)
   pointPtr = &c;   // assign address of Circle to pointPtr
   circlePtr = (Circle *) pointPtr;  // cast base to derived
   cout << "\nArea of c (via circlePtr): "
        << circlePtr->area() << endl;

   // DANGEROUS: Treat a Point as a Circle
   pointPtr = &p;   // assign address of Point to pointPtr
   circlePtr = (Circle *) pointPtr;  // cast base to derived
   cout << "\nRadius of object circlePtr points to: "
        << circlePtr->getRadius() << endl;

   return 0;
}
```

```
Point p: [3.5, 5.3]
Circle c: Center = [1.2, 8.9]; Radius = 2.70

Area of c (via circlePtr): 22.90

Radius of object ciclePtr points to: 4.02e-38
```

Fig. 19.4 Conversión explícita de apuntadores de clase base a apuntadores de clase derivada (parte 5 de 5).

La clase **Circle** (figura 19.4, parte 3) hereda de la clase **Point** mediante herencia pública. Esto se especifica en la primera línea de la definición de clase

```
class Circle : public Point {     // Circle inherits from Point
```

Los dos puntos (**:**) en el encabezado de la definición de clase indican herencia. La palabra reservada **public** indica el tipo de herencia (vea la sección 19.7). Todos los miembros de la clase **Point** son heredados a la clase **Circle**. Esto significa que la interfaz pública a **Circle** incluye la funciones miembro públicas de **Point**, así como las funciones miembros **Circle** de nombre **area, setRadius**, y **getRadius**.

El constructor **Circle** (figura 19.4, parte 4), deberá invocar al constructor **Point** para inicializar la porción de clase base de un objeto **Circle**. Esto se lleva a cabo con la sintaxis de inicializador de miembros que se presentó en el capítulo 17, como sigue

```
Circle::Circle(float r, float a, float b)
   : Point(a, b)       // call base-class constructor
```

La segunda línea del encabezado de función constructor invoca al constructor **Point** por su nombre. Los valores **a** y **b** son pasados del constructor **Circle** al constructor **Point**, a fin de inicializar los miembros de clase base **x** e **y**. Note que la función de operador de inserción de flujo homónima es capaz de hacer referencia directa a las variables **x** y **y**, porque estos son miembros protegidos de la clase base **Point**. Note también que es necesario hacer referencia a **x** y a **y** mediante objetos como en **c.x** y en **c.y**. Esto es debido a que la función del operador de inserción de flujo homónima no es un miembro de la clase **Circle**, sino un amigo de la clase.

El programa manejador (figura 19.4, parte 5) crea **pointPtr** como un apuntador a un objeto **Point** y produce el objeto **p** de la clase **Point**, y a continuación crea **circlePtr**, como un apuntador a un objeto **Circle** y produce el objeto **c** de **Circle**. Los objetos **Point** y **Circle** son extraídos utilizando sus operadores de inserción de flujo homónimos, para demostrar que fueron inicializados en forma correcta. A continuación, el manejador demuestra la asignación de un apuntador de clase derivada (la dirección del objeto **c**) a un apuntador de clase base de nombre **pointPtr** y la conversión explícita (cast) de **pointPtr** de regreso a **Circle ***. El resultado de la operación de conversión explícita (cast) se asigna a **circlePtr**. El área del objeto **Circle** de nombre **c** es extraído vía **circlePtr**. Esto da como resultado un valor válido de superficie, porque los apuntadores están siempre señalando a un objeto **Circle**. Siempre resulta válido asignar un apuntador de clase derivada a un apuntador de clase base, porque un objeto de clase derivada *es un* objeto de clase base. El apuntador de clase base sólo "ve" la parte correspondiente de clase base del objeto de clase derivada. El compilador lleva a cabo una conversión implícita del apuntador de clase derivada a un apuntador de clase base. Un apuntador de clase base no puede ser asignado en forma directa a un apuntador de clase derivada, porque sería una asignación inherente peligrosa los apuntadores de clase derivada se espera estén apuntados a objetos de clase derivada. En este caso el compilador no llevará a cabo una conversión implícita. El uso de una conversión explícita (cast) informará al compilador que el programador sabe que este tipo de conversión de apuntador es peligrosa y que el programador asume la responsabilidad de utilizar adecuadamente dicho apuntador.

A continuación, el manejador demuestra la asignación del apuntador de clase base (la dirección del operador **p**) al apuntador de clase base de nombre **pointPtr** y la conversión explícita (cast) de **pointPtr**, de regreso a **Circle ***. El resultado de la operación de conversión explícita (cast) es asignado a **circlePtr**. El radio del objeto al cual **circlePtr** señala (objeto **p** de **Point**) es extraído vía **circlePtr**. Esto da como resultado un valor inválido, porque los apuntadores están siempre señalando a un objeto **Point**. Y un objeto **Point** no tiene un miembro **radius**. Por lo tanto, el programa extraerá cualquier valor que en ese momento exista en memoria en la posición que **circlePtr** espera que esté el miembro de datos **radius**.

19.5 Cómo utilizar funciones miembro

Cuando se forma una clase derivada a partir de una clase base, las funciones miembro de la clase derivada pudiera ser necesario que contengan ciertos miembros de clase base.

Observación de ingeniería de software 19.1

Una clase derivada no puede tener acceso directo a miembros privados de su clase base.

Este es un aspecto crucial de ingeniería de software en C++. Si una clase derivada podría tener acceso a los miembros privados de la clase base, esto violaría el encapsulado de la clase base. El ocultamiento de los miembros privados es gran ayuda en la prueba, depuración y modificación correcta de los sistemas. Si una clase derivada tuviera acceso a los miembros privados de su clase base, sería entonces posible para las clases derivadas de dicha clase derivada también tener acceso a dichos datos, y así en lo sucesivo. Esto propagaría el acceso de lo que se supone deben ser datos privados y a todo lo largo de la jerarquía de la clase se perderían los beneficios del encapsulado.

19.6 Cómo redefinir los miembros de clase base en una clase derivada

Una clase derivada puede redefinir una función miembro de clase base. Cuando en la clase derivada dicha función es mencionada por su nombre, de forma automática se selecciona la versión de la clase derivada. Puede utilizarse el operador de resolución de alcance, para tener acceso a la versión de clase base partiendo de la clase derivada.

Error común de programación 19.3

Cuando se redefine una función miembro de clase base en una clase derivada, es común hacer que la versión de clase derivada llame a la versión de clase base y haga algún trabajo adicional. Causa recursión infinita no utilizar el operador de resolución de alcance para hacer referencia a la función miembro de la clase base, porque la función miembro de la clase derivada, de hecho, está llamando a sí misma.

Práctica sana de programación 19.1

Al heredar capacidades que no son necesarias en la clase derivada, dichas capacidades se deben enmascarar mediante redefinición de las funciones.

Observación de ingeniería de software 19.2

Una redefinición de una función miembro de clase base en una clase derivada no necesariamente debe tener la misma signatura que la función miembro de clase base.

UNIVERSIDAD DE LA REPUBLICA
FACULTAD DE INGENIERIA
DEPARTAMENTO DE

Veamos por ejemplo una clase **Employee** simplificada. Almacena el **firstName** y el **lastName** del empleado. Esta información es común a todos los empleados, incluyendo a aquellos de clases derivadas de la clase **Employee**. De la clase **Employee** derivan ahora las clases **Hourlyworker**, **PieceWorker**. **Boss**, y **CommissionWorker**. Al **Hourlyworker** se le paga por hora, "una vez y media" por las horas extra en exceso de 40 horas por semana. Al **PieceWorker** se le paga una tasa fija por cada elemento producido y para simplificar, suponga que esta persona fabrica un tipo de elemento, por lo que los miembros de datos privados son el número de elementos producidos y la tasa correspondiente por cada elemento. El **Boss** recibe un salario fijo por semana. El **CommissionWorker** recibe un reducido salario base semanal fijo, más un porcentaje fijo de las ventas brutas efectuadas por dicha persona durante la semana. A fin de simplificar, ahora estudiaremos sólo la clase **Employee** y la clase derivada **Hourlyworker**.

Nuestro siguiente ejemplo aparece en la figura 19.5, partes 1 hasta la 5. Las parte 1 y 2 muestran la definición de clase **Employee** y las definiciones de función miembro **Employee**. Las partes 3 y 4 muestran la definición de clase **Hourlyworker** y la definición de función miembro **Hourlyworker**. La parte 5 muestra un programa manejador para la jerarquía de herencia **Employee/Hourlyworker**, que sólo produce un objeto **Hourlyworker**, lo inicializa y llama a la función miembro **print** de **Hourlyworker**, con el fin de extraer los datos del objeto.

```
// EMPLOY.H
// Definition of class Employee
#ifndef EMPLOY_H
#define EMPLOY_H

class Employee {
public:
   Employee(const char*, const char*);  // constructor
   void print() const;  // output first and last name
   ~Employee();         // destructor
private:
   char *firstName;     // dynamically allocated string
   char *lastName;      // dynamically allocated string
};

#endif
```

Fig. 19.5 Definición de la clase **Employee** (parte 1 de 5).

```
// EMPLOY.CPP
// Member function definitions for class Employee
#include <string.h>
#include <iostream.h>
#include <assert.h>
#include "employ.h"

// Constructor dynamically allocates space for the
// first and last name and uses strcpy to copy
// the first and last names into the object.
Employee::Employee(const char *first, const char *last)
{
   firstName = new char[ strlen(first) + 1 ];
   assert(firstName != 0); // terminate if memory not allocated
   strcpy(firstName, first);

   lastName = new char[ strlen(last) + 1 ];
   assert(lastName != 0);  // terminate if memory not allocated
   strcpy(lastName, last);
}

// Output employee name
void Employee::print() const
{ cout << firstName << ' ' << lastName; }

// Destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   delete [] firstName;    // reclaim dynamic memory
   delete [] lastName;     // reclaim dynamic memory
}
```

Fig. 19.5 Definiciones de funciones miembro para la clase **Employee** (parte 2 de 5).


```
// HOURLY.H
// Definition of class HourlyWorker
#ifndef HOURLY_H
#define HOURLY_H

#include "employ.h"

class HourlyWorker : public Employee {
public:
   HourlyWorker(const char*, const char*, float, float);
   float getPay() const;  // calculate and return salary
   void print() const;    // redefined base-class print
private:
   float wage;            // wage per hour
   float hours;           // hours worked for week
};

#endif
```

Fig. 19.5 Definición de la clase **HourlyWorker** (parte 3 de 5).

```
// HOURLY_B.CPP
// Member function definitions for class HourlyWorker
#include <iostream.h>
#include <iomanip.h>
#include "hourly.h"

// Constructor for class HourlyWorker
HourlyWorker::HourlyWorker(const char *first, const char *last,
                           float initHours, float initWage)
   : Employee(first, last)   // call base-class constructor
{
   hours = initHours;
   wage = initWage;
}

// Get the HourlyWorker's pay
float HourlyWorker::getPay() const { return wage * hours; }

// Print the HourlyWorker's name and pay
void HourlyWorker::print() const
{
   cout << "HourlyWorker::print()\n\n";

   Employee::print();   // call base-class print function

   cout << " is an hourly worker with pay of"
        << " $" << setiosflags(ios::showpoint)
        << setprecision(2) << getPay() << endl;
}
```

Fig. 19.5 Definiciones de funciones miembro para la clase **HourlyWorker** (parte 4 de 5).

```
// FIG19_5.CPP
// Redefining a base-class member function in a
// derived class.
#include <iostream.h>
#include "hourly.h"

main()
{
   HourlyWorker h("Bob", "Smith", 40.0, 7.50);
   h.print();
   return 0;
}
```

```
HourlyWorker::print()

Bob Smith is an hourly worker with pay of $300.00
```

Fig. 19.5 Cómo redefinir una función miembro de clase base en una clase derivada (parte 5 de 5).

La definición de clase **Employee** (figura 19.5 parte 1) está formada de dos miembros de datos privados **char *** que son **firstName** y **lastName** y de tres funciones miembro un constructor, un destructor y **print**. La función constructor (figura 19.5 part 2) recibe dos cadenas y asigna dinámicamente arreglos de caracteres para almacenar las cadenas. Note que es utilizado el macro **assert** (analizado en el capítulo 14, "Temas avanzados"), para determinar si se asignó memoria a **firstName** y **lastName**. Si no es así, el programa termina con mensaje de error indicando la condición probada, el número de línea en el cual aparece dicha condición, y el archivo en el cual la condición está almacenada. Dado que los datos de **Employee** son **private**, el único acceso a los datos es mediante la función miembro **print**, que sólo extrae el nombre y el apellido del empleado. La función destructor devuelve al sistema la memoria dinámicamente asignada.

La clase **HourlyWorker** (figura 19.5 parte 3) hereda de la clase **Employee** mediante herencia pública. Otra vez, esto queda especificado en la primera línea de la definición de clase, como sigue:

```
class HourlyWorker : public Employee
```

La interfaz pública de **HourlyWorker** incluye la función **Employee print** y las funciones miembro de **HourlyWorker**, de nombre **getPay** y **print**. Note que la clase **HourlyWorker** define su propia función **print**. Por lo tanto, la clase **HourlyWorker** tiene acceso a dos funciones **print**. La clase **HourlyWorker** también contiene los miembros de datos privados **wage** y **hours**, para el cálculo del salario semanal del empleado.

El constructor **HourlyWorker** (figura 19.5 parte 4) utiliza sintaxis de inicializador de miembro para pasar las cadenas **first** y **last** al constructor **Employee**, de forma tal, que puedan ser inicializados los miembros de la clase base, y a continuación inicializa a los miembros **wage** y **hours**. La función miembro **getPay** calcula el salario de **HourlyWorker**.

La función miembro de **HourlyWorker**, de nombre **print**, es un ejemplo de la función miembro de clase base, redefinida en una clase derivada. A menudo se redefinen las funciones miembro de clase base en una clase derivada, a fin de dar más funcionalidad. Las funciones redefinidas a veces llaman a la versión de clase base de la función para llevar a cabo parte de la

nueva tarea. En este ejemplo, la función de clase derivada **print** llama la función de clase base **print**, para extraer el nombre del empleado (la clase base **print** es la única función con acceso a los datos privados de la clase base). La función de clase derivada **print** también extrae la paga del empleado. Note cómo es llamada la versión de clase base de la función **print**

```
Employee::print();
```

Dado que tanto la función de clase base como la función de clase derivada tienen el mismo nombre, la función de clase base debe ser antecedida por su nombre de clase y por el operador de resolución de alcance. De lo contrario, la función correspondiente a la versión de la clase derivada será llamada, lo que resultaría una recursión infinita (la función **HourlyWorker print** se llamaría a sí misma).

19.7 Clases base públicas, protegidas y privadas

Al derivar una clase a partir de una clase base, la clase base puede ser heredada como **public**, **protected**, o **private**. La herencia protegida y la herencia privada son raras y deberían ser utilizadas sólo con extremo cuidado; en los ejemplos de este libro solo utilizamos herencia pública.

Al derivar una clase a partir de una clase base pública, los miembros públicos de la clase base se convierten en miembros públicos de la clase derivada, y los miembros protegidos de la clase base se convierten en miembros protegidos de la clase derivada. Los miembros privados de una clase base nunca son accesibles en forma directa desde una clase derivada.

Al derivar una clase a partir de una clase base protegida, los miembros públicos y protegidos de la clase base se convierten en miembros protegidos de la clase derivada.

Cuando se deriva una clase a partir de una clase base privada, los miembros públicos y protegidos de la clase base se convierten en miembros privados de la clase derivada.

19.8 Clases base directas y clases base indirectas

Una clase base puede ser una *clase base directa* de una clase derivada, o una clase base puede ser una *clase base indirecta* de una clase derivada. Un clase base directa de una clase derivada es listada en forma explícita en el encabezado de dicha clase derivada cuando se declara esta clase derivada. Una clase base indirecta no se lista en forma explícita en el encabezado de la clase derivada; más bien la clase base indirecta se hereda desde varios niveles arriba en la jerarquía de la clase.

19.9 Cómo utilizar constructores y destructores en clases derivadas

Dado que una clase derivada hereda los miembros de su clase base, cuando es producido un objeto de una clase derivada, el constructor de la clase base deberá de ser llamado, para inicializar los miembros de la clase base del objeto de la clase derivada. El constructor de la clase derivada puede llamar de forma implícita al constructor de la clase base o en el constructor de la clase derivada un *inicializador de clase base* (mismo que utiliza la sintaxis de inicializador de miembro que ya hemos visto) puede ser proveído, a fin de llamar de manera explícita al constructor de la clase base.

Los constructores de la clase base y los operadores de asignación de la clase base no son heredados por las clases derivadas. Sin embargo, los constructores y los operadores de asignación de las clases derivadas, pueden llamar a los constructores y a los operadores de asignación de la clase base.

Un constructor de clase derivada siempre llamará primero al constructor correspondiente de su clase base para inicializar los miembros de la clase base de la clase derivada. Si el constructor de la clase derivada ha sido omitido, el constructor por omisión de la clase derivada llamará al constructor de la clase base. Los destructores son llamados en orden inverso a las llamadas de constructor, por lo que un destructor de clase derivada será llamado antes de su destructor de clase base.

Observación de ingeniería de software 19.3

Cuando en una clase derivada se cree un objeto, primero se ejecutará el constructor de clase base, después se ejecutarán los constructores de los objetos miembro correspondientes a la clase derivada, y después se ejecutará el constructor de la clase derivada. Los destructores serán llamados en orden inverso en el cual fueron llamados sus correspondientes constructores.

El estándar de C++ especifica que el orden en que son construidos los objetos miembros es el orden en el cual han sido declarados dichos objetos dentro de la definición de clase. No afecta a la construcción el orden en el cual los inicializadores de miembro están listados. En la herencia, los constructores de clase base son llamados en el orden en el cual se haya especificado la herencia en la definición de clase derivada. El orden en el cual se han especificado los constructores de clase base en el constructor de clase derivada, no afecta a la construcción.

Error común de programación 19.4

Escribir un programa que dependa de que los objetos miembros sean construidos en un orden particular, puede producir errores sutiles.

El programa de la figura 19.6 demuestra el orden en que los constructores y destructores de clase base y de clase derivada son llamados. El programa está formado de cinco partes. Las partes 1 y 2 muestran una clase **Point** simple que contiene un constructor, un destructor y miembros de datos protegidos **x** y **y**. Tanto el constructor como el destructor ambos imprimen el objeto **Point** para el cual son invocados. Las partes 3 y 4 muestran una clase simple **Circle** derivada de **Point** mediante herencia pública y que contiene un constructor, un destructor y el miembro de datos privado **radius**. Tanto el constructor como el destructor imprimen el objeto **Circle** para el cual fueron invocados. El constructor **Circle** también invoca al constructor **Point** utilizando sintaxis de inicializador de miembro y pasa los valores **a** y **b**, de tal forma que los miembros de datos de clase base puedan ser inicializados.

En la parte 5 se muestra un programa manejador para esta jerarquía **Point/Circle**. El programa empieza produciendo un objeto **Point** en su propio alcance dentro de **main**. El objeto entra y sale de alcance de inmediato, por lo que tanto el constructor como el destructor **Point** son llamados. El programa a continuación produce el objeto **Circle** de nombre **circle1**. Esto invoca al constructor **Point** para llevar a cabo una salida con los valores pasados a partir del constructor **Circle**, y a continuación ejecuta la salida especificada en el constructor **Circle**. A continuación se produce el objeto **Circle** de nombre **circle2**. Otra vez, tanto el constructor **Point** como el **Circle** son llamados. Note que se ejecuta el cuerpo del constructor **Point** antes del cuerpo del constructor **Circle**. Se alcanza el fin de **main**, por lo que deberán ser llamados los destructores tanto para **circle1** como para **circle2**. Los destructores son llamados en orden inverso a sus constructores correspondientes. Por lo tanto el destructor **Circle** y el destructor **Point** son llamados en este orden para el objeto **circle2**, y a continuación el destructor **Circle** y el destructor **Point** son llamados en ese orden para el objeto **circle1**.


```
// POINT2.H
// Definition of class Point
#ifndef POINT2_H
#define POINT2_H

class Point {
public:
   Point(float = 0.0, float = 0.0);  // default constructor
   ~Point();     // destructor
protected:       // accessible by derived classes
   float x, y;   // x and y coordinates of Point
};

#endif
```

Fig. 19.6 Definición de clase **Point** (parte 1 de 5).

```
// POINT2.CPP
// Member function definitions for class Point
#include <iostream.h>
#include "point2.h"

// Constructor for class Point
Point::Point(float a, float b)
{
   x = a;
   y = b;

   cout << "Point  constructor: "
        << '[' << x << ", " << y << ']' << endl;
}

// Destructor for class Point
Point::~Point()
{
   cout << "Point  destructor:  "
        << '[' << x << ", " << y << ']' << endl;
}
```

UNIVERSIDAD DE LA REPUBLICA
FACULTAD DE INGENIERIA
DEPARTAMENTO

Fig. 19.6 Definiciones de funciones miembro para la clase **Point** (parte 2 de 5).

19.10 Conversión implícita de objeto de clase derivada a objeto de clase base

A pesar del hecho que un objeto de clase derivada también "es un" objeto de clase base, el tipo de la clase derivada y el tipo de la clase base son distintos. Los objetos de clase derivada pueden ser tratados como objetos de clase base. Tiene sentido este tipo de asignación, porque la clase derivada tiene miembros que corresponden a cada uno de los miembros de la clase base —recuerde que por lo regular la clase derivada tiene más miembros que la clase base. En la otra dirección la asignación no es permitida, porque asignar un objeto de clase base a un objeto de clase derivada dejaría sin definición a los miembros adicionales de la clase derivada. Aunque tal asignación no

```
// CIRCLE2.H
// Definition of class Circle
#ifndef CIRCLE2_H
#define CIRCLE2_H

#include "point2.h"
#include <iomanip.h>

class Circle : public Point {
public:
   // default constructor
   Circle(float r = 0.0, float x = 0, float y = 0);

   ~Circle();       // destructor
private:
   float radius;    // radius of Circle
};

#endif
```

Fig. 19.6 Definición de clase `Circle` (parte 3 de 5).

```
// CIRCLE2.CPP
// Member function definitions for class Circle
#include "circle2.h"

// Constructor for Circle calls constructor for Point
Circle::Circle(float r, float a, float b)
   : Point(a, b)    // call base-class constructor
{
   radius = r;

   cout << "Circle constructor: radius is "
        << radius << " [" << a << ", " << b << ']' << endl;
}

// Destructor for class Circle
Circle::~Circle()
{
   cout << "Circle destructor:  radius is "
        << radius << " [" << x << ", " << y << ']' << endl;
}
```

Fig. 19.6 Definiciones de funciones miembro para la clase `Circle` (parte 4 de 5).

es permitida en forma "natural", se puede legitimar proporcionando un operador de asignación y/o un constructor de conversión correctamente homónimos.

Error común de programación 19.5

Asignar un objeto de clase derivada a un objeto de una clase base correspondiente, y a continuación intentar hacer referencia a miembros de sólo la clase derivada en el objeto nuevo de clase.


```
// FIG19_6.CPP
// Demonstrate when base-class and derived-class
// constructors and destructors are called.
#include <iostream.h>
#include "point2.h"
#include "circle2.h"

main()
{
   // Show constructor and destructor calls for Point
   {
      Point p(1.1, 2.2);
   }

   cout << endl;
   Circle circle1(4.5, 7.2, 2.9);
   cout << endl;
   Circle circle2(10, 5, 5);
   cout << endl;
   return 0;
}
```

```
Point  constructor: [1.1, 2.2]
Point  destructor:  [1.1, 2.2]

Point  constructor: [7.2, 2.9]
Circle constructor: radius is 4.5 [7.2, 2.9]

Point  constructor: [5, 5]
Circle constructor: radius is 10 [5, 5]

Circle destructor: radius is 10 [5, 5]
Point  destructor: [5, 5]
Circle destructor: radius is 4.5 [7.2, 2.9]
Point  destructor: [7.2, 2.9]
```

Fig. 19.6 Orden en el cual son llamados los constructores y destructores de clase base y clase derivada (parte 5 de 5).

Un apuntador a un objeto de clase derivada puede ser convertido en forma implícita en un apuntador a un objeto de una clase base, porque un objeto de una clase derivada es un objeto de una clase base.

Existen cuatro formas posibles para mezclar y hacer coincidir apuntadores de clase base y apuntadores de clase derivada con objetos de clase base y objetos de clase derivada:

1. Es simple hacer referencia a un objeto de clase base con un apuntador de clase base.
2. También es simple hacer referencia a un objeto de clase derivada con un apuntador de clase derivada.
3. Es seguro hacer referencia a un objeto de clase derivada con un apuntador de clase base, porque el objeto de clase derivada es también un objeto de su base clase. El programa

sólo puede referirse a miembros de clase base. Si mediante el apuntador de clase base el programa hace referencia a miembros sólo de clase derivada, el compilador informará de un error de sintaxis.

4. Es un error de sintaxis hacer referencia a un objeto de clase base mediante un apuntador de clase derivada. El apuntador de clase derivada deberá ser convertido primero explícitamente en un apuntador de clase base.

Error común de programación 19.6

Puede causar error convertir explícitamente (cast) un apuntador de clase base a un apuntador de clase derivada, si a continuación dicho apuntador es utilizado para hacer referencia a un objeto de clase base que no tenga los miembros deseados de la clase derivada.

Independiente de lo conveniente que resulte tratar los objetos de la clase derivada como si fueran objetos de clase base, y hacerlo manipulando todos estos objetos mediante apuntadores de clase base, existe un problema. En un sistema de nóminas, por ejemplo, nos gustaría ser capaces de recorrer una lista enlazada de empleados y calcular la paga semanal para cada persona. Pero utilizar apuntadores de clase base sólo le permite al programa llamar la rutina de cálculo de nóminas de clase base (si en verdad tal rutina existe en la clase base). Necesitamos una forma para invocar para cada objeto la rutina de cálculo de nóminas adecuada, sea éste un objeto de clase base o un objeto de clase derivada, y hacerlo con facilidad utilizando el apuntador de clase base. La solución es utilizar funciones virtuales y polimorfismo, como se analiza en el capítulo 20.

19.11 Ingeniería de software con herencia

Podemos utilizar la herencia para *personalizar* el software existente. Heredamos los atributos y comportamientos de una clase existente, y a continuación añadimos o retiramos atributos y comportamientos para personalizar la clase a fin de que cumpla con nuestras necesidades. Esto en C++ se efectúa sin que la clase derivada tenga acceso al código fuente de la clase base, pero la clase derivada sí debe tener la capacidad de enlazarse con el código objeto de la clase base. Esta poderosa capacidad resulta atractiva para los fabricantes independientes de software (ISV). Los ISV pueden desarrollar clases propietarias para venta o para licencia, y poner estas clases a disposición de los usuarios en formato de código objeto. Los usuarios pueden entonces con rápidez derivar de estas clases de biblioteca clases nuevas, sin necesidad de tener acceso al código fuente propiedad de los ISV. Todo lo que los ISV tienen que suministrar junto con el código objeto son los archivos de cabecera.

Puede ser difícil para los estudiantes darse cuenta de los problemas a los que los diseñadores y los instaladores se enfrentan tratándose de proyectos de software a gran escala. Las personas con experiencia en estos proyectos invariablemente declararán que, para mejorar el proceso de desarrollo de software, resulta clave fomentar la reutilización del software. La programación orientada a objetos en general, y C++ en particular, alienta y fomenta la reutilización del software.

Es la disponibilidad de bibliotecas de clases sustanciales y útiles la que proporciona el máximo de beneficios en la reutilización del software mediante la herencia. Conforme crezca el interés en C++, aumentará el interés en las bibliotecas de clases. Igual que con la llegada de la computadora personal, el software listo para usarse, producido por fabricantes independientes de software, se convirtió en una industria con un crecimiento explosivo, también, por lo tanto, será la creación y la venta de bibliotecas de clases. Partiendo de estas bibliotecas los diseñadores de aplicaciones construirán sus aplicaciones, y los diseñadores de bibliotecas se verán premiados al empacarse dichas bibliotecas junto con las aplicaciones. Las bibliotecas actualmente incluidas con los

compiladores de C++ tienden a ser más bien de uso general y de alcance limitado. Lo que estamos viendo venir es un compromiso masivo a nivel mundial para el desarrollo de bibliotecas de clases y para una gran variedad de campos de aplicaciones.

Observación de ingeniería de software 19.4

Crear una clase derivada no afecta el código fuente o el código objeto de su clase base; mediante la herencia se conserva la integridad de la clase base.

Una clase base define un estado común. Todas las clases derivadas de una clase base heredan las capacidades de dicha clase base. En el proceso de diseño orientado a objetos, el diseñador busca el estado común y lo extrae para formar clases base deseables. Las clases derivadas después son personalizadas, más allá de las capacidades heredadas de las clases base.

De igual forma que el diseñador de sistemas no orientados a objetos busca evitar una proliferación innecesaria de funciones, el diseñador de sistemas orientados a objetos debería evitar una proliferación innecesaria de clases. Dicha proliferación de clases crea problemas de administración y puede dificultar la reutilización del software, debido a que para el reutilizador potencial de una clase resulta más difícil localizar dicha clase en una gran colección. La contrapartida es crear menos clases, capaz cada una de ellas de proporcionar funcionalidad adicional. Para ciertos reutilizadores dichas clases podrían resultar demasiado ricas; esta funcionalidad excesiva pudiera ser enmascarada, "aminorando" entonces dichas clases a fin de que cumplan con sus necesidades.

Sugerencia de rendimiento 19.1

Si las clases producidas mediante la herencia son más grandes de lo requerido, pudiera dar como resultado un desperdicio de memoria y de recursos de proceso.

Note que puede resultar confuso leer un conjunto de declaraciones de clase derivada, porque no se muestran los miembros heredados. Pero aún así en las clases derivadas los miembros heredados están presentes.

Observación de ingeniería de software 19.5

En un sistema orientado a objetos, las clases a menudo están muy relacionadas. "Disgregue y elimine" atributos y comportamientos comunes y colóquelos en una clase base. A continuación utilice la herencia para formar clases derivadas.

Observación de ingeniería de software 19.6

Una clase derivada contiene los atributos y comportamientos de su clase base. Una clase derivada también puede contener atributos y comportamientos adicionales. Con la herencia, la clase base puede ser compilada independientemente de la clase derivada. Para tener la capacidad de combinar estos atributos y comportamientos adicionales con la clase base para formar la clase derivada basta compilar los atributos y comportamientos incrementados de la clase derivada.

Observación de ingeniería de software 19.7

Las modificaciones a una clase base no requieren que sea modificada la clase derivada, siempre y cuando la interfaz pública de la clase base se mantenga sin modificación. Pudiera sin embargo ser necesario recompilar las clases derivadas.

19.12 Composición en comparación con herencia

Hemos discutido relaciones *es un* basadas en la herencia. También hemos analizado relaciones *tiene un* (y hemos visto ejemplos en capítulos anteriores) en las cuales un objeto puede tener otros

objetos —como miembros mediante la *composición* de clases existentes estas relaciones crean nuevas clases. Por ejemplo, dadas las clases **Employee**, **BirthDate**, y **TelephoneNumber**, no es correcto decir que un **Employee** *es un* **BirthDate** o que un **Employee** *es un* **TelephoneNumber**. Pero es correcto decir que un **Employee** *tiene un* **BirthDate** y un **Employee** *tiene un* **TelephoneNumber**.

Recuerde que el orden en el cual los objetos miembros se construyen se define como el orden en el cual ha sido declarado dicho objeto. Los constructores de clase base serán llamados en el orden en el cual ha sido especificada la herencia en la clase derivada.

Observación de ingeniería de software 19.8

Las modificaciones a una clase miembro no requieren que su clase compuesta que la encierra sea modificada, siempre y cuando se mantenga sin modificación la interfaz pública de la clase miembro. Note que pudiera ser necesario recompilar la clase compuesta.

19.13 Relaciones "utiliza un" y "conoce un"

La herencia y la composición, al crear nuevas clases que tengan mucho en común con clases existentes, cada una de ellas fomentan la reutilización del software. Existen otras formas de utilizar los servicios de las clases. Aunque un objeto persona no es un automóvil, y un objeto persona no contiene un automóvil, un objeto persona *utiliza un automóvil*. Una función utiliza un objeto, sólo al emitir una llamada de función a una función miembro de dicho objeto.

Un objeto puede estar *consciente de* otro objeto. Las redes de conocimiento a menudo tienen estas relaciones. Para estar consciente de un objeto, otro objeto puede contener un apuntador o una referencia a dicho objeto. En este caso, se dice que un objeto tiene una relación de conciencia con el otro objeto.

19.14 Estudio de caso: Point, Circle, Cylinder

Veamos ahora el ejercicio recapitulativo correspondiente a este capítulo. Veremos una jerarquía de punto, círculo y cilindro. Primero desarrollaremos y utilizaremos la clase **Point** (figura 19.7). Después presentaremos un ejemplo en el cual de la clase **Point** derivaremos la clase **Circle** (figura 19.8). Por último, presentaremos un ejemplo en el cual de la clase **Circle** derivaremos la clase **Cylinder** (figura 19.9).

La figura 19.7 muestra la clase **Point**. La Parte 1 de la figura 19.7 muestra la definición de la clase **Point**. Note que los miembros de datos de **Point** están protegidos. Entonces, cuando de la clase **Point** derivemos la clase **Circle**, las funciones miembro de la clase **Circle** podrán hacer referencia directa a las coordenadas **x** y **y**, en vez de tener que utilizar funciones de acceso. Esto pudiera resultar en un mejor rendimiento.

En la figura 19.7 parte 2, se muestran las definiciones de funciones miembro correspondientes a la clase **Point**. En la figura 19.7 parte 3, se muestra el programa manejador para la clase **Point**. Note que **main** deberá utilizar las funciones de acceso **getX** y **getY** para leer los valores de los miembros de dato protegidos **x** e **y**; recuerde que los miembros de dato protegidos son accesibles sólo a miembros y amigos de su clase y a miembros y amigos de sus clases derivadas.

Nuestro siguiente ejemplo se muestra en la figura 19.8 partes 1 a 3. La definición de clase **Point** y las definiciones de función miembro correspondientes de la figura 19.7 son utilizadas aquí otra vez. Las partes 1 a 3 muestran la definición de clase **Circle**, las definiciones de función miembro **Circle**, y el programa manejador, respectivamente. Note que la clase **Circle** hereda de la clase **Point** mediante herencia pública. Esto significa que la interfaz pública de **Circle**


```
// POINT2.H
// Definition of class Point
#ifndef POINT2_H
#define POINT2_H

class Point {
   friend ostream &operator<<(ostream &, const Point &);
public:
   Point(float = 0, float = 0);       // default constructor
   void setPoint(float, float);       // set coordinates
   float getX() const { return x; }   // get x coordinate
   float getY() const { return y; }   // get y coordinate
protected:         // accessible to derived classes
   float x, y;     // coordinates of the point
};

#endif
```

Fig. 19.7 Definición de clase **Point** (parte 1 de 3).

```
// POINT2.CPP
// Member functions for class Point
#include <iostream.h>
#include "point2.h"

// Constructor for class Point
Point::Point(float a, float b)
{
   x = a;
   y = b;
}

// Set the x and y coordinates
void Point::setPoint(float a, float b)
{
   x = a;
   y = b;
}

// Output the Point
ostream &operator<<(ostream &output, const Point &p)
{
   output << '[' << p.x << ", " << p.y << ']';

   return output;             // enables concatenation
}
```

Fig. 19.7 Funciones miembro para la clase **Point** (parte 2 de 3).

incluye las funciones miembro **Point**, así como las funciones miembro **Circle** de nombre **setRadius**, **getRadius**, y **area**. Note que la función de operador de inserción de flujo homónimo **Circle** puede hacer referencia directa a las variables **x** y **y**, porque éstas son

```
// FIG19_7.CPP
// Driver for class Point
#include <iostream.h>
#include "point2.h"

main()
{
   Point p(7.2, 11.5);    // instantiate Point object p

   // protected data of Point inaccessible to main
   cout << "X coordinate is " << p.getX()
        << "\nY coordinate is " << p.getY();

   p.setPoint(10, 10);
   cout << "\n\nThe new location of p is " << p << endl;

   return 0;
}
```

```
X coordinate is 7.2
Y coordinate is 11.5

The new location of p is [10, 10]
```

Fig. 19.7 Manejador para la clase **Point** (parte 3 de 3).

miembros protegidos de la clase base **Point**. Note también que es necesario hacer referencia a **x** y a **y** mediante objetos, como en **c.x** y en **c.y**. Esto es debido a que la función de operador de inserción de flujo homónima no es una función miembro de la clase **Circle**, pero un amigo de la clase. El programa manejador produce un objeto de la clase **Circle**, a continuación utiliza funciones *get* para obtener la información sobre el objeto **Circle**. Otra vez, **main** no es una función miembro ni amigo de la clase **Circle**, por lo cual no puede hacer referencia directa a los datos protegidos de la clase **Circle**. El programa manejador utiliza entonces la función *set* de nombre **setRadius** y **setPoint** para redefinir el radio y las coordenadas del centro del círculo. Por último, el manejador hace algo en especial interesante. Inicializa la variable de referencia **pRef** del tipo "referencia a un objeto **Point**" (**Point &**), al objeto **c** de **Circle**. El manejador a continuación imprime **pRef**, el cual, a pesar del hecho que es inicializado con un objeto **Circle**, "piensa" que es un objeto **Point**, por lo que el objeto **Circle** de hecho se imprime como un objeto **Point**.

Nuestro último ejemplo se muestra en la figura 19.9, partes 1 a 3. Las definiciones de la clase **Point** y de la clase **Circle**, y las definiciones de funciones miembro correspondientes a las figuras 19.7 y 19.8 son vueltas a usar aquí. Las partes 1 a 3 muestran la definición de clase **Cylinder**, las definiciones de función miembro **Cylinder**, y un programa manejador, respectivamente. Note que la clase **Cylinder** hereda de la clase **Circle** mediante herencia pública. Esto significa que la interfaz pública a **Cylinder** incluye las funciones miembro **Circle**, así como las funciones miembro **Cylinder**, de nombre **setHeight**, **getHeigtht**, **area** (redefinidos a partir de **Circle**) y **volume**. Note que la función de operador de inserción de flujo homónima **Cylinder** es capaz de hacer referencia directa a las variables **x**, **y**, y **radius**, porque


```
// CIRCLE2.H
// Definition of class Circle
#ifndef CIRCLE2_H
#define CIRCLE2_H

#include "point2.h"

class Circle : public Point {
   friend ostream &operator<<(ostream &, const Circle &);
public:
   // default constructor
   Circle(float r = 0.0, float x = 0, float y = 0);
   void setRadius(float);       // set radius
   float getRadius() const;     // return radius
   float area() const;          // calculate area
protected:          // accessible to derived classes
   float radius;    // radius of the Circle
};

#endif
```

Fig. 19.8 Definición de la clase **Circle** (parte 1 de 3).

```
// CIRCLE2.CPP
// Member function definitions for class Circle
#include <iostream.h>
#include <iomanip.h>
#include "circle2.h"

// Constructor for Circle calls constructor for Point
// with a member initializer and initializes radius
Circle::Circle(float r, float a, float b)
   : Point(a, b)        // call base-class constructor
{ radius = r; }

// Set radius
void Circle::setRadius(float r) { radius = r; }

// Get radius
float Circle::getRadius() const { return radius; }

// Calculate area of Circle
float Circle::area() const
   { return 3.14159 * radius * radius; }

// Output a circle in the form:
// Center = [x, y]; Radius = #.##
ostream &operator<<(ostream &output, const Circle &c)
{
   output << "Center = [" << c.x << ", " << c.y
          << "]; Radius = " << setiosflags(ios::showpoint)
          << setprecision(2) << c.radius;

   return output;    //  enables concatenated calls
}
```

Fig. 19.8 Definiciones de función miembro para la clase **Circle** (parte 2 de 3).

```
// FIG19_8.CPP
// Driver for class Circle
#include <iostream.h>
#include "point2.h"
#include "circle2.h"

main()
{
   Circle c(2.5, 3.7, 4.3);

   cout << "X coordinate is " << c.getX()
        << "\nY coordinate is " << c.getY()
        << "\nRadius is " << c.getRadius();

   c.setRadius(4.25);
   c.setPoint(2, 2);
   cout << "\n\nThe new location and radius of c are\n"
        << c << "\nArea " << c.area() << endl;

   Point &pRef = c;
   cout << "\nCircle printed as a Point is: "
        << pRef << endl;

   return 0;
}
```

```
X coordinate is 3.7
Y coordinate is 4.3
Radius is 2.5

The new location and radius of c are
Center = [2, 2]; Radius = 4.25
Area: 56.74

Circle printed as a Point is: [2.00, 2.00]
```

Fig. 19.8 Manejador para la clase **Circle** (parte 3 de 3).

estos son miembros protegidos de la clase base **Circle** (**x** y **y** fueron heredados por **Circle** a partir de **Point**). Note también que es necesario hacer referencia a **x**, **y** y **radius** mediante objetos, como en **c.x**, **c.y** y **c.radius**. Esto es debido a que la función de operador de inserción de flujo homónima no es una función miembro de la clase **Cylinder**, pero es un amigo de la clase. El programa manejador produce un objeto de la clase **Cylinder** y a continuación utiliza las funciones *get* para obtener la información respectiva del objeto **Cylinder**. Otra vez, **main** no es ni una función miembro ni un amigo de la clase **Cylinder**, por lo que no puede hacer referencia directa a los datos protegidos de la clase **Cylinder**. El programa manejador a continuación utiliza funciones *set* de nombre **setHeigt**, **setRadius** y **setPoint** para redefinir la altura, el radio y las coordenadas del cilindro. Por último, el manejador hace algo en especial interesante. Inicializa la variable de referencia **pRef** del tipo "referencia al objeto **Point**" (**Point &**) al objeto **cyl** de **Cylinder**. A continuación imprime **pRef** el cual, a pesar

del hecho que ha sido inicializado con un objeto **Cylinder**, "piensa" que se trata de un objeto **Point**, por lo tanto, el objeto **Cylinder** de hecho imprime un objeto **Point**. El manejador a continuación inicializa la variable de referencia **cRef** del tipo "referencia a un objeto **Circle**" (**Circle &**), al objeto **cyl** de **Cylinder**. A continuación imprime **cRef** el cual, a pesar del hecho de que ha sido inicializado con un objeto **Cylinder**, "piensa" que es un objeto **Circle**, por lo que el objeto **Cylinder** de hecho se imprime como un objeto **Circle**. También es extraída el área del **Circle**.

Este ejemplo demuestra muy bien la herencia pública y la definición y referencia de miembros de datos protegidos. A estas alturas el lector deberá sentirse familiarizado con los fundamentos de la herencia. En el siguiente capítulo mostraremos cómo programar con jerarquías de herencia de forma general mediante el polimorfismo. La abstracción de datos, la herencia y el polimorfismo forman el núcleo de la programación orientada a objetos.

19.15 Herencia múltiple

Hasta ahora en el capítulo hemos analizado la herencia simple, en la cual cada clase es derivada de una clase base. Una clase puede ser derivada a partir de más de una clase base; esta derivación se conoce como *herencia múltiple*. La herencia múltiple significa que una clase derivada hereda los miembros de varias clases base. Esta poderosa capacidad fomenta formas interesantes de reutilización de software, pero puede causar una variedad de problemas de ambigüedad.

Práctica sana de programación 19.2

La herencia múltiple, si se utiliza de forma adecuada, es una capacidad poderosa. La herencia múltiple deberá ser utilizada cuando exista una relación "es una" entre un nuevo tipo y dos o más tipos existentes (es decir, tipo A "es un" tipo B y "es un" tipo C).

```
// CYLINDR2.H
// Definition of class Cylinder
#ifndef CYLINDR2_H
#define CYLINDR2_H
#include "circle2.h"

class Cylinder : public Circle {
   friend ostream& operator<<(ostream&, const Cylinder&);
public:
   // default constructor
   Cylinder(float h = 0.0, float r = 0.0,
            float x = 0.0, float y = 0.0);
   void setHeight(float);   // set height
   float getHeight() const; // return height
   float area() const;      // calculate and return area
   float volume() const;    // calculate and return volume
protected:
   float height;            // height of the Cylinder
};

#endif
```

Fig. 19.9 Definición de clase **Cylinder** (parte 1 de 3).

```
// CYLINDR2.CPP
// Member and friend function definitions for class Cylinder
#include <iostream.h>
#include <iomanip.h>
#include "cylindr2.h"

// Cylinder constructor calls Circle constructor
Cylinder::Cylinder(float h, float r, float x, float y)
   : Circle(r, x, y)   // call base-class constructor
{ height = h; }

// Set height of Cylinder
void Cylinder::setHeight(float h) { height = h; }

// Get height of Cylinder
float Cylinder::getHeight() const { return height; }

// Calculate area of Cylinder (i.e., surface area)
float Cylinder::area() const
{
   return 2 * Circle::area() +
          2 * 3.14159 * radius * height;
}

// Calculate volume of Cylinder
float Cylinder::volume() const
   { return 3.14159 * radius * radius * height; }

// Output Cylinder dimensions
ostream& operator<<(ostream &output, const Cylinder& c)
{
   output << "Center = [" << c.x << ", " << c.y
          << "]; Radius = " << setiosflags(ios::showpoint)
          << setprecision(2) << c.radius
          << "; Height = " << c.height;

   return output;   // enables concatenated calls
}
```

Fig. 19.9 Definiciones de función miembro y de función amigo para la clase **Cylinder** (parte 2 de 3).

Considere el ejemplo de herencia múltiple de la figura 19.10. La clase **Base1** contiene un miembro de datos protegido **int value**. **Base1** contiene un constructor que define **value** y la función miembro pública **getData**, que lee **value**.

La clase **Base2** es similar a la clase **Base1**, excepto que sus datos protegidos son **char letter**. **Base2** también tiene una función miembro pública **getData**, pero su función lee el valor de **char letter**.

La clase **Derived** es heredada tanto de la clase **Base1** como de la clase **Base2** mediante herencia múltiple. **Derived** tiene el miembro de datos privados **float real**, y tiene la función miembro pública **getReal**, que lee el valor de **float real**.


```
// FIG19_9.CPP
// Driver for class Cylinder
#include <iostream.h>
#include <iomanip.h>
#include "point2.h"
#include "circle2.h"
#include "cylindr2.h"

main()
{
   // create Cylinder object
   Cylinder cyl(5.7, 2.5, 1.2, 2.3);

   // use get functions to display the Cylinder
   cout << "X coordinate is " << cyl.getX()
        << "\nYcoordinate is " << cyl.getY()
        << "\nRadius is " << cyl.getRadius()
        << "\nHeight is " << cyl.getHeight();

   // use set functions to change the Cylinder's attributes
   cyl.setHeight(10);
   cyl.setRadius(4.25);
   cyl.setPoint(2, 2);
   cout << "\n\nThe new location, radius, "
        << "and height of cyl are:\n" << cyl << endl << "Area: "
        << cyl.area() << "   Volume: " << cyl.volume();

   // display the Cylinder as a Point
   Point &pRef = cyl;   // pRef "thinks" it is a Point
   cout << "\n\nCylinder printed as a Point is: " << pRef;

   // display the Cylinder as a Circle
   Circle &cRef = cyl;  // cRef "thinks" it is a Circle
   cout << "\n\nCylinder printed as a Circle is:\n" << cRef
        << "\nArea: " << cRef.area() << endl;
   return 0;
}
```

```
X coordinate is 1.2
Y coordinate is 2.3
Radius is 2.5
Height is 5.7

The new location, radius, and height of cyl are:
Center = [2, 2]; Radius = 4.25; Height = 10.00
Area: 380.53   Volume: 567.45

Cylinder printed as a Point is: [2.00, 2.00]

Cylinder printed as a Circle is:
Center = [2.00, 2.00]; Radius = 4.25
Area: 56.74
```

Fig. 19.9 Manejador para la clase **Cylinder** (parte 3 de 3).

```
// BASE1.H
// Definition of class Base1
#ifndef BASE1_H
#define BASE1_H

class Base1 {
public:
   Base1(int x) { value = x; }
   int getData() const { return value; }
protected:      // accessible to derived classes
   int value;   // inherited by derived class
};

#endif
```

Fig. 19.10 Definición de la clase **Base1** (parte 1 de 6).

```
// BASE2.H
// Definition of class Base2
#ifndef BASE2_H
#define BASE2_H

class Base2 {
public:
   Base2(char c) { letter = c; }
   char getData() const { return letter; }
protected:        // accessible to derived classes
   char letter;   // inherited by derived class
};

#endif
```

Fig. 19.10 Definición de clase **Base2** (parte 2 de 6).

Note qué directo es indicar herencia múltiple haciendo seguir los dos puntos (**:**) después de **class Derived** con una lista separada por comas de clases base públicas. Note también que para cada una de sus clases base, el constructor **Derived** llama a los constructores de clase base **Base1** y **Base2**, mediante la sintaxis de inicializador de miembro.

El operador de inserción de flujo homónimo para **Derived** utiliza la notación punto sobre el objeto derivado **d** para imprimir **value**, **letter** y **real**. Esta función de operador es un amigo de **Derived**, por lo que **operator<<** puede tener acceso directo al miembro de datos privado **real** de **Derived**. También, dado que este operador es un amigo de una clase derivada, puede tener acceso a los miembros protegidos **value** y **letter** de **Base1** y de **Base2**, respectivamente.

Ahora examinemos el programa manejador en **main**. Creamos el objeto **b1** de clase **Base1** y lo inicializamos al valor **int 10**. Creamos el objeto **b2** de clase **Base2** y lo inicializamos al valor **char 'Z'**. Entonces, creamos el objeto **d** de la clase **Derived** y lo inicializamos para contener el valor **int 7**, el valor **char 'A'** y el valor **float 3.5**.

El contenido de cada uno de estos objetos de clase base es impreso utilizando ligaduras estáticas. Aún cuando existen dos funciones **getData**, no hay ambigüedad en las llamadas, porque hacen referencia directa a la versión **b1** del objeto de **getData** y a la versión **b2** de **getData**.


```
// DERIVED.H
// Definition of class Derived which inherits
// multiple base classes (Base1 and Base2).
#ifndef DERIVED_H
#define DERIVED_H

#include <iostream.h>
#include "base1.h"
#include "base2.h"

// multiple inheritance
class Derived : public Base1, public Base2 {
   friend ostream &operator<<(ostream &, const Derived &);
public:
   Derived(int, char, float);
   float getReal() const;
private:
   float real;    // derived class's private data
};

#endif
```

Fig. 19.10 Definición de clase **Derived** (parte 3 de 6).

```
// DERIVED.CPP
// Member function definitions for class Derived
#include "derived.h"

// Constructor for Derived calls constructors for
// class Base1 and class Base2.
Derived::Derived(int i, char c, float f)
   : Base1(i), Base2(c)  // call both base-class constructors
{ real = f; }

// Return the value of real
float Derived::getReal() const { return real; }

// Display all the data members of Derived
ostream &operator<<(ostream &output, const Derived &d)
{
   output << "    Integer: " << d.value
          << "\n  Character: " << d.letter
          << "\nReal number: " << d.real;

   return output;   // enables concatenated calls
}
```

Fig. 19.10 Definición de funciones miembro para la clase **Derived** (parte 4 de 6).

A continuación imprimimos el contenido del objeto **d** de **Derived** mediante ligadura estática. Pero aquí sí existe un problema de ambigüedad, porque este objeto contiene dos funciones **getData**, una heredada de **Base1** y la otra de **Base2**. Este problema es fácil de resolver mediante

```
// FIG19_10.CPP
// Driver for multiple inheritance example
#include <iostream.h>
#include "base1.h"
#include "base2.h"
#include "derived.h"

main()
{
   Base1 b1(10), *base1Ptr;   // create base-class object
   Base2 b2('Z'), *base2Ptr;  // create other base-class object
   Derived d(7, 'A', 3.5);    // create derived-class object

   // print data members of base class objects
   cout << "Object b1 contains integer "
        << b1.getData()
        << "\nObject b2 contains character "
        << b2.getData()
        << "\nObject d contains:\n" << d;

   // print data members of derived class object
   // scope resolution operator resolves getData ambiguity
   cout << "\n\nData members of Derived can be"
        << " accessed individually:\n"
        << "    Integer: " << d.Base1::getData()
        << "\n  Character: " << d.Base2::getData()
        << "\nReal number: " << d.getReal() << "\n\n";

   cout << "Derived can be treated as an "
        << "object of either base class:\n";

   // treat Derived as a Base1 object
   base1Ptr = &d;
   cout << "base1Ptr->getData() yields "
        << base1Ptr->getData();

   // treat Derived as a Base2 object
   base2Ptr = &d;
   cout << "\nbase2Ptr->getData() yields "
        << base2Ptr->getData() << endl;

   return 0;
}
```

Fig. 19.10 Manejador para el ejemplo de herencia múltiple (parte 5 de 6).

el operador de resolución de alcance binario, como **d.Base1::getData()** para imprimir el **int** en **value**, y **d.Base2::getData()** para imprimir el **char** en **letter**. En **real** el valor **float** es impreso sin ambigüedades mediante la llamada **d.getReal()**.

A continuación demostramos que las relaciones *es una* de herencia simple también son aplicables a la herencia múltiple. Asignamos la dirección del objeto derivado **d** al apuntador de clase base **base1Ptr**, e imprimimos **int value** invocando la función miembro **getData** de **Base1** desde **base1Ptr**. A continuación asignamos la dirección del objeto derivado **d** al


```
Object b1 contains integer 10
Object b2 contains character Z
Object d contains:
    Integer: 7
  Character: A
Real number: 3.5

Data members of Derived can be accessed individually:
    Integer: 7
  Character: A
Real number: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

Fig. 19.10 Manejador para el ejemplo de herencia múltiple (parte 6 de 6).

apuntador de clase base **base2Ptr** e imprimimos **char letter** invocando la función miembro **getData** de **Base2** partiendo de **base2Ptr**.

Este ejemplo mostró la mecánica de la herencia múltiple e introdujo un problema simple de ambigüedad. La herencia múltiple es un tema complejo, tratado en mayor detalle en textos de C++, como nuestro C++ *How to Program* (Prentice-Hall, 1994).

Resumen

- Una de las claves al poder de la programación orientada a objetos es conseguir la reutilización del software mediante la herencia.
- El programador puede determinar que la nueva clase debe heredar los miembros de datos y las funciones miembro de una clase base previa definida. En este caso, la nueva clase se conoce como la clase derivada.
- Con la herencia simple, una clase se deriva de una sola clase base. En la herencia múltiple, una clase derivada hereda de múltiples clases base (posiblemente no relacionadas).
- Una clase derivada por lo regular añade miembros de datos y funciones miembro propias, por lo que una clase derivada en general tiene una definición mayor que su clase base. Una clase derivada es más específica que su clase base y normalmente representa menos objetos.
- Una clase derivada no puede tener acceso a los miembros privados de su clase base; permitirlo violaría el encapsulado de la clase base. Una clase base puede, sin embargo, tener acceso a los miembros públicos y protegidos de su clase base.
- Al constructor de clase derivada siempre llamará primero al constructor de su clase base, a fin de crear y de inicializar los miembros de la clase base de la clase derivada.

- Los destructores serán llamados en orden inverso a las llamadas de constructor, por lo que un destructor de clase derivada será llamado antes del destructor de su clase base.
- La herencia permite la reutilización del software, lo que ahorra tiempo de desarrollo y fomenta el uso de software de alta calidad previamente probado y depurado.
- La herencia puede ser realizada a partir de bibliotecas de clases existentes.
- En algún momento el software será construido en su mayor parte partiendo de componentes estándar reutilizables, exactamente de la misma forma que como hoy en día se construye la mayor parte del hardware.
- El responsable de una puesta en práctica de una clase derivada no necesita tener acceso al código fuente de una clase base, pero necesita conocer la interfaz de la clase base y ser capaz de enlazar con el código objeto de la clase base.
- Un objeto de una clase derivada puede ser tratado como un objeto de su correspondiente clase base pública. Sin embargo, lo inverso no es cierto.
- Una clase base existe en una relación jerárquica con sus clases derivadas.
- Una clase puede existir por sí misma. Cuando dicha clase es utilizada con el mecanismo de herencia, se convierte ya sea en clase base, que proporciona atributos y comportamientos a otras clases, o la clase se convierte en una clase derivada, que hereda dichos atributos y comportamientos.
- Una jerarquía de herencia puede tener una profundidad arbitraria dentro de las limitaciones físicas de un sistema particular.
- Las jerarquías son herramientas útiles para comprender y administrar la complejidad. Siendo que el software se está haciendo cada vez más complejo, C++ proporciona mecanismos para apoyar estructuras jerárquicas mediante la herencia y el polimorfismo.
- Se puede utilizar una conversión explícita (cast) para convertir un apuntador de clase base a un apuntador de clase derivada. Si dicho apuntador debe ser desreferenciado, primero deberá hacerse que señale a un objeto del tipo de la clase derivada.
- El acceso protegido sirve como un nivel intermedio de protección entre el acceso público y el acceso privado. Se puede tener acceso a los miembros protegidos de una clase base por los miembros y amigos de la clase base y por los miembros y amigos de las clases derivadas; ninguna otra de las funciones pueden tener acceso a los miembros protegidos de la clase base.
- Los miembros protegidos pueden ser utilizados para extender privilegios a clases derivadas, en tanto que niegan dichos privilegios a funciones no de clase y no amigos.
- La herencia múltiple produce jerarquías arborescentes. Estas gráficas no tienen ciclos porque todas las flechas apuntan en la misma dirección. Dichas gráficas se conocen como gráficas acíclicas dirigidas (DAG).
- Al derivar una clase de una clase base, la clase base puede ser declarada **`public`**, **`protected`**, o **`private`**.
- Al derivar una clase de una clase base **`public`**, los miembros **`public`** de la clase base se convierten en miembros **`public`** de la clase derivada, y los miembros **`protected`** de la clase base se convierten en miembros **`protected`** de la clase derivada.

- Al derivar una clase de una clase base **`protected`**, los miembros **`public`** y **`protected`** de la clase base se convierten en miembros **`protected`** de la clase derivada.
- Al derivar una clase de la clase base **`private`**, los miembros **`public`** y **`protected`** de la clase base se convierten en miembros **`private`** de la clase derivada.
- Una clase base puede ser o una clase base directa de una clase derivada o una clase base indirecta de una clase derivada. Una clase base directa estará de forma explícita listada al declararse la clase derivada. Una clase base indirecta no está listada explícitamente; más bien es herencia a partir de varios niveles hacia arriba en la estructura de jerarquía arbórea.
- Cuando un miembro de clase base es inapropiado para una clase derivada, podemos redefinir dicho miembro en la clase derivada.
- Es importante distinguir entre relaciones "es un" y relaciones "tiene un". En una relación "tiene un", un objeto de clase tiene un objeto de otra clase como un miembro. En una relación "es un", un objeto de una clase de tipo derivado también puede ser tratado como un objeto de tipo de clase base. "Es un" es herencia. "Tiene un" es composición.
- Un objeto de clase derivada puede ser asignado a un objeto de clase base. Este tipo de asignación tiene sentido por que la clase derivada tiene miembros que corresponden a cada uno de los miembros de la clase base. Un apuntador a un objeto de clase derivada puede ser convertido implícitamente en un apuntador a un objeto de clase base.
- Es posible convertir un apuntador de clase base a un apuntador de clase derivada utilizando conversión explícita (cast). El destino debe ser un objeto de clase derivada.
- Una clase base especifica un estado común. Todas la clases derivadas de una clase base heredan las capacidades de dicha clase base. En el proceso de diseño orientado a objetos, el diseñador busca el estado común y lo segrega para formar clases base deseables. Las clases derivadas a continuación son personalizadas más allá de las capacidades heredadas de la clase base.
- Puede ser confuso leer un conjunto de declaraciones de clase derivada, porque no todos los miembros de la clase derivada están presentes en dichas declaraciones. En particular, los miembros heredados no están listados en las declaraciones de clase derivada, pero estos miembros están de hecho presentes en dichas clases derivadas.
- Las relaciones "tiene un" son ejemplos de creación de nuevas clases mediante composición de clase existentes.
- Los constructores de objeto miembro son llamados en el orden en el cual dichos objetos han sido declarados. En la herencia, los constructores de clase base son llamados en el orden en el cual se especificó la herencia, y antes del constructor de la clase derivada.
- Tratándose de un objeto de clase derivada, primero se llama al constructor de clase base, y a continuación el constructor de clase derivada.
- Cuando un objeto de clase derivada sale de alcance, se llaman a los destructores en orden inverso a la de los constructores —primero se llama al destructor de la clase derivada y a continuación al destructor de la clase base.
- Una clase puede ser derivada de más de una clase base; dicha derivación se llama herencia múltiple.

- Indique herencia múltiple haciendo seguir al indicador de herencia de dos puntos (`:`) con una lista separada por comas de clases base.
- El constructor de clase derivada llama a los constructores de clase base para cada una de sus clases base, mediante la sintaxis de inicializador de miembro.

Terminología

abstracción
ambigüedad en herencia múltiple
clase base
constructor por omisión de clase base
constructor de clase base
destructor de clase base
inicializador de clase base
apuntador de clase base
jerarquía de clase
biblioteca de clase
cliente de una clase
composición
personalizar software
clase derivada
constructor de clase derivada
destructor de clase derivada
apuntador de clase derivada
clase base directa
gráfica acíclica dirigida (DAG)
amigos de una clase base
amigos de una clase derivada
relación *tiene un*
relación jerárquica
clase base indirecta
error de recursión infinito
herencia
relación *es un*
relación *conoce a*
control de acceso de miembro
clase miembro
objeto miembro
herencia múltiple
programación orientada a objetos (OOP)
apuntador a un objeto de clase base
apuntador a un objeto de clase derivada
clase base privada
herencia privada
clase base protegida
derivación protegida
palabra reservada `protected`
miembro protegido de una clase
clase base pública
herencia pública
redefinición de un miembro de clase base
herencia simple
reutilización de software
componentes estándar de software
subclase
superclase
relación *utiliza un*

Errores comunes de programación

19.1 Puede causar errores tratar un objeto de clase base como si fuera un objeto de clase derivada.

19.2 Efectuar la conversión explícita de un apuntador de clase base que señala a un objeto de clase base a un apuntador de clase derivada y a continuación hacer referencia a miembros de clase derivada que no existen en dicho objeto.

19.3 Cuando se redefine una función miembro de clase base en una clase derivada, es común hacer que la versión de clase derivada llame a la versión de clase base y haga algún trabajo adicional. Causa recursión infinita no utilizar el operador de resolución de alcance para hacer referencia a la función miembro de la clase base, porque la función miembro de la clase derivada, de hecho, está llamando a sí misma.

19.4 Escribir un programa que dependa de que los objetos miembros sean construidos en un orden particular, puede producir errores sutiles.

19.5 Asignar un objeto de clase derivada a un objeto de una clase base correspondiente, y a continuación intentar hacer referencia a miembros de sólo la clase derivada en el objeto nuevo de clase base.

19.6 Puede causar error convertir en forma explícita (cast) un apuntador de clase base a un apuntador de clase derivada, si a continuación dicho apuntador es utilizado para hacer referencia a un objeto de clase base que no tenga los miembros deseados de la clase derivada.

Prácticas sanas de programación

19.1 Al heredar capacidades que no son necesarias en la clase derivada, dichas capacidades se deben enmascarar mediante redefinición de las funciones.

19.2 La herencia múltiple, si se utiliza adecuadamente, es una capacidad poderosa. La herencia múltiple deberá ser utilizada cuando exista una relación "es una" entre un nuevo tipo y dos o más tipos existentes (es decir, tipo A "es un" tipo B y "es un" tipo C).

Sugerencia de rendimiento

19.1 Si las clases producidas mediante la herencia son más grandes de lo requerido, pudiera dar como resultado un desperdicio de memoria y de recursos de proceso.

Observaciones de ingeniería de software

19.1 Una clase derivada no puede tener acceso directo a miembros privados de su clase base.

19.2 Una redefinición de una función miembro de clase base en una clase derivada no necesaria debe tener la misma signatura que la función miembro de clase base.

19.3 Cuando en una clase derivada se cree un objeto, primero se ejecutará el constructor de clase base, después se ejecutarán los constructores de los objetos miembros correspondientes a la clase derivada, y después se ejecutará el constructor de la clase derivada. Los destructores serán llamados en orden inverso en el cual fueron llamados sus correspondientes constructores.

19.4 Crear una clase derivada no afecta el código fuente o el código objeto de su clase base; mediante la herencia se conserva la integridad de la clase base.

19.5 En un sistema orientado a objetos, las clases a menudo están íntimamente relacionadas. "Disgregue y elimine" atributos y comportamientos comunes y colóquelos en una clase base. A continuación utilice la herencia para formar clases derivadas.

19.6 Una clase derivada contiene los atributos y comportamientos de su clase base. Una clase derivada también puede contener atributos y comportamientos adicionales. Con la herencia, la clase base puede ser compilada independiente de la clase derivada. Para tener la capacidad de combinar estos atributos y comportamientos adicionales con la clase base para formar la clase derivada basta compilar los atributos y comportamientos incrementados de la clase derivada.

19.7 Las modificaciones a una clase base no requieren que sea modificada la clase derivada, siempre y cuando la interfaz pública de la clase base se mantenga sin modificación. Pudiera sin embargo, ser necesario recompilar las clases derivadas.

19.8 Las modificaciones a una clase miembro no requieren que su clase compuesta que la encierra sea modificada, siempre y cuando se mantenga sin modificación la interfaz pública de la clase miembro. Note que pudiera ser necesario recompilar la clase compuesta.

Ejercicios de autoevaluación

19.1 Llene cada uno de los siguientes espacios en blanco:

a) Si la clase `Alpha` hereda de la clase `Beta`, la clase `Alpha` se llama la clase _____ y la clase `Beta` se llama la clase _____.

b) C++ proporciona _____, lo que permite que una clase derivada herede de muchas clases base, aún si dichas clases base no están relacionadas.
c) La herencia permite _____ lo que ahorra tiempo en el desarrollo, y fomenta el uso de software de alta calidad previamente probado.
d) Un objeto de una clase _____ puede ser tratado como si fuera un objeto de su correspondiente clase _____.
e) Para convertir un apuntador de clase base a un apuntador de clase derivada deberá utilizarse un _____ porque el compilador considera lo anterior una operación peligrosa.
f) Los tres especificadores de acceso de miembros son _____, _____ y _____.
g) Al derivar una clase de una clase base con herencia `public`, los miembros `public` de la clase se convierten en miembros _____ de la clase derivada, y los miembros `protected` de la clase base se convierten en miembros _____ de la clase derivada.
h) Al derivar una clase de una clase base con herencia `protected`, los miembros `public` de la clase base se convierten en miembros _____ de la clase derivada, y los miembros `protected` de la clase base e convierten en miembros _____ de la clase derivada.
i) Una relación "tiene un" entre clases representa _____ y una relación "es un" entre clases representa _____.

Respuesta a los ejercicios de autoevaluación

19.1 a) derivado, base. b) herencia múltiple. c) reutilización del software. d) derivada, base. e) conversión explícita (cast). f) `public, protected, private`. g) `public protected`. h) `protected`, `protected`. i) composición, herencia.

Ejercicios

19.2 Considere la clase `bicycle`. En base en sus conocimientos de algunos componentes básicos y comunes de las bicicletas, muestre una jerarquía de clases en la cual la clase `bicycle` hereda de otras clases, las cuales a su vez también heredan de otras clases. Analice la producción de varios objetos de la clase `bicycle`. Analice la herencia correspondiente a la clase `bicycle` para otras clases derivadas íntimamente relacionadas.

19.3 Defina brevemente cada uno de los términos siguientes: herencia, herencia múltiple, clase base y clase derivada.

19.4 Analice el por qué se considera peligroso para el compilador la conversión del apuntador de clase base a un apuntador de clase derivada.

19.5 Señale todas las formas que le vengan a la mente tanto de dos como de tres dimensiones, y organice estas formas en una jerarquía de formas. Su jerarquía deberá tener una clase base `Shape`, a partir de la cual se derivarán la clase `TwoDimensionalShape` y la clase `ThreeDimensionalShape`. Una vez desarrollada la jerarquía, defina cada una de las clases en la jerarquía. Utilizaremos esta jerarquía en los ejercicios del capítulo 20 para procesar todas las formas como objetos de la clase base `Shape`. Esta es una técnica conocida como polimorfismo.

19.6 Un tipo popular de clase es la que se llama una clase colección o una clase contenedor. Una clase de este tipo contiene elementos de otras clases. Algunos tipos de clases de colección son los arreglos, las pilas, las colas, las listas enlazadas, los árboles, las cadenas de caracteres, los conjuntos, las bolsas, los diccionarios, las tablas, etcétera. Una clase contenedor o de colección proporciona en forma típica servicios como es insertar un elemento, borrarlo y buscarlo, combinar dos conjuntos o colecciones, determinar la intersección de dos colecciones (es decir, los elementos que sean comunes a ambas), imprimir una colección, encontrar el elemento más grande en la colección, encontrar el elemento más pequeño en la colección, encontrar la suma de los elementos de la colección, etcétera.

a) Haga una lista de todos los tipos de clases de colección que se le puedan ocurrir (incluyendo aquellos que ya hemos mencionado).
b) Arregle estas clases de colección en una jerarquía de clases. Quizás desee distinguir entre colecciones ordenadas y colecciones desordenadas.
c) Ponga en práctica todas las clases que pueda, utilizando la herencia para minimizar la cantidad de código nuevo que deba escribir para la creación de cada una de las nuevas clases.
d) Escriba un programa manejador que pruebe cada una de las clases en su jerarquía de herencia.

20

Funciones virtuales y polimorfismo

Objetivos

- Comprender el concepto de polimorfismo.
- Comprender cómo declarar y utilizar las funciones virtuales para conseguir el polimorfismo.
- Comprender la diferencia entre clases abstractas y clases concretas.
- Aprender a declarar funciones virtuales puras a fin de crear clases abstractas.
- Valorizar cómo el polimorfismo consigue que los sistemas sean extensibles y mantenibles.

¡Oh! tiene usted una repetición maldecible, y es ciertamente capaz de corromper a un santo.
William Shakespeare
Henry IV, Part I, 1, ii

Proposiciones generales no deciden los casos concretos.
Oliver Wendell Holmes

Un filósofo de imponente estatura no piensa en el vacío. Inclusive sus más abstractas ideas están, hasta cierto punto, condicionadas por lo que es o no sabido en la era en que vive.
Alfred North Whitehead

Sinopsis

20.1 Introducción

Mediante las *funciones virtuales* y el *polimorfismo*, se hace posible diseñar y poner en práctica sistemas que son de mayor facilidad extensibles. Se pueden escribir programas para procesar objetos de clases existentes y de clases que aun no existan cuando el programa esté en desarrollo. Si esas clases deben ser derivadas de clases base que el programa conozca, éste puede proporcionar un marco general para manipular los objetos de las clases base, y los objetos de las clases derivadas encajarán bien dentro de este marco. De hecho, basándose en este principio, productos completos son elaborados, conocidos como marcos de aplicación.

20.2 Campos de tipo y enunciados switch

Una forma de tratar los objetos de muchos tipos distintos es utilizar un enunciado **`switch`**, a fin de tomar la acción apropiada sobre cada objeto, basándose en su tipo. Al utilizar la lógica **`switch`** se presentan muchos problemas. Se le podría olvidar al programador llevar a cabo la correspondiente prueba de tipo, cuando sea requerida. También se le podría olvidar probar todos los casos posibles, incluidos en un **`switch`**. Si se modifica un sistema basado en **`switch`**, el programador podría olvidar insertar los nuevos casos en los enunciados **`switch`** existentes. Toda modificación a los enunciados **`switch`** efectuados para manejar nuevos tipos, exigen que sea modificado cada enunciado **`switch`** del sistema; rastrear dichos enunciados puede resultar tardado y sujeto a errores.

Como veremos, las funciones virtuales y la programación polimórfica puede eliminar la necesidad de la lógica **`switch`**. El programador puede utilizar el mecanismo de las funciones virtuales para ejecutar en forma automática la lógica equivalente, evitando por lo tanto los tipos de errores asociados típicamente con la lógica **`switch`**.

Observación de ingeniería de software 20.1

Una consecuencia interesante del uso de funciones virtuales y polimorfismo es que los programas adquieren una apariencia simplificada. Contienen menos lógica de bifurcación, prefiriendo un código secuencial más sencillo. Esta simplificación facilita probar, depurar y mantener el programa.

20.3 Funciones virtuales

Suponga un conjunto de clases de forma, como **`Circle`**, **`Triangle`**, **`Rectangle`**, **`Square`**, etcétera, todas ellas derivadas de la clase base **`Shape`**. En la programación orientada a objetos, cada una de estas clases estaría investida con la capacidad de dibujarse a sí misma. Aunque cada clase tiene su propia función **`draw`**, la función **`draw`** correspondiente a cada forma es bien distinta. Cuando se dibuja una forma, cualquiera que ésta sea, sería agradable tener la capacidad de tratar todas estas formas en una genérica, como objetos de la clase base **`Shape`**. Entonces al dibujar cualquier forma, sólo llamaríamos a la función **`draw`** de la clase base **`Shape`**, y dejaríamos que el programa determinase en forma dinámica (es decir, en tiempo de ejecución) cual de las funciones **`draw`**, de las clases derivadas, utilizar.

Para habilitar este tipo de comportamiento, declaramos **`draw`** en la clase base en forma de una *función virtual*, a continuación, en cada una de las clases derivadas *redefinimos* **`draw`**, a fin de que se dibuje la forma apropiada. Se declara una función virtual en la clase base precediendo el prototipo de función con la palabra reservada **`virtual`**.

Observación de ingeniería de software 20.2

Una vez declarada una función como virtual, se conserva como virtual a lo largo de toda la jerarquía de herencia, a partir de dicho punto.

Práctica sana de programación 20.1

A pesar que ciertas funciones en forma implícita son virtuales, en razón de una declaración ya hecha en la jerarquía de clase, algunos programadores prefieren declarar estas funciones virtuales en forma explícita en cada uno de los niveles de la jerarquía, a fin de promover claridad en el programa.

Observación de ingeniería de software 20.3

Cuando una clase derivada decide no definir una función virtual, la clase derivada sólo heredará la función virtual de la clase base inmediata.

Si en la clase base la función **`draw`** ha sido declarada **`virtual`**, y si entonces utilizamos un apuntador de clase base para señalar al objeto de la clase derivada y utilizando este apuntador invocamos la función **`draw`**, el programa seleccionará de manera dinámica (es decir, en tiempo de ejecución) la función correcta **`draw`** de la clase derivada. Esto se conoce como *ligadura dinámica* (vea las figuras 20.1 y 20.2).

Observación de ingeniería de software 20.4

Las funciones virtuales redefinidas deben tener el mismo tipo de regreso y la misma firma que la función virtual base.

Error común de programación 20.1

Resultará un error de sintaxis redefinir una clase derivada como función virtual de clase base, sin asegurarse que la función derivada tenga el mismo tipo de regreso y signatura que la versión de clase base.

Si una función virtual es llamada haciendo referencia a un objeto específico por nombre, y utilizando el operador de selección miembro punto, la referencia se resuelve en tiempo de compilación (esto se llama *ligadura estática*) y la función virtual llamada es aquella definida para, (o heredada por) la clase de dicho objeto en particular.

La homonimia no utiliza ligadura dinámica. Más bien, la homonimia es resuelta en tiempo de compilación al seleccionar la definición de función con una firma que coincida con la llamada de función (utilizando posiblemente una conversión implícita cast de tipo para que la coincidencia ocurra). Esto corresponde a ligadura estática.

20.4 Clases base abstractas y clases concretas

Cuando pensamos sobre una clase como un tipo, suponemos que serán producidos objetos de dicho tipo. Existen, sin embargo, muchas situaciones en las cuales resulta útil definir clases para las cuales el programador no tiene intención de producir ningún objeto. Estas clases se conocen como *clases abstractas*. Dado que en situaciones de herencia son utilizadas como clases base, a menudo nos referiremos a ellas como *clases base abstractas*. A partir de una clase base abstracta no se pueden producir objetos.

El único fin de una clase abstracta es proporcionar una clase base apropiada, a partir de la cual las clases pueden heredar interfaz y/o puesta en práctica. Las clases a partir de las cuales los objetos se pueden producir, se conocen como *clases concretas*.

Podríamos tener una clase base abstracta **`TwoDimensionalObject`** y derivar clases concretas, como **`Square`**, **`Circle`**, **`Triangle`**, etcétera. También podríamos tener una clase base abstracta **`ThreeDimensionalObject`** y derivar clases concretas como **`Cube`**, **`Sphere`**, **`Cylinder`**, **`Pyramid`**, etcétera. Estas clases base abstractas resultan demasiado genéricas para definir objetos reales; antes de pensar en la producción de objetos necesitamos ser más específicos. Y esto es lo que hacen las clases concretas; dan la definición que convierte en razonable la producción de objetos.

Una clase con funciones virtuales se hace abstracta al declarar uno o más de sus funciones virtuales como puras. Una *función virtual pura* es una que en su declaración contenga un *inicializador de* **`= 0`**, como en

```
virtual float earnings() const = 0;     // pure virtual
```

Observación de ingeniería de software 20.5

Si una clase se deriva de una clase con una función virtual pura, y para dicha función virtual pura no se ha dado definición en la clase derivada, entonces la función virtual también se conserva pura en la clase derivada. Por consecuencia, también la clase derivada será una clase abstracta.

Error común de programación 20.2

Es un error de sintaxis intentar producir un objeto a partir de una clase abstracta (es decir, de una clase que contenga una o más funciones virtuales puras).

Una jerarquía no necesita contener ninguna clase abstracta, pero como veremos, mucho sistemas de calidad orientados a objetos tienen jerarquías de clase encabezadas por una clase abstracta. En algunos casos, las clases abstractas forman los primeros niveles superiores de la jerarquía. Un buen ejemplo de lo anterior es la jerarquía de forma. La jerarquía podría ser encabezada por la clase base abstracta **`Shape`**. En el siguiente nivel inferior, podríamos tener dos

clases base más, de tipo abstracto, es decir **`TwoDimensionalShape`** y **`ThreeDimensionalShape`**. El siguiente nivel hacia abajo empezaría a definir clases concretas correspondientes a formas de dos dimensiones, como círculos, cuadrados y clases concretas correspondientes a formas tridimensionales, como esferas y cubos.

20.5 Polimorfismo

C++ permite el *polimorfismo* —la capacidad de objetos de clases diferentes, relacionados mediante la herencia, a responder de forma distinta a una misma llamada de función miembro. Si, por ejemplo, la clase **`Rectangle`** es derivada de la clase **`Quadrilateral`**, entonces un objeto **`Rectangle`** *es una* versión más específica de un objeto **`Quadrilateral`**. Una operación (como el cálculo del perímetro o de la superficie) que pueda ser efectuada en un objeto de la clase **`Quadrilateral`**, también podrá ser ejecutada en un objeto de la clase **`Rectangle`**.

El polimorfismo se pone en práctica vía las funciones virtuales. Cuando se hace una solicitud, a través de un apuntador de clase base (o una referencia) para usar una función virtual, C++ escoge la función redefinida correcta, en la clase derivada apropiada, asociada con el objeto.

Algunas veces, una función miembro no virtual es definida en una clase base y redefinida en una derivada. Si una función miembro como ésta es llamada mediante un apuntador de clase base, se utilizará la versión de clase base. Si la función miembro es llamada mediante un apuntador de clase derivada, se utilizará la versión de clase derivada. Esto es comportamiento no polimórfico.

Mediante el uso de las funciones virtuales y del polimorfismo, una llamada de función miembro puede hacer que ocurran distintas acciones, dependiendo del tipo del objeto que reciba la llamada. Esto le da al programador una capacidad de expresión tremenda. En las siguientes secciones veremos ejemplos del poder del polimorfismo y de las funciones virtuales.

Observación de ingeniería de software 20.6

Mediante las funciones virtuales y el polimorfismo, el programador puede ocuparse de generalidades y dejar que en tiempo de ejecución, el entorno se preocupe de lo específico. El programador puede dirigir una amplia variedad de objetos haciendo que se comporten de formas apropiadas a dichos objetos incluso sin conocer los tipos de los mismos.

Observación de ingeniería de software 20.7

El polimorfismo fomenta la extensibilidad: software escrito para invocar comportamiento polimórfico se escribe en forma independiente del tipo de los objetos a los cuales los mensajes son enviados. Por lo tanto, nuevos tipos de objetos, que pudieran responder a mensajes existentes, pueden ser añadidos en dicho sistema sin modificar el sistema base. A excepción de la parte de código cliente que produce nuevos objetos, los programas no necesitan ser recompilados.

Observación de ingeniería de software 20.8

Una clase abstracta define una interfaz para los distintos miembros de una jerarquía de clase. La clase abstracta contiene funciones virtuales puras, que serán definidas en las clases derivadas. Mediante el polimorfismo todas las funciones en la jerarquía pueden utilizar esta misma interfaz.

A pesar de que no podemos producir objetos de clases base abstractas, sí *podemos* declarar apuntadores a clases base abstractas. Dichos apuntadores podrán ser utilizados después para permitir manipulaciones polimórficas de objetos de clases derivadas, cuando dichos objetos son producidos a partir de clases concretas.

Consideremos aplicaciones del polimorfismo y de las funciones virtuales. Un administrador de pantalla necesita mostrar una variedad de objetos, incluyendo nuevos tipos de objetos, que serán añadidos al sistema, aún después de que haya sido escrito el administrador de pantalla. El sistema pudiera necesitar mostrar varias formas (es decir, la clase base es **Shape**) como cuadrados, círculos, triángulos, rectángulos y similar (cada una de estas clases de formas es derivada de la clase base **Shape**). El administrador de pantalla utiliza apuntadores de clase base para administrar todos los objetos a mostrar. Para dibujar cualquier objeto (independiente del nivel en la jerarquía de herencia en el cual aparezca dicho objeto), el administrador de pantalla utiliza un apuntador de clase base a dicho objeto, y envía al objeto un mensaje **draw**. En la clase base **Shape** la función **draw** ha sido declarada como virtual pura y en cada una de las clases derivadas ha sido redefinida. Cada objeto sabe como dibujarse a sí mismo. El administrador de pantalla no tiene que preocuparse sobre estos detalles; le indica a cada objeto que se dibuje a sí mismo.

El polimorfismo es en particular efectivo para la puesta en práctica de sistemas de software en capas. Por ejemplo, en los sistemas operativos, cada tipo de dispositivo físico pudiera operar en forma bastante distinta a los demás. Independiente de lo anterior, órdenes o comandos para *leer* o *escribir* datos de y hacia los dispositivos pueden tener una cierta uniformidad. El mensaje *write* enviado a un objeto manejador de dispositivo, necesita ser interpretado en específico en el contexto de dicho manejador de dispositivo, y de la forma en que dicho manejador de dispositivo manipula dispositivos de un tipo específico. Pero la llamada *write* por sí misma en realidad no es distinta de la llamada *write* para cualquier otro dispositivo —simplemente colocar cierto número de bytes de la memoria en dicho dispositivo. Un sistema operativo orientado a objetos pudiera utilizar una clase base abstracta para proporcionar una interfaz apropiada para todos los manejadores de dispositivo. Entonces, mediante herencia, a partir de dicha clase base abstracta, se formarían clases derivadas, todas operando en forma similar. Las capacidades (es decir, la interfaz pública) ofrecida por los manejadores de dispositivo se dan en la clase base abstracta como funciones virtuales puras. Las puestas en práctica de estas funciones virtuales se dan en las clases derivadas, correspondientes a los tipos específicos de manejadores de dispositivo.

En el capítulo 17, introdujimos el concepto de los iteradores. Es común definir una *clase iterador*, que recorra todos los objetos de una colección. Si por ejemplo, usted desea imprimir una lista de objetos en una lista enlazada, se puede producir un objeto iterador, que, cada vez que se llame al iterador, devolverá el siguiente elemento de la lista enlazada. Los iteradores son por lo general usados en la programación polimórfica para recorrer una lista enlazada de objetos, correspondiente a varios niveles de una jerarquía. Los apuntadores a una lista como ésta, serían todos apuntadores de clase base.

20.6 Estudio de caso: un sistema de nóminas utilizando polimorfismo

Usemos funciones virtuales y polimorfismo para llevar a cabo cálculos de nómina, basados en el tipo de un empleado (figura 20.1). Utilizamos una clase base **Employee**. Las clases derivadas de **Employee** son **Boss**, mismo que se le paga un salario semanal fijo, independiente del número de horas trabajadas, **CommissionWorker**, que obtiene un salario base fijo más un porcentaje de las ventas, **PieceworkWorker**, a quien se le paga según el número de elementos producidos, y **HourlyWorker**, que cobra por hora y recibe paga por tiempo extraordinario.

Una llamada de función **earnings** es ciertamente aplicable en forma genérica a todos los empleados. Pero la forma en que se calculen los ingresos de cada persona dependerá de la clase del empleado y estas clases son todas derivadas de la clase base **Employee**. Por lo tanto, en la clase base **Employee**, **earnings** es declarado **virtual**, y se dan puestas en práctica apropiadas

de **earnings** para cada una de las clases derivadas. A continuación, a fin de calcular los ingresos de cada empleado, el programa sólo utiliza un apuntador de clase base al objeto de dicho empleado, e invoca la función **earnings**. En un sistema real de nóminas, los distintos objetos "empleado" pudieran estar almacenados en una lista enlazada, en la cual, cada uno de los apuntadores es del tipo **Employee ***. El programa entonces sólo recorrería la lista enlazada, nodo por nodo, utilizando los apuntadores **Employee *** a fin de invocar la función **earnings** sobre cada objeto.

Veamos la clase **Employee** (figura 20.1, parte 1 y 2). Las funciones miembro públicas incluyen un constructor, que toma como argumentos el nombre y el apellido; un destructor, que recupera la memoria asignada dinámicamente; una función *get*, que devuelve el nombre; una función *get*, que devuelve el apellido; y, por último, dos funciones virtuales puras **earnings** y **print**. ¿Por qué estas funciones son virtuales puras? La respuesta es que no tiene sentido hacer puestas en práctica de dichas funciones en la clase **Employee**. Al hacer estas funciones virtuales puras, estamos indicando que proporcionaremos puestas en práctica en las clase derivada, pero no en la clase base misma. No es posible calcular los ingresos de un empleado genérico —primero tenemos que saber qué tipo de empleado es— ni tampoco, a partir de un empleado genérico, podemos imprimir el tipo de empleado.

La clase **Boss** (figura 20.1, parte 3 y 4) es derivada de **Employee** mediante herencia pública. Las funciones miembro públicas incluyen un constructor, que toma como argumentos un nombre, un apellido y un salario semanal, y pasa el nombre y el apellido al constructor **Employee** para inicializar los miembros **firstName** y **lastName** de la parte de clase base del objeto de clase derivada; una función *set*, para asignar un valor nuevo al miembro de datos privado **weeklySalary**; una función virtual **earnings**, que define cómo calcular los ingresos **Boss**; y una función virtual **print**, que extrae el tipo del empleado, seguido por el nombre del mismo.

```
// EMPLOY2.H
// Abstract base class Employee
#ifndef EMPLOY2_H
#define EMPLOY2_H

class Employee {
public:
   Employee(const char *, const char *);
   ~Employee();
   const char *getFirstName() const;
   const char *getLastName() const;

   // Pure virtual functions make
   // Employee an abstract base class.
   virtual float earnings() const = 0; // pure virtual
   virtual void print() const = 0;     // pure virtual
private:
   char *firstName;
   char *lastName;
};

#endif
```

Fig. 20.1 Clase base abstracta **Employee** (parte 1 de 12).

```
// EMPLOY2.CPP
// Member function definitions for
// abstract base class Employee.
//
// Note: No definitions given for pure virtual functions.
#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "employ2.h"

// Constructor dynamically allocates space for the
// first and last name and uses strcpy to copy
// the first and last names into the object.
Employee::Employee(const char *first, const char *last)
{
   firstName = new char[ strlen(first) + 1 ];
   assert(firstName != 0);     // test that new worked
   strcpy(firstName, first);

   lastName = new char[ strlen(last) + 1 ];
   assert(lastName != 0);      // test that new worked
   strcpy(lastName, last);
}

// Destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   delete [] firstName;
   delete [] lastName;
}

// Return a pointer to the first name
const char *Employee::getFirstName() const
{
   // Const prevents caller from modifying private data.
   // Caller should copy returned string before destructor
   // deletes dynamic storage to prevent undefined pointer.

   return firstName;   // caller must delete memory
}

// Return a pointer to the last name
const char *Employee::getLastName() const
{
   // Const prevents caller from modifying private data.
   // Caller should copy returned string before destructor
   // deletes dynamic storage to prevent undefined pointer.

   return lastName;   // caller must delete memory
}
```

Fig. 20.1 Definiciones de función miembro para la clase base abstracta **Employee** (parte 2 de 12).


```
// BOSS1.H
// Boss class derived from Employee
#ifndef BOSS1_H
#define BOSS1_H
#include "employ2.h"

class Boss : public Employee {
public:
   Boss(const char *, const char *, float = 0.0);
   void setWeeklySalary(float);
   virtual float earnings() const;
   virtual void print() const;
private:
   float weeklySalary;
};

#endif
```

Fig. 20.1 Clase **Boss** derivada de la clase base abstracta **Employee** (parte 3 de 12).

```
// BOSS1.CPP
// Member function definitions for class Boss
#include <iostream.h>
#include "boss1.h"

// Constructor function for class Boss
Boss::Boss(const char *first, const char *last, float s)
   : Employee(first, last)  // call base-class constructor
{ weeklySalary = s > 0 ? s : 0; }

// Set the Boss's salary
void Boss::setWeeklySalary(float s)
{ weeklySalary = s > 0 ? s : 0; }

// Get the Boss's pay
float Boss::earnings() const { return weeklySalary; }

// Print the Boss's name
void Boss::print() const
{
   cout << "\n             Boss: " << getFirstName()
        << ' ' << getLastName();
}
```

Fig. 20.1 Definiciones de función miembro para la clase **Boss** (parte 4 de 12).

La clase `CommissionWorker` (figura 20.1, parte 5 de 6) se deriva de **`Employee`** mediante herencia pública. Las funciones miembro públicas incluyen un constructor, que toma como argumentos un nombre, un apellido, un salario, una comisión y una cantidad de elementos vendidos y pasa el nombre y el apellido al constructor **`Employee`**, a fin de inicializar los miembros **`firstName`** y **`lastName`** de la parte de clase base del objeto de clase derivada; funciones *set*, para asignar nuevos valores a los miembros de datos privados **`salary`**, **`commission`** y **`quantity`**;

```
// COMMIS1.H
// CommissionWorker class derived from Employee
#ifndef COMMIS1_H
#define COMMIS1_H
#include "employ2.h"

class CommissionWorker : public Employee {
public:
   CommissionWorker(const char *, const char *,
                    float = 0.0, float = 0.0, int = 0);
   void setSalary(float);
   void setCommission(float);
   void setQuantity(int);
   virtual float earnings() const;
   virtual void print() const;
private:
   float salary;     // base salary per week
   float commission; // amount per item sold
   int quantity;     // total items sold for week
};

#endif
```

Fig. 20.1 Clase **CommissionWorker** derivada de la clase base abstracta **Employee** (parte 5 de 12).

una función virtual **earnings**, que define cómo calcular los ingresos **CommissionWorker**; y una función virtual **print**, que extrae el tipo del empleado seguido por el nombre del mismo.

La clase **PieceWorker** (figura 20.1, partes 7 y 8) se deriva de **Employee** mediante herencia pública. Las funciones miembro públicas incluyen un constructor, que toma como argumentos un nombre, un apellido, un salario por pieza y una cantidad de elementos producidos, y pasa el nombre y apellido al constructor **Employee**, a fin de inicializar los miembros **firstName** y **lastName** de la parte de clase base del objeto de clase derivada; funciones *set*, para asignar nuevos valores a los miembros de datos privados **wagePerPiece** y **quantity**; una función virtual **earnings**, que define cómo calcular los ingresos **PieceWorker**; y una función virtual **print**, que extrae el tipo del empleado seguido por el nombre del mismo.

La clase **HourlyWorker** (figura 20.1, partes 9 y 10) se deriva de **Employee** mediante herencia pública. Las funciones miembro públicas incluyen un constructor, que toma como argumentos un nombre, apellido, un salario, el número de horas trabajadas y pasa el nombre y el apellido al constructor **Employee**, a fin de inicializar los miembros **firstName** y **lastName** de la parte de clase base del objeto de clase derivada; funciones *set*, a fin de asignar nuevos valores a los miembros de datos privados **wage** y **hours**; una función virtual **earnings**, que define cómo calcular los ingresos **HourlyWorker**; y una función virtual **print**, que extrae el tipo del empleado seguido por el nombre del mismo.

El programa manejador (figura 20.1 partes 11 y 12) empieza declarando el apuntador de clase base, **ptr**, del tipo **Employee ***. Cada uno de los tres segmentos de código en **main** es similar, por lo que analizaremos sólo el primer segmento, que trata del objeto **Boss**. La línea

```
Boss b("John", "Smith", 800.00);
```



```
// COMMIS1.CPP
// Member function definitions for class CommissionWorker
#include <iostream.h>
#include "commis1.h"

// Constructor for class CommissionWorker
CommissionWorker::CommissionWorker(const char *first,
   const char *last, float s, float c, int q)
   : Employee(first, last)  // call base-class constructor
{
   salary = s > 0 ? s : 0;
   commission = c > 0 ? c : 0;
   quantity = q > 0 ? q : 0;
}

// Set CommissionWorker's weekly base salary
void CommissionWorker::setSalary(float s)
{ salary = s > 0 ? s : 0; }

// Set CommissionWorker's commission
void CommissionWorker::setCommission(float c)
{ commission = c > 0 ? c : 0; }

// Set CommissionWorker's quantity sold
void CommissionWorker::setQuantity(int q)
{ quantity = q > 0 ? q : 0; }

// Determine CommissionWorker's earnings
float CommissionWorker::earnings() const
{ return salary + commission * quantity; }

// Print the CommissionWorker's name
void CommissionWorker::print() const
{
   cout << "\nCommission worker: " << getFirstName()
        << ' ' << getLastName();
}
```

Fig. 20.1 Definiciones de función miembro para la clase **CommissionWorker** (parte 6 de 12).

produce el objeto de clase derivada **b** de la clase **Boss** y proporciona varios argumentos de constructor incluyendo un nombre, un apellido y un salario fijo semanal. La línea

```
ptr = &b;  // base-class pointer to derived-class object
```

coloca la dirección de la clase derivada del objeto **b** en el apuntador de clase base **ptr**. Esto es precisamente lo que debemos hacer para utilizar comportamiento polimórfico. La línea

```
ptr->print();                                  // dynamic binding
```

invoca a la función miembro **print** del objeto al cual señala **ptr**. Dado que **print** ha sido declarado como una función virtual de la clase base, el sistema invoca la función **print** del

```
// PIECE1.H
// PieceWorker class derived from Employee
#ifndef PIECE1_H
#define PIECE1_H
#include "employ2.h"

class PieceWorker : public Employee {
public:
   PieceWorker(const char *, const char *,
               float = 0.0, int = 0);
   void setWage(float);
   void setQuantity(int)
   virtual float earnings() const;
   virtual void print() const;

private:
   float wagePerPiece; // wage for each piece output
   int quantity;       // output for week
};

#endif
```

Fig. 20.1 Clase **PieceWorker** derivada de la clase base abstracta **Employee** (parte 7 de 12).

objeto de clase derivada —otra vez precisamente lo que llamamos comportamiento polimórfico. Esta llamada de función es un ejemplo de ligadura dinámica— la función es invocada a través de un apuntador de clase base, por lo que la decisión de cuál función deberá invocarse se pospone hasta el tiempo de ejecución. La línea

```
cout << " earned $" << ptr->earnings(); // dynamic binding
```

invoca la función miembro **earnings** del objeto al cual señala **ptr**. Dado de que **earnings** ha sido declarada una función virtual de la clase base, el sistema invoca a la función **earnings** del objeto de clase derivada. También éste es un ejemplo de ligadura dinámica. La línea

```
b.print();                          // static binding
```

invoca en forma explícita a la función **Boss** de la función miembro **print**, mediante el uso del operador de selección miembro punto sobre el objeto **b** específico de **Boss**. Esto es un ejemplo de ligadura estática, porque en tiempo de compilación el tipo del objeto para el cual se llama a la función es ya conocido. Esta llamada ha sido incluida para efectos de comparación con el fin de ilustrar que la función **print** correcta es invocada mediante el uso de la ligadura dinámica. La línea

```
cout << " earned $" << b.earnings();   // static binding
```

invoca de manera explícita la versión **Boss** de la función **earnings**, mediante el uso del operador de selección de miembro punto sobre el objeto **b** específico de **Boss**. También este es un ejemplo de ligadura estática. También se incluye esta llamada para efectos de comparación, a fin de ilustrar que la función **earnings** correcta es invocada utilizando ligadura dinámica.


```
// PIECE1.CPP
// Member function definitions for class PieceWorker

#include <iostream.h>
#include "piece1.h"

// Constructor for class PieceWorker
PieceWorker::PieceWorker(const char *first, const char *last,
                         float w, int q)
   : Employee(first, last)  // call base-class constructor
{
   wagePerPiece = w > 0 ? w : 0;
   quantity = q > 0 ? q : 0;
}

// Set the wage
void PieceWorker::setWage(float w)
{ wagePerPiece = w > 0 ? w : 0; }

// Set the number of items output
void PieceWorker::setQuantity(int q)
{ quantity = q > 0 ? q : 0; }

// Determine the PieceWorker's earnings
float PieceWorker::earnings() const
{ return quantity * wagePerPiece; }

// Print the PieceWorker's name
void PieceWorker::print() const
{
   cout << "\n     Piece worker: " << getFirstName()
        << ' ' << getLastName();
}
```

Fig. 20.1 Definiciones de función miembro para la clase **PieceWorker** (parte 8 de 12).

20.7 Clases nuevas y ligadura dinámica

El polimorfismo y las funciones virtuales ciertamente funcionarían bien en un mundo en el cual todas las clases posibles fueran conocidas con antelación. Pero también funcionan cuando en forma regular a los sistemas se les añaden nuevos tipos de clases.

Se les hace sitio a las nuevas clases mediante la ligadura dinámica (también conocido como *ligadura tardía*). Para que sea compilada una llamada de función virtual, no es necesario conocer el tipo de un objeto en tiempo de compilación. La llamada de función virtual se hace coincidir en tiempo de ejecución con la función miembro del objeto llamado.

Un programa administrador de pantalla puede ahora manejar nuevos objetos de exibición, conforme sean añadidos al sistema, sin necesidad de recompilación. La llamada de función **draw** se conserva idéntica. Los nuevos objetos mismos contienen sus propias capacidades de dibujo. Esto facilita añadir con un impacto mínimo nuevas capacidades a sistemas. También promueve la reutilización del software.

```
// HOURLY1.H
// Definition of class HourlyWorker
#ifndef HOURLY1_H
#define HOURLY1_H
#include "employ2.h"

class HourlyWorker : public Employee {
public:
   HourlyWorker(const char *, const char *,
                float = 0.0, float = 0.0);
   void setWage(float);
   void setHours(float);
   virtual float earnings() const;
   virtual void print() const;
private:
   float wage;   // wage per hour
   float hours;  // hours worked for week
};

#endif
```

Fig. 20.1 La clase **HourlyWorker** derivada de la clase base abstracta **Employee** (parte 9 de 12).

```
// HOURLY1.CPP
// Member function definitions for class HourlyWorker
#include <iostream.h>
#include "hourly1.h"

// Constructor for class HourlyWorker
HourlyWorker::HourlyWorker(const char *first, const char *last,
                           float w, float h)
   : Employee(first, last)   // call base-class constructor
{
   wage = w > 0 ? w : 0;
   hours = h >= 0 && h < 168 ? h : 0;
}

// Set the wage
void HourlyWorker::setWage(float w) { wage = w > 0 ? w : 0; }

// Set the hours worked
void HourlyWorker::setHours(float h)
{ hours = h >= 0 && h < 168 ? h : 0; }

// Get the HourlyWorker's pay
float HourlyWorker::earnings() const { return wage * hours; }

// Print the HourlyWorker's name
void HourlyWorker::print() const
{
   cout << "\n    Hourly worker: " << getFirstName()
        << ' ' << getLastName();
}
```

Fig. 20.1 Definiciones de función miembro para las clases **HourlyWorker** (parte 10 de 12).


```
// FIG20_1.CPP
// Driver for Employee hierarchy

#include <iostream.h>
#include <iomanip.h>
#include "employ2.h"
#include "boss1.h"
#include "commis1.h"
#include "piece1.h"
#include "hourly1.h"

main()
{
   // set output formatting
   cout << setiosflags(ios::showpoint) << setprecision(2);

   Employee *ptr;  // base-class pointer

   Boss b("John", "Smith", 800.00);
   ptr = &b;  // base-class pointer to derived-class object
                                          // dynamic binding
   ptr->print();                          // dynamic binding
   cout << " earned $" << ptr->earnings(); // dynamic binding
   b.print();                             // static binding
   cout << " earned $" << b.earnings();    // static binding

   CommissionWorker c("Sue", "Jones", 200.0, 3.0, 150);
   ptr = &c;  // base-class pointer to derived-class object
                                          // dynamic binding
   ptr->print();                          // dynamic binding
   cout << " earned $" << ptr->earnings(); // static binding
   c.print();                             // static binding
   cout << " earned $" << c.earnings();    // static binding

   PieceWorker p("Bob", "Lewis", 2.5, 200);
   ptr = &p;  // base-class pointer to derived-class object
                                          // dynamic binding
   ptr->print();                          // dynamic binding
   cout << " earned $" << ptr->earnings(); // static binding
   p.print();                             // static binding
   cout << " earned $" << p.earnings();    // static binding

   HourlyWorker h("Karen", "Price", 13.75, 40);
   ptr = &h;  // base-class pointer to derived-class object
                                          // dynamic binding
   ptr->print();                          // dynamic binding
   cout << " earned $" << ptr->earnings(); // static binding
   h.print();                             // static binding
   cout << " earned $" << h.earnings();    // static binding

   cout << endl;

   return 0;
}
```

Fig. 20.1 Jerarquía de derivación de clase "empleado" que utiliza una clase base abstracta (parte 11 de 12).

```
          Boss: John Smith earned $800.00
          Boss: John Smith earned $800.00
Commission worker: Sue Jones earned $650.00
Commission worker: Sue Jones earned $650.00
     Piece worker: Bob Lewis earned $500.00
     Piece worker: Bob Lewis earned $500.00
    Hourly worker: Karen Price earned $550.00
    Hourly worker: Karen Price earned $550.00
```

Fig. 20.1 Jerarquía de derivación de clase "empleado" que utiliza una clase base abstracta (parte 12 de 12).

La ligadura dinámica permite que fabricantes independientes de software (*ISV*), distribuyan software sin tener que revelar secretos propietarios. Estas distribuciones de software pueden estar formadas de solo los archivos de cabecera y archivos objeto. No es necesario revelar el código fuente. Los desarrolladores de software entonces pueden utilizar la herencia, para derivar nuevas clases a partir de aquellas proporcionadas por los ISV. El software, que funciona con las clases que proporcionan los ISV, continuará funcionando con las clases derivadas y utilizará (vía ligaduras dinámicas) las funciones virtuales redefinidas proporcionadas en dichas clases.

La ligadura dinámica requiere que en tiempo de ejecución, se encamine la llamada a una función miembro virtual a la versión de función virtual apropiada para la clase. Se pone en práctica una *tabla de funciones virtuales* llamada la *vtable*, en forma de un arreglo que contiene apuntadores de función. Cada clase que contenga funciones virtuales tiene una *vtable*. Para cada función virtual dentro de la clase, la *vtable* tiene una entrada que contiene un apuntador de función a la versión de la función virtual para uso de un objeto de dicha clase. La función virtual a usar para una clase particular podría ser la función definida en dicha clase, o podría ser una función heredada, ya sea directa o indirecta, de una clase base más alta dentro de la jerarquía.

Cuando una clase base proporciona una función miembro y la declara **`virtual`**, las clases derivadas pueden redefinir la función virtual, pero no están obligadas a redefinirla. Por lo tanto, una clase derivada puede utilizar la versión de clase base de una función miembro virtual, y esto quedaría indicado en la *vtable*.

Cada objeto de una clase con funciones virtuales contiene un apuntador a la *vtable* para dicha clase. Este apuntador no es accesible para el programador. Se obtiene el apuntador apropiado de función en la *vtable* y se desreferencia para completar la llamada en tiempo de ejecución.

Esta búsqueda y desreferenciación de apuntador en *vtable* requiere de una sobrecarga nominal en tiempo de ejecución.

Sugerencia de rendimiento 20.1

El polimorfismo, tal y como se pone en práctica mediante funciones virtuales y ligaduras dinámicas, resulta eficiente. Los programadores pueden utilizar estas capacidades con un impacto solo nominal sobre el rendimiento del sistema.

Sugerencia de rendimiento 20.2

Las funciones virtuales y las ligaduras dinámicas permiten la programación polimórfica, en contraste con la programación lógica **switch**. *Los compiladores optimizantes de C++ por lo regular generan código que se ejecuta con una eficiencia por lo menos igual a la de la lógica basada en* **switch** *codificada manualmente.*

20.8 Destructores virtuales

Puede ocurrir un problema, al utilizar el polimorfismo para procesar objetos dinámicamente asignados de una jerarquía de clase. Si es destruido, aplicando el operador **`delete`** a un apuntador de clase base al objeto, la función destructor de clase base será llamada sobre el objeto. Esto ocurrirá independiente del tipo del objeto al cual está señalado el apuntador de clase base e independiente del hecho que el destructor de cada clase tenga un nombre distinto.

Existe una solución simple para este problema —declarar virtual el destructor de clase base. Esto hará virtuales automáticamente a todos los destructores de clase derivada, a pesar de que no tengan el mismo nombre que el destructor de clase base. Ahora, si un objeto en la jerarquía es destruido en forma explícita, mediante la aplicación del operador **`delete`** a un apuntador de clase base a un objeto de clase derivada, se llamará al destructor de la clase apropiada.

Práctica sana de programación 20.2

Si una clase tiene funciones virtuales, incluya un destructor virtual, inclusive si para dicha clase no se requiere uno. Las clases que se deriven de esta clase pudieran contener destructores que deberán ser llamados en forma correcta.

Error común de programación 20.3

Declarar un constructor como función virtual. Los constructores no pueden ser virtuales.

20.9 Estudio de caso: cómo heredar interfaz, y puesta en práctica

Nuestro siguiente ejemplo (figura 20.2) vuelve a examinar la jerarquía de punto, círculo y cilindro del capítulo anterior, excepto que ahora encabezamos la jerarquía con la clase base abstracta **`Shape`**. **`Shape`** tiene una función virtual pura —**`printShapeName`**— de tal forma que **`Shape`** es una clase base abstracta. **`Shape`** contiene otras dos funciones virtuales, es decir **`area`** y **`volume`**, cada una de las cuales tiene una puesta en práctica que regresa un valor cero. **`Point`** hereda estas puestas en práctica de **`Shape`**. Esto tiene sentido, porque tanto el área como el volumen de un punto son cero. **`Circle`** hereda la función **`volume`** de **`Point`**, pero **`Circle`** proporciona su propia puesta en práctica correspondiente a la función **`area`**. **`Cylinder`** proporciona sus propias puestas en práctica, tanto para la función **`area`** como para la función **`volume`**.

Note que aunque **`Shape`** es una clase base abstracta, aún así contiene puestas en práctica de dichas funciones miembro, y dichas puestas en práctica son heredables. La clase **`Shape`** proporciona una interfaz heredable en la forma de tres funciones virtuales, que pueden ser contenidas por todos los miembros de la jerarquía. La clase **`Shape`** también proporciona algunas puestas en práctica, que podrán utilizar las clases derivadas en los primeros pocos niveles de la jerarquía.

Este estudio de caso hace énfasis en el hecho de que una clase puede heredar la interfaz y/o la puesta en práctica de una clase base. Las jerarquías diseñadas para la herencia de puestas en práctica tienden a tener su funcionalidad alta dentro de la jerarquía. Las jerarquías diseñadas para la herencia de interfaz tienden a tener su funcionalidad baja dentro de la jerarquía.

La clase base **`Shape`** (figura 20.2, parte 1) está formada de tres funciones virtuales públicas y no contiene ningun dato. La función **`printShapeName`** es virtual pura, por lo que es redefinida en cada una de las clases derivadas. Las funciones **`area`** y **`volume`** se definen para devolver **`0.0`**. Estas funciones se redefinen en las clases derivadas cuando es apropiado para dichas clases el tener un cálculo distinto para **`area`** y/o un cálculo distinto de **`volume`**.

```
// SHAPE.H
// Definition of abstract base class Shape
#ifndef SHAPE_H
#define SHAPE_H

class Shape {
public:
   virtual float area() const { return 0.0; }
   virtual float volume() const { return 0.0; }
   virtual void printShapeName() const = 0; // pure virtual
};

#endif
```

Fig. 20.2 Definición de la clase base abstracta **Shape** (parte 1 de 9).

La clase **Point** (figura 20.2 partes 2 y 3) es derivada de **Shape** con herencia pública. Un **Point** no tiene ni área ni volumen, por lo que las funciones miembro de clase base **area** y **volume** aquí no se redefinen —se heredan tal y como están definidas en **Shape**. La función **printShapeName** es una puesta en práctica de una función virtual, que en la clase base fue definida como una virtual pura. Otras funciones miembro incluyen una función *set*, a fin de asignar coordenadas nuevas **x** e **y** a un **Point** y función *get*, para regresar las coordenadas **x** e **y** de un **Point**

La clase **Circle** (figura 20.2 partes 4 y 5) es derivada de **Point** con herencia pública. Un **Circle** no tiene volumen, por lo que la función miembro de clase base **volume** no se redefine aquí —es heredada de **Shape** (y subsecuentemente en **Point**). Un **Circle** tiene área, por lo que en esta clase se redefine la función **area**. La función **printShapeName** es una puesta en

```
// POINT1.H
// Definition of class Point
#ifndef POINT1_H
#define POINT1_H
#include <iostream.h>
#include "shape.h"

class Point : public Shape {
   friend ostream &operator<<(ostream &, const Point &);
public:
   Point(float = 0, float = 0);  // default constructor
   void setPoint(float, float);
   float getX() const { return x; }
   float getY() const { return y; }
   virtual void printShapeName() const { cout << "Point: "; }
private:
   float x, y;    // x and y coordinates of Point
};

#endif
```

Fig. 20.2 Definición de clase **Point** (parte 2 de 9).


```
// POINT1.CPP
// Member function definitions for class Point
#include <iostream.h>
#include "point1.h"

Point::Point(float a, float b)
{
   x = a;
   y = b;
}

void Point::setPoint(float a, float b)
{
   x = a;
   y = b;
}

ostream &operator<<(ostream &output, const Point &p)
{
   output << '[' << p.x << ", " << p.y << ']';

   return output;   // enables concatenated calls
}
```

Fig. 20.2 Definiciones de función miembro para la clase **Point** (parte 3 de 9).

```
// CIRCLE1.H
// Definition of class Circle
#ifndef CIRCLE1_H
#define CIRCLE1_H
#include "point1.h"

class Circle : public Point {
   friend ostream &operator<<(ostream &, const Circle &);
public:
   // default constructor
   Circle(float r = 0.0, float x = 0.0, float y = 0.0);

   void setRadius(float);
   float getRadius() const;
   virtual float area() const;
   virtual void printShapeName() const { cout << "Circle: "; }
private:
   float radius;   // radius of Circle
};

#endif
```

Fig. 20.2 Definición de clase **Circle** (parte 4 de 9).

práctica de una función virtual, que en la clase base fue definida como una virtual pura. Si esta función no se redefine aquí, sería heredada la versión **Point** de la función.

```
// CIRCLE1.CPP
// Member function definitions for class Circle
#include <iostream.h>
#include <iomanip.h>
#include "circle1.h"

// Constructor for Circle call constructor for Point
Circle::Circle(float r, float a, float b)
   : Point(a, b)  // call base-class constructor
{ radius = r > 0 ? r : 0; }

// Set radius
void Circle::setRadius(float r) { radius = r > 0 ? r : 0; }

// Get radius
float Circle::getRadius() const { return radius; }

// Calculate area of a Circle
float Circle::area() const { return 3.14159 * radius * radius; }

// Output a circle in the form: Center=[x, y]; Radius=#.##
ostream &operator<<(ostream &output, const Circle &c)
{
   output << '[' << c.getX() << ", " << c.getY()
          << "]; Radius = " << setiosflags(ios::showpoint)
          << setprecision(2) << c.radius;

   return output;   // enables concatenated calls
}
```

Fig. 20.2 Definiciones de función miembro para la clase **Circle** (parte 5 de 9).

Otras funciones miembro incluyen una función *set,* para asignar un nuevo **radius** a un **Circle** y una función *get,* para devolver **radius** de un **Circle**.

La clase **Cylinder** (figura 20.2 partes 6 y 7) es derivada de **Circle** con herencia pública. Un **Cylinder** tiene área y volumen, por lo que las funciones **area** y **volume** ambas se redefinen en esta clase. La función **printShapeName** es una puesta en práctica de una función virtual, que en la clase base fue definida como virtual pura. Si aquí no se redefine esta función, sería heredada la versión **Circle** de la función. Otras funciones miembro incluyen una función *set,* para asignar un nuevo **radius** a un **Circle** y una función *get,* para devolver **radius** de un **Circle**.

El programa manejador (figura 20.2 partes 8 y 9) empieza produciendo el objeto **Point** de la clase **point**, el objeto **circle** de la clase **Circle** y el objeto **cylinder** de clase **Cylinder**. Se invoca la función **printShapeName** para cada uno de los objetos y cada objeto es extraído con su operador de inserción de flujo homónimo, a fin de ilustrar que los objetos han sido inicializados en forma correcta. A continuación, se declara el apuntador **ptr** del tipo **Shape ***. Se utiliza este apuntador para señalar a cada uno de los objetos de la clase derivada. Primero se asigna la dirección de **point** a **ptr** y se efectúan las siguientes llamadas

```
ptr->printShapeName()
ptr->area()
ptr->volume()
```



```
// CYLINDER.H
// Definition of class Cylinder
#ifndef CYLINDER_H
#define CYLINDER_H
#include "circle1.h"

class Cylinder : public Circle {
   friend ostream &operator<<(ostream &, const Cylinder &);
public:
   // default constructor
   Cylinder(float h = 0.0, float r = 0.0,
            float x = 0.0, float y = 0.0);

   void setHeight(float);
   virtual float area() const;
   virtual float volume() const;
   virtual void printShapeName() const { cout << "Cylinder: "; }
private:
   float height;   // height of Cylinder
};

#endif
```

Fig. 20.2 Definición de clase **Cylinder** (parte 6 de 9).

para invocar estas funciones para el objeto al cual apunta **ptr**. La salida ilustra que las funciones han sido de forma correcta invocadas para **point**— **"Point: "** es extraído y el área y el volumen son ambos **0.00**. A continuación, se asigna a **ptr** la dirección del objeto **circle** y se invocan las mismas funciones. La salida ilustra que las funciones han sido correctamente invocadas para el objeto **circle** se extrae— **"Circle: "** el área de **circle** se calcula, y el volumen es **0.00**. Por último, **ptr** se asigna a la dirección del objeto **cylinder** y se invocan las mismas funciones. La salida ilustra que las funciones han sido invocadas correctamente para el objeto **cylinder**, se extrae **cylinder**— **"Cylinder: "** se calcula el área de **cylinder** y se calcula el volumen de **cylinder**. Todas las llamadas de función **printShapeName**, **area** y **volume** se resuelven en tiempo de ejecución mediante ligaduras dinámicas.

Resumen

- Con las funciones virtuales y el polimorfismo, se hace posible diseñar y poner en práctica sistemas que son más extensibles. Se pueden escribir programas para procesar objetos de tipos que pudieran no existir cuando el programa está aún en desarrollo.
- Las funciones virtuales y la programación polimórfica pueden eliminar la necesidad de la lógica **switch**. El programador puede utilizar el mecanismo de la funciones virtuales para llevar a cabo de manera automática la lógica equivalente y, por lo tanto, evitando los tipos de errores típicamente asociados con la lógica **switch**. Un código cliente que tome decisiones sobre tipos de objetos y representaciones indicará un diseño de clase pobre.
- Se declara una función virtual haciendo preceder en la clase base el prototipo de función por la palabra reservada **virtual**.

```
// CYLINDR1.CPP
// Member function definitions for class Cylinder
#include <iostream.h>
#include <iomanip.h>
#include "cylindr1.h"

Cylinder::Cylinder(float h, float r, float x, float y)
   : Circle(r, x, y)  // call base-class constructor
{ height = h > 0 ? h : 0; }

void Cylinder::setHeight(float h)
{ height = h > 0 ? h : 0; }

float Cylinder::area() const
{
   // surface area of Cylinder
   return 2 * Circle::area() +
          2 * 3.14159 * Circle::getRadius() * height;
}

float Cylinder::volume() const
{
   float r = Circle::getRadius();
   return 3.14159 * r * r * height;
}

ostream &operator<<(ostream &output, const Cylinder& c)
{
   output << '[' << c.getX() << ", " << c.getY()
          << "]; Radius = " << setiosflags(ios::showpoint)
          << setprecision(2) << c.getRadius()
          << "; Height = " << c.height;

   return output;   // enables concatenated calls
}
```

Fig. 20.2 Definiciones de función miembro para la clase `Cylinder` (parte 7 de 9).

- Si así lo desean, las clases derivadas pueden proveer sus propias puestas en práctica de una función virtual de clase base, pero si no es así, entonces se utilizará la puesta en práctica de la clase base.
- Si se llama una función virtual haciendo referencia a un objeto específico por su nombre y utilizando el operador de selección miembro punto, dicha referencia se resuelve en tiempo de compilación (esto se conoce como *ligadura estática*) y la función virtual llamada es aquella definida (o heredada) por la clase de dicho objeto particular.
- Existen muchas situaciones, sin embargo, en las cuales resulta útil definir clases para las cuales el programador no tiene jamás intención de producir objeto alguno. Estas clases se conocen como clases abstractas. Dado que en situaciones de herencia éstas son utilizadas como clases base, nos referiremos por lo general a ellas como clases base abstractas. No se pueden producir objetos de una clase base abstracta.


```
// FIG20_2.CPP
// Driver for point, circle, cylinder hierarchy
#include <iostream.h>
#include <iomanip.h>
#include "shape.h"
#include "point1.h"
#include "circle1.h"
#include "cylindr1.h"

main()
{
   // create some shape objects
   Point point(7, 11);
   Circle circle(3.5, 22, 8);
   Cylinder cylinder(10, 3.3, 10, 10);

   point.printShapeName();      // static binding
   cout << point << endl;

   circle.printShapeName();     // static binding
   cout << circle << endl;

   cylinder.printShapeName();   // static binding
   cout << cylinder << "\n\n";

   cout << setiosflags(ios::showpoint) << setprecision(2);
   Shape *ptr;                  // create base-class pointer

   // aim base-class pointer at derived-class Point object
   ptr = &point;
   ptr->printShapeName();       // dynamic binding
   cout << "x = " << point.getX() << "; y = " << point.getY()
        << "\nArea = " << ptr->area()
        << "\nVolume = " << ptr->volume() << "\n\n";

   // aim base-class pointer at derived-class Circle object
   ptr = &circle;
   ptr->printShapeName();       // dynamic binding
   cout << "x = " << circle.getX() << "; y = " << circle.getY()
        << "\nArea = " << ptr->area()
        << "\nVolume = " << ptr->volume() << "\n\n";

   // aim base-class pointer at derived-class Cylinder object
   ptr = &cylinder;
   ptr->printShapeName();       // dynamic binding
   cout << "x = " << cylinder.getX()
        << "; y = " << cylinder.getY()
        << "\nArea = " << ptr->area()
        << "\nVolume = " << ptr->volume() << endl;

   return 0;
}
```

Fig. 20.2 Manejador para la jerarquía punto, círculo y cilindro (parte 8 de 9).

```
Point: [7, 11]
Cicle: [22, 8]; Radius = 3.50
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00

Point: x = 7.00; y = 11.00
Area = 0.00
Volume = 0.00

Cicle: x = 22.00; y = 8.00
Area = 38.48
Volume = 0.00

Cylinder: x = 10.00; y = 10.00
Area = 275.77
Volume = 342.12
```

Fig. 20.2 Manejador para la jerarquía punto, círculo y cilindro (parte 9 de 9).

- Las clases a partir de las cuales se pueden producir objetos se conocen como clases concretas.
- Una clase con funciones virtuales se puede convertir en abstracta al declarar como pura una o más de sus funciones virtuales. Una función virtual pura es una que tenga un inicializador de = 0 dentro de su declaración.
- Si una clase es derivada de una clase con una función virtual pura sin proporcionar en la clase derivada una definición para dicha función virtual pura, entonces dicha función virtual se conserva como pura en la clase derivada. Y en consecuencia, la clase derivada también es una clase abstracta.
- C++ permite el polimorfismo —la habilidad de los objetos de clases distintas, emparentados mediante la herencia, de responder en forma diferente a la misma llamada de función miembro.
- El polimorfismo se pone en práctica vía funciones virtuales.
- Cuando se hace una solicitud mediante un apuntador de clase base para el uso de una función virtual, C++ escoge la función redefinida correcta, en la clase derivada apropiada asociada con dicho objeto.
- Mediante el uso de las funciones virtuales y del polimorfismo, una llamada de función miembro puede causar distintas acciones, dependiendo del tipo de objeto que reciba la llamada.
- Aunque no podemos producir objetos de clases base abstractas, *podemos* declarar apuntadores a clases base abstractas. Estos apuntadores pueden entonces utilizarse para permitir manipulaciones polimórficas de objetos de clase derivada, cuando dichos objetos sean producidos a partir de clases concretas.
- Continuamente se están añadiendo nuevos tipos de clase a los sistemas. Se les da sitio a la nuevas clases mediante la ligadura dinámica (también llamada ligadura tardía). No es necesario conocer en tiempo de compilación el tipo de un objeto, para que una llamada de función virtual sea compilada. La llamada de función virtual se hace coincidir en tiempo de ejecución con la función miembro del objeto llamado.

- La ligadura dinámica permite que fabricantes independientes de software (ISV) distribuyan software sin tener que revelar secretos propietarios. Las distribuciones de software pueden estar sólo constituidas de archivos de cabecera y archivos objeto. No es necesario revelar código fuente. A partir de las clases proveídas por los ISV, los desarrolladores de software podrán entonces utilizar la herencia para derivar nuevas clases. Aquel software que funcione con las clases proveídas por los ISV, continuará funcionando con las clases derivadas y utilizará (vía la ligadura dinámica) las funciones virtuales redefinidas proporcionadas en esas clases.
- La ligadura dinámica requiere que, en tiempo de ejecución, se encamine la llamada a una función miembro virtual, a la versión de función virtual apropiada para dicha clase. Se pone en operación una tabla de función virtual, conocida como la vtable, como un arreglo que contiene apuntadores de función. Cada clase, que contenga funciones virtuales, tiene una vtable. Para cada función virtual dentro de la clase, la vtable tiene una entrada que incluye un apuntador de función a la versión de la función virtual a utilizar para un objeto de dicha clase. La función virtual a utilizar para una clase particular pudiera ser la función definida en dicha clase, o pudiera ser una función heredada, ya sea directa o indirecta de una clase base más alta, dentro de la jerarquía.
- Cuando una clase base proporciona una función miembro y la declara **`virtual`**, las clases derivadas pueden redefinir la función virtual, pero no están obligadas a hacerlo. Por lo tanto, una clase derivada puede utilizar la versión de clase base de un función miembro virtual y esto quedaría indicado en la vtable.
- Cada objeto de una clase con funciones virtuales contiene un apuntador a la vtable para dicha clase. Este apuntador no es accesible para el programador. El apuntador de función apropiado para la vtable es obtenido y desreferenciado para completar la llamada en tiempo de ejecución. Esta búsqueda y desreferenciación de apuntadores en la vtable requiere de una sobrecarga nominal en tiempo de ejecución, usualmente menor que la del mejor código cliente posible.
- Declare el destructor de clase base virtual, si la clase contiene funciones virtuales. Esto hará virtuales a todos los destructores de clase derivada, aunque no tengan el mismo nombre que el destructor de clase base. Si un objeto en la jerarquía es explícitamente destruido mediante la aplicación del operador **`delete`** a un apuntador de clase base a un objeto de clase derivada, el destructor de la clase apropiada será llamado.

Terminología

clase base abstracta
clase abstracta
función de clase base **`virtual`**
jerarquía de clase
convertir objeto de clase derivada a objeto de clase base
convertir apuntador de clase derivada a apuntadoy de clase base
clase derivada
constructor de clase derivada
clase base directa
ligadura dinámica
ligadura temprana
eliminación de enunciados **`switch`**
conversión explícita de apuntadores
extensibilidad
conversión implícita de apuntador
clase base indirecta
herencia
ligadura tardía
programación orientada a objetos (OOP)
apuntador a una clase base
apuntador a una clase derivada
apuntador a una clase abstracta

polimorfismo
función virtual pura (`=0`)
función virtual redefinida
referencia a una clase base
referencia a una clase derivada
referencia a una clase abstracta
reutilización del software
ligadura estática
lógica `switch`
destructor virtual
función virtual
palabra reservada `virtual`

Errores comunes de programación

20.1 Resultará un error de sintaxis redefinir una clase derivada como función virtual de clase base, sin asegurarse que la función derivada tenga el mismo tipo de regreso y signatura que la versión de clase base.

20.2 Es un error de sintaxis intentar producir un objeto a partir de una clase abstracta (es decir, de una clase que contenga una o más funciones virtuales puras).

20.3 Declarar un constructor como función virtual. Los constructores no pueden ser virtuales.

Prácticas sanas de programación

20.1 A pesar que ciertas funciones en forma implícita son virtuales, en razón de una declaración hecha anteriormente en la jerarquía de clase, algunos programadores prefieren declarar estas funciones virtuales en forma explícita en cada uno de los niveles de la jerarquía, a fin de promover claridad en el programa.

20.2 Si una clase tiene funciones virtuales, incluya un destructor virtual, inclusive si para dicha clase no se requiere uno. Las clases que se deriven de esta clase pudieran contener destructores que deberán ser llamados en forma correcta.

Sugerencias de rendimiento

20.1 El polimorfismo, tal y como se pone en práctica mediante funciones virtuales y ligaduras dinámicas, resulta eficiente. Los programadores pueden utilizar estas capacidades con un impacto sólo nominal sobre el rendimiento del sistema.

20.2 Las funciones virtuales y las ligaduras dinámicas permiten la programación polimórfica, en contraste con la programación lógica `switch`. Los compiladores optimizantes de C++ por lo regular generan código que se ejecuta con una eficiencia por lo menos igual a la de la lógica basada en `switch` codificada manualmente.

Observaciones de ingeniería de software

20.1 Una consecuencia interesante del uso de funciones virtuales y polimorfismo es que los programas adquieren una apariencia simplificada. Contienen menos lógica de bifurcación, prefiriendo un código secuencial más sencillo. Esta simplificación facilita probar, depurar y mantener el programa.

20.2 Una vez declarada una función como virtual, se conserva como virtual a lo largo de toda la jerarquía de herencia, a partir de dicho punto.

20.3 Cuando una clase derivada decide no definir una función virtual, la clase derivada heredará la función virtual de la clase base inmediata.

20.4 Las funciones virtuales redefinidas deben tener el mismo tipo de regreso y la misma signatura que la función virtual base.

20.5 Si una clase se deriva de una clase con una función virtual pura, y para dicha función virtual pura no se ha dado definición en la clase derivada, entonces la función virtual también se conserva pura en la clase derivada. Por consecuencia, también la clase derivada será una clase abstracta.

20.6 Mediante las funciones virtuales y el polimorfismo, el programador puede ocuparse de generalidades y dejar que en tiempo de ejecución, el entorno se preocupe de lo específico. El programador puede dirigir una amplia variedad de objetos haciendo que se comporten de formas apropiadas a dichos objetos incluso sin conocer los tipos de los mismos.

20.7 El polimorfismo fomenta la extensibilidad: software escrito para invocar comportamiento polimórfico se escribe en forma independiente del tipo de los objetos a los cuales los mensajes son enviados. Por lo tanto, nuevos tipos de objetos, que pudieran responder a mensajes existentes, pueden ser añadidos en dicho sistema sin modificar el sistema base. A excepción de la parte de código cliente que produce nuevos objetos, los programas no necesitan ser recompilados.

20.8 Una clase abstracta define una interfaz para los distintos miembros de una jerarquía de clase. La clase abstracta contiene funciones virtuales puras, que serán definidas en las clases derivadas. Mediante el polimorfismo todas las funciones en la jerarquía pueden utilizar esta misma interfaz.

Ejercicios de autoevaluación

20.1 Llene cada uno de los siguientes espacios vacíos:

a) Utilizando la herencia y el polimorfismo se ayuda a eliminar la lógica _____.
b) Una función virtual pura se define colocando una _____ al fin de su prototipo en la definición de clase.
c) Si una clase contiene una o más funciones virtuales puras, es una _____.
d) Una llamada de función resuelta en tiempo de compilación se conoce como ligadura _____.
e) Una llamada de función resuelta en tiempo de ejecución se conoce como ligadura _____

Respuestas a los ejercicios de autoevaluación

20.1 a) `switch`. b) `= 0`. c) clase base abstracta. d) estática. e) dinámica.

Ejercicios

20.2 ¿Qué son las funciones virtuales? Describa alguna circunstancia en la cual las funciones virtuales serían apropiadas.

20.3 Dado que los constructores no pueden ser virtuales, describa algún procedimiento mediante el cual usted podría conseguir un efecto similar.

20.4 En el ejercicio 19.5 usted desarrolló una jerarquía de clase `Shape` y definió las clases dentro de la jerarquía. Modifique dicha jerarquía, de tal forma que la clase `Shape` sea una clase base abstracta que contenga la interfaz a la jerarquía. Derive `TwoDimensionalShape` y `ThreeDimensionalShape` de la clase `Shape` —estas clases también deberán ser abstractas. Utilice una función virtual `print` para extraer el tipo y dimensiones de cada clase. También incluya las funciones virtuales `area` y `volume` de tal forma que estos cálculos puedan ser ejecutados para objetos de cada clase concreta dentro de la jerarquía. Escriba un programa manejador que pruebe la jerarquía de clase `Shape`.

20.5 Utilizando la clase de plantilla `List` que se desarrolló en el ejercicio 17.10, cree una lista enlazada de apuntadores a objetos `Shape` (es decir, apuntadores a cualquier forma dentro de la jerarquía). Proporcione un operador de inserción de flujo homónimo para la clase `Shape`, que sólo llame la función virtual pura `print`. Utilice la capacidad de impresión de la lista enlazada para recorrer la lista y extraer cada objeto.

21

Flujo de entrada/salida de C++

Objetivos

- Comprender cómo utilizar el flujo de entrada/salida orientada a objetos de C++.
- Ser capaz de darle formato a las entradas y a las salidas.
- Comprender la jerarquía de clase de flujo de entradas/salidas.
- Comprender cómo introducir/extraer objetos de tipos definidos por usuario.
- Ser capaz de crear manipuladores de flujo definidos por usuario.
- Ser capaz de determinar el éxito o el fracaso de operaciones de entrada/salida.
- Ser capaz de ligar flujos de salida con flujos de entrada.

Al parecer, el estar consciente... no está dividido en pedazos
...Las metáforas mediante las cuales quedaría descrito
con más naturalidad son "río" o "flujo".
William James

Todas las noticias que merecen imprimirse.
Adolph S. Ochs

Sinopsis

21.1 Introducción

Las bibliotecas estándar de C++ proporcionan un conjunto extenso de capacidades de entrada/salida. Este capítulo analiza un conjunto de capacidades suficientes para llevar a cabo las operaciones de entradas y salidas más comunes y da un resumen general de las capacidades restantes.

Muchas de las características de entrada/salida descritas aquí son orientadas a objetos. El lector deberá encontrar de interés ver cómo dichas capacidades se ponen en práctica. Este estilo de entradas/salidas también hace mucho uso de características C++ como referencias, homonimia de funciones y homonimia de operadores.

Como veremos, C++ utiliza *entradas/salidas de tipo seguro*. Cada operación de entrada/salida se ejecuta de una manera que es sensible al tipo de los datos. Si una función de entrada/salida ha sido definida correctamente para manejar un tipo particular de datos, entonces dicha función es llamada en forma automática para manejar dicho tipo de datos. Si no existe coincidencia entre el tipo de los datos reales y una función para el manejo de dichos tipos de datos, se establece una indicación de error de compilación. Por lo tanto, no podrán introducirse datos equivocados dentro del sistema (como lo pueden hacer en C —una falla de C, que permite errores bastante sutiles y a menudo, extraños).

Los usuarios pueden especificar entradas/salidas de tipos definidos por usuario, así como de tipos estándar. Esta *extensibilidad* es una de las características más valiosas de C++.

Práctica sana de programación 21.1

En programas C++, utilice exclusivamente la forma de entradas/salidas de C++, a pesar del hecho que para los programadores C++ esté disponible el estilo de entradas/salidas de C.

Observación de ingeniería de software 21.1

El estilo de entradas/salidas de C++ es de tipo seguro.

Observación de ingeniería de software 21.2

C++ permite un tratamiento común de entradas/salidas de tipos predefinidos y de tipos definidos por usuario. Este tipo de estado común facilita el desarrollo de software en general y de la reutilización de software en particular.

21.2 Flujos

La entrada/salida en C++ ocurre en *flujos* de bytes. Un flujo es solo una secuencia de bytes. En operaciones de entrada, los bytes fluyen de un dispositivo (es decir, un teclado, una unidad de disco o una conexión de red) a la memoria principal. En operaciones de salida, los bytes fluyen de la memoria principal a un dispositivo (es decir, una pantalla de exhibición, una impresora, una unidad de disco o una conexión de red).

La aplicación asocia el significado con bytes. Los bytes pueden representar caracteres ASCII, datos en bruto de formato interno, imágenes gráficas, voz digitalizada, video digitalizado o cualquier otro tipo de información que pudiera una aplicación requerir.

El trabajo de los mecanismos de entrada y salida del sistema es mover los bytes de los dispositivos a la memoria y viceversa de forma consistente y confiable. Dichas transferencias a menudo, involucran movimientos mecánicos, como son el giro de un disco o de una cinta, o el tecleo en un teclado. El tiempo que, por lo general, toman esas transferencias es enorme en comparación con el tiempo que toma el procesador para la manipulación interna de los datos. Por lo tanto, para asegurar un rendimiento máximo, las operaciones de entrada/salida requieren una

cuidadosa planeación y sintonización. Temas como éste se discuten en detalle en los libros de texto de los sistemas operativos (De90).

C++ proporciona tanto capacidades de entradas y salidas de "bajo nivel" como de "alto nivel". Las capacidades de entrada/salida de bajo nivel (es decir, entradas/salidas sin formato) especifican en forma típica que cierto número de bytes sólo deben ser transferidos del dispositivo a la memoria o de la memoria al dispositivo. En este tipo de transferencias, el byte individual es el elemento de interés. Estas capacidades de bajo nivel, de hecho proporcionan transferencias de alta velocidad y volumen, pero estas capacidades no son particularmente convenientes para las personas.

Las personas prefieren una visión de mayor nivel de las entradas/salidas, es decir, *entradas/salidas con formato*, en el cual los bytes están agrupados en unidades significativas como enteros, números de punto flotante, caracteres, cadenas y tipos definidos por usuario. Estas capacidades, orientadas al tipo, son satisfactorias para la mayor parte de las entradas/salidas que no correspondan al proceso de archivos de alto volumen.

Sugerencia de rendimiento 21.1

Utilice entradas/salidas sin formato para un rendimiento máximo en procesos de alto volumen de archivos.

21.2.1 Archivos de cabecera de biblioteca iostream

La biblioteca **iostream** de C++ proporciona cientos de capacidades de entrada/salida. Varios archivos de cabecera contienen porciones de la interfaz de la biblioteca.

La mayor parte de los programas de C++ deben incluir el archivo de cabecera **iostream.h**, que contiene información básica requerida para todas las operaciones de flujo de entrada/salida. El archivo de cabecera **iostream.h** contiene los objetos **cin**, **cout**, **cerr** y **clog** los cuales, como veremos, corresponden al flujo de entrada estándar, flujo de salida estándar y flujo de error estándar. Se proporcionan tanto capacidades de entrada y salida con, como sin formato.

El encabezado **iomanip.h** contiene información útil para llevar a cabo entradas/salidas con formato, mediante *manipuladores de flujo parametrizados*.

El encabezado **fstream.h** contiene información de importancia para llevar a cabo las operaciones de proceso de archivos controlados por el usuario.

El encabezado **strstream.h** contiene información de importancia para llevar a cabo *formato en memoria* (también conocido como *formato en núcleo*). Esto se parece al proceso de archivos, pero en las operaciones "entrada y salida" se ejecutan hacia y desde arreglos de caracteres, en vez de hacia y desde archivos.

El encabezado **stdiostream.h** contiene información de importancia para aquellos programas que combinen los estilos de C y de C++ para entradas/salidas. Los programas nuevos deberían evitar entradas/salidas de estilo C, pero aquellas personas que deben modificar programas en C existentes, o que transforman programas de C a C++, pueden encontrar útiles estas capacidades.

Cada puesta en práctica de C++ por lo general, contiene otras bibliotecas relacionadas con entradas/salidas que proporcionan capacidades específicas a cada sistema, como el control de dispositivos de uso especial para entradas/salidas de audio y de video.

21.2.2 Clases y objetos de flujo de entrada/salida

La biblioteca **iostream** contiene muchas clases para el manejo una gran variedad de operaciones de entradas/salidas. La clase **istream** apoya operaciones de entrada de flujo. La clase **ostream** apoya operaciones de salida de flujo. La clase **iostream** apoya tanto operaciones de entrada como de salida de flujo.

Las clases **istream** y **ostream** cada una de ellas han sido derivadas mediante herencia simple de la clase base **ios**. La clase **iostream** es una derivación mediante herencia múltiple, tanto de la clase **istream** como de la clase **ostream**. Estas relaciones de herencia se resumen en la figura 21.1.

La homonimia de operadores permite una forma de notación conveniente para llevar a cabo entradas/salidas. Se hace la homonimia del operador de desplazamiento a la izquierda (<<) para designar salida de flujo y se conoce como *operador de inserción de flujo*. Se hace la homonimia del operador de desplazamiento a la derecha (>>) a fin de designar entrada de flujo y se conoce como *operador de extracción de flujo*. Estos operadores son utilizadas con los objetos de flujo estándar **cin**, **cout**, **cerr** y **clog**, y con objetos de flujo definidos por usuario.

cin es un objeto de la clase **istream** y se dice que está "ligado a" (o conectado con) el dispositivo de entrada estándar, que por lo regular es el teclado. El operador de extracción de flujo, tal y como se usa en el siguiente enunciado, hace que se introduzca un valor para la variable entera **grade** desde **cin** a la memoria

```
cin >> grade;
```

Note que la operación de extracción de flujo es "lo suficiente inteligente" para "saber" cuál es el tipo de los datos. Suponiendo que **grade** ha sido declarado en forma correcta, no es necesario especificar ningún tipo adicional de información para utilizar el operador de extracción de flujo (como sería el caso, incidentalmente, en entradas/salidas en estilo C).

cout es un objeto de la clase **ostream** y se dice que está "ligado con" el dispositivo de salida estándar, que por lo regular es la pantalla de exhibición. El operador de inserción de flujo, tal y como se usa en el enunciado siguiente, hace que se extraiga el valor de la variable entera **grade** de la memoria hacia el dispositivo de salida estándar.

```
cout << grade;
```

Note que el operador de inserción de flujo es "lo suficiente inteligente "para saber" el tipo de **grade** (suponiendo que éste haya sido declarado en forma correcta), por lo que no es necesario ningún tipo adicional de información para uso con el operador de inserción de flujo.

cerr es un objeto de la clase **ostream** y se dice que "está ligado con" el dispositivo de error estándar. Las salidas al objeto **cerr** no pasan por almacenamientos temporales o búfers. Esto significa que cada inserción a **cerr** hará que su salida aparezca de inmediato; esto resulta apropiado para una notificación instantánea de algún problema al usuario.

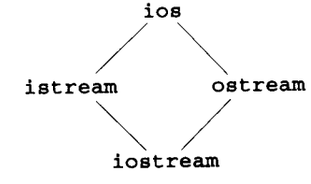


Fig. 21.1 Porción de la jerarquía de clase de flujo de entradas/salidas.

clog es un objeto de la clase **ostream** y también se dice que "está ligado con" el dispositivo de error estándar. Las salidas a **clog** sí pasan por almacenamiento temporal o búfer.

El procesamiento de archivo en C++ utiliza las clases **ifstream** para llevar a cabo operaciones de entrada de archivo, utiliza a **ofstream** para operaciones de salida de archivo, y **fstream** para operaciones de entrada/salida de archivo. La clase **ifstream** hereda de **istream**, la clase **ofstream** hereda de **ostream**, y la clase **fstream** hereda de **iostream**. Las relaciones de herencia de las clases relacionadas con entradas/salidas se resumen en la figura 21.2. En la jerarquía completa de clase de flujo de entradas/salidas en la mayor parte de las instalaciones existen muchas más clases incluidas, pero las clases que aquí se muestran proporcionan la gran mayoría de las capacidades que necesitarán la mayor parte de los programadores. Vea las referencias de biblioteca de clase correspondientes a su sistema C++ para más información de procesamiento de archivos.

21.3 Salida de flujo

La clase **ostream** de C++ da la capacidad para llevar a cabo salida con y sin formato. Las capacidades de salida incluyen: la salida de tipos de datos estándar mediante el operador de inserción de flujo; la salida de caracteres utilizando la función miembro **put**; salida sin formato mediante la función miembro **write** (sección 21.5); salida de enteros en formatos decimal, octal y hexadecimal (sección 21.6.1); salida de valores en punto flotante con distintas precisiones (sección 21.6.2), con puntos decimales obligados (21.7.2), en notación científica, y en notación fija (sección 21.7.6); salida de datos justificados en campos de anchos de campo designados (sección 21.7.3); salida de datos en campos rellenos de caracteres especiales (sección 21.7.4); y salida de letras mayúsculas en notación científica y hexadecimal (sección 21.7.7).

21.3.1 Operador de inserción de flujo

La salida de flujo puede ser ejecutada mediante el operador de inserción de flujo, es decir, el operador homónimo <<. Se hace la homonimia del operador << para extraer elementos de datos de tipos incorporados, para extraer cadenas y para extraer valores de apuntadores. En la sección 21.9, veremos como hacer la homonimia de << para extraer elementos de datos de tipos definidos por usuario. La figura 21.3 demuestra la salida de una cadena.

El ejemplo anterior utiliza un único enunciado de inserción de flujo a fin de mostrar la cadena de salida. Se pueden utilizar varios enunciados de inserción, como en la figura 21.4. Cuando este programa sea ejecutado, producirá la misma salida que el programa anterior.

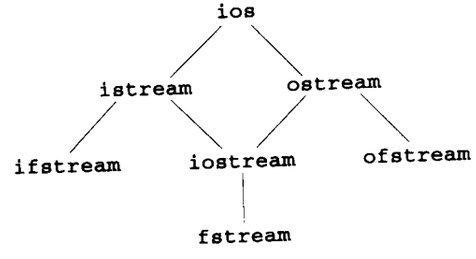


Fig. 21.2 Porción de la jerarquía de clase de flujo de entradas/salidas mostrando las clases de procesamiento de archivo clave.


```
// fig21_03.cpp
// Outputting a string using stream insertion.

#include <iostream.h>

main()
{
   cout << "Welcome to C++!\n";

   return 0;
}
```

```
Welcome to C++!
```

Fig. 21.3 Cómo extraer una cadena utilizando la inserción de flujo.

```
// fig21_04.cpp
// Outputting a string using two stream insertions.

#include <iostream.h>

main()
{
   cout << "Welcome to ";
   cout << "C++!\n";

   return 0;
}
```

```
Welcome to C++!
```

Fig. 21.4 Cómo extraer una cadena utilizando dos inserciones de flujo.

El efecto de la secuencia de escape **\n** (nueva línea) también se puede conseguir mediante el *manipulador de flujo* **endl** (terminar línea) como en la figura 21.5. El manipulador de flujo **endl** emite un carácter de nueva línea y, además, vacía el búfer de salida (es decir, hace que el búfer de salida sea extraído de inmediato, aunque todavía no esté lleno). El búfer de salida también puede ser vaciado, sólo mediante

```
cout << flush;
```

En la sección 21.6 se analizan en detalle los manipuladores de flujo.

Las expresiones pueden ser extraídas tal y como se muestra en la figura 21.6.

Práctica sana de programación 21.2

Al extraer expresiones, colóquelas entre paréntesis, para evitar problemas de precedencia de operadores entre los operadores de la expresión y el operador <<.

```
// fig21_05.cpp
// Using the endl stream manipulator.
#include <iostream.h>

main()
{
   cout << "Welcome to ";
   cout << "C++!";
   cout << endl;    // end line stream manipulator

   return 0;
}
```

```
Welcome to C++!
```

Fig. 21.5 Cómo utilizar el manipulador de flujo **endl**.

```
// fig21_06.cpp
// Outputting expression values.
#include <iostream.h>

main()
{
   cout << "47 plus 53 is ";
   cout << 47 + 53;       // expression
   cout << endl;

   return 0;
}
```

```
47 plus 53 is 100
```

Fig. 21.6 Cómo extraer valores de expresiones.

21.3.2 Cómo concatenar operadores de inserción/extracción de flujo

Los operadores homónimos << y >> cada uno de ellos puede ser utilizado en forma concatenada, tal y como se muestra en la figura 21.7.

Las múltiples inserciones de flujo de la figura 21.7 se ejecutan como si hubieran sido escritas:

```
(((cout << "47 plus 53 is ") << 47 + 53) << endl);
```

Esto está permitido porque el operador homónimo << devuelve una referencia a su objeto operando izquierdo, es decir, **cout**. Por lo tanto, la expresión entre paréntesis más a la izquierda

```
(cout << "47 plus 53 is ")
```



```
// fig21_07.cpp
// Concatenating the overloaded << operator.
#include <iostream.h>

main()
{
   cout << "47 plus 53 is " << 47 + 53 << endl;

   return 0;
}
```

```
47 plus 53 is 100
```

Fig. 21.7 Cómo concatenar el operador homónimo <<.

extrae la cadena especificada de caracteres y devuelve una referencia a **cout**. Esto permite que se evalúe la expresión entre paréntesis intermedia como

```
(cout << 47 + 53)
```

que extrae el valor entero **100** y devuelve una referencia a **cout**. A continuación se evalúa la expresión entre paréntesis más a la derecha como

```
cout << endl
```

que extrae una nueva línea, vacía a **cout** y devuelve una referencia a **cout**. Este último regreso no es utilizado.

21.3.3 Salida de variables char*

En entradas/salidas de estilo C, es necesario proporcionar información de tipo. C++ determina los tipos de datos en forma automática —lo que resulta una agradable mejoría respecto a C. Pero algunas veces esto "resulta un estorbo". Por ejemplo, sabemos que una cadena de caracteres es del tipo **char***. Suponga que deseamos imprimir el valor de dicho apuntador, es decir, la dirección en memoria del primer carácter de dicha cadena. Pero se ha hecho la homonimia del operador << para imprimir datos del tipo **char*** como si fuera una cadena terminada por **null**. La solución es hacer una conversión explícita (cast) del apuntador a **void*** (esto debe ser hecho a cualquier variable de apuntador a la cual el programador desee extraer como una dirección). En la figura 21.8 se demuestra la impresión de una variable **char*** tanto en formato de cadena como de dirección. Note que la dirección se imprime como un número hexadecimal (de base 16). En C++ los números hexadecimales empiezan con **0x**, o **0X**. En las secciones 21.6.1, 21.7.4, 21.7.5 y 21.7.7 ampliamos la información respecto a cómo controlar las bases de los números.

21.3.4 Extracción de caracteres mediante la función miembro put; cómo concatenar put

La función miembro **put** extrae un carácter como en

```
cout.put('A');
```

```
// fig21_08.cpp
// Printing the address stored in a char* variable

#include <iostream.h>

main()
{
   char *string = "test";

   cout << "Value of string is: " << string
        << "\nValue of (void *)string is: "
        << (void *)string << endl;

   return 0;
}
```

```
Value of string is: test
Value of (void *)string is: 0x00aa
```

Fig. 21.8 Cómo imprimir la dirección almacenada en una variable `char *`.

que exhibe **A** en pantalla. Las llamadas a `put` pueden ser concatenadas como en

```
cout.put('A').put('\n');
```

que extrae **A**, seguida por un carácter de nueva línea. La función `put` también puede ser llamada con una expresión de valor ASCII, como en `cout.put(65)`, que también extrae a **A**.

21.4 Entrada de flujo

Consideremos ahora entradas de flujo. Esto se puede ejecutar con el operador de extracción de flujo, es decir, con el operador homónimo >>. Este operador por lo general, pasa por alto los *caracteres de espacio en blanco* (como espacios en blanco, tabuladores y nuevas líneas) existentes en el flujo de entrada. Más tarde veremos cómo modificar este comportamiento. El operador de extracción de flujo devuelve cero (falso) cuando dentro de un flujo encuentra el final de archivo. Cada flujo contiene un conjunto de *bits de estado* que se utilizan para controlar el estado del flujo (es decir, estados de formato, de establecimiento de errores, etcétera). La extracción de flujo hace que para entradas de tipo incorrecto se active `failbit` del flujo, y si la operación falla hace que se active `badbit` del flujo. Veremos pronto cómo probar estos bits después de una operación de entrada/salida. En las sección 21.7 y 21.8 se analizan los bits de estado de flujo en detalle.

21.4.1 Operador de extracción de flujo

Para leer dos enteros, utilice el objeto `cin` y el operador de extracción de flujo >> homónimo, tal y como aparece en la figura 21.9.

La relativamente alta precedencia de los operadores >> y << pueden causar problemas. Por ejemplo, el programa de la figura 21.10 no se compilará en forma correcta sin los paréntesis que encierran la expresión condicional. El lector deberá de verificar lo anterior.


```
// fig21_09.cpp
// Calculating the sum of two integers
// input from the keyboard with cin
// and the stream extraction operator.
#include <iostream.h>

main()
{
   int x, y;

   cout << "Enter two integers: ";
   cin >> x >> y;
   cout << "Sum of " << x << " and " << y << " is: "
        << x + y << endl;

   return 0;
}
```

```
Enter two integers: 30 92
Sum of 30 and 92 is: 122
```

Fig. 21.9 Cómo calcular la suma de dos enteros introducidos desde el teclado mediante `cin` y el operador de extracción de flujo.

```
// fig21_10.cpp
// Revealing a precedence problem.
// Need parentheses around the conditional expression.
#include <iostream.h>

main()
{
   int x, y;

   cout << "Enter two integers: ";
   cin >> x >> y;
   cout << x << (x == y ? " is" : " is not")
        << " equal to " << y << endl;

   return 0;
}
```

```
Enter two integers: 7 5
7 is not equal to 5
```

```
Enter two integers: 8 8
8 is equal to 8
```

Fig. 21.10 Cómo evitar un problema de precedencia entre el operador de inserción de flujo y el operador condicional.

Error común de programación 21.1

Intentar leer de un ostream (o de cualquier otro flujo de solo salida).

Error común de programación 21.2

Intentar escribir a un istream (o a cualquier otro flujo de sólo entrada).

Error común de programación 21.3

Omitir paréntesis para obligar a una correcta precedencia al utilizar los operadores de inserción de flujo << o de extracción de flujo >> que tienen una relativamente alta precedencia.

Una forma popular para introducir una serie de valores es utilizar la operación de extracción de flujo en el encabezado de un ciclo **while**. La extracción devuelve falso (cero) cuando encuentra el fin de archivo. Considere el programa de la figura 21.11, mismo que encuentra la calificación más alta de un examen. Suponga que el número de calificaciones no es conocido con anticipación, y que el usuario escribirá un fin de archivo, a fin de indicar que todas las calificaciones han sido introducidas. La condición **while (cin >> grade)**, se convierte en cero (es decir, falso) cuando el usuario introduzca el fin de archivo.

```
// fig21_11.cpp
// Stream extraction operator returning
// false on end-of-file.
#include <iostream.h>

main()
{
   int grade, highestGrade = -1;

   cout << "Enter grade (enter end-of-file to end): ";
   while (cin >> grade) {
      if (grade > highestGrade)
         highestGrade = grade;

      cout << "Enter grade (enter end-of-file to end): ";
   }

   cout << "\nHighest grade is: " << highestGrade
        << endl;
   return 0;
}
```

```
Enter grade (enter end-of-file to end): 67
Enter grade (enter end-of-file to end): 87
Enter grade (enter end-of-file to end): 73
Enter grade (enter end-of-file to end): 95
Enter grade (enter end-of-file to end): 34
Enter grade (enter end-of-file to end): 99
Enter grade (enter end-of-file to end): ^Z

Highest grade is: 99
```

Fig. 21.11 El operador de extracción de flujo devolviendo falso al fin de archivo.

21.4.2 Funciones miembro get y getline

La función miembro **get** sin ningún argumento, introduce un carácter del flujo designado (incluso si se trata de un espacio en blanco) y devuelve dicho carácter como el valor de la llamada de función. Esta versión de **get** devuelve **EOF** cuando en el flujo se encuentra el fin de archivo. **EOF** por lo regular es **-1**, a fin de distinguirlo de valores de caracteres ASCII (esto de un sistema a otro pudiera variar).

En la figura 21.12 se demuestra el uso de las funciones miembro **eof** y **get** en el flujo de entrada **cin** y de la función miembro **put** en el flujo de salida **cout**. El programa primero imprime el valor de **cin.eof()**, es decir, **0**, (falso) a fin de mostrar que en **cin** no ha ocurrido el fin de archivo. El usuario escribe una línea de texto, seguida por un fin de archivo (**<ctrl>-z** seguido por **<return>**, en sistemas IBM PC y compatibles, y **<ctrl>-d**, en sistemas UNIX y Macintosh). El programa leerá cada uno de los caracteres y mediante la función miembro **put** lo extraerá a **cout**. Cuando se encuentra con el fin de archivo, se termina el **while**, y **cin.eof()** —ahora **1** (verdadero)— se vuelve a imprimir, a fin de demostrar que en **cin** se ha definido el fin de archivo. Note que este programa utiliza la versión de la función miembro **get** de **istream**, que no toma argumentos y que devuelve el carácter que se está introduciendo.

```
// fig21_12.cpp
// Using member functions get, put, and eof.

#include <iostream.h>

main()
{
   int c;

   cout << "Before input, cin.eof() is "
        << cin.eof() << '\n'
        << "Enter a sentence followed by end-of-file:\n";

   while ((c = cin.get()) != EOF)
      cout.put(c);

   cout << "\nAfter input, cin.eof() is "
        << cin.eof() << endl;

   return 0;
}
```

```
Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions^Z
Testing the get and put member functions
After input cin.eof() is 1
```

Fig. 21.12 Cómo utilizar las funciones miembro **get**, **put** y **eof**.

La función miembro **get** con un argumento de carácter, introduce el carácter siguiente del flujo de entrada (aun si se trata de un carácter en blanco). Esta versión de **get** devuelve falso cuando se encuentra el fin de archivo; de lo contrario esta versión de **get** devuelve una referencia al objeto **istream** para la cual se ha invocado a la función miembro **get**.

Una tercera versión de la función miembro **get** toma tres argumentos —un arreglo de caracteres, un límite de tamaño y un delimitador (con un valor por omisión de **'\n'**). Esta versión lee caracteres a partir del flujo de entrada. Lee hasta uno menos que el número máximo especificado de caracteres y termina, o termina en cuanto lea el delimitador. Se inserta un carácter nulo para terminar la cadena de caracteres, en el arreglo de caracteres que se utiliza como búfer por el programa. El delimitador no es colocado en el arreglo de caracteres, sino que es conservado en el flujo de entrada. En la figura 21.13 se compara la entrada utilizando **cin** con la extracción de flujo y la entrada con **cin.get**.

```
// fig21_13.cpp
// Contrasting input of a string with cin and cin.get.

#include <iostream.h>

const int SIZE = 80;

main()
{
   char buffer1[SIZE], buffer2[SIZE];

   cout << "Enter a sentence:\n";
   cin >> buffer1;
   cout << "\nThe string read with cin was:\n"
        << buffer1 << "\n\n";

   cin.get(buffer2, SIZE);
   cout << "The string read with cin.get was:\n"
        << buffer2 << endl;

   return 0;
}
```

```
Enter a sentence:
Contrasting string input with cin and cin.get

The string read with cin was:
Contrasting

The string read with cin.get was:
 string input with cin and cin.get
```

Fig. 21.13 Comparación de entradas de una cadena mediante **cin**, con la extracción de flujo y la entrada mediante **cin.get**.

La función miembro **getline** opera como la tercera versión de la función miembro **get** e inserta un carácter nulo después de la línea en el arreglo de caracteres. La función **getline** elimina el delimitador del flujo, pero no lo almacena en el arreglo de caracteres. El programa de la figura 21.14 demuestra el uso de la función miembro **getline** para la introducción de una línea de texto.

21.4.3 Otras funciones miembro istream (peek, putback, ignore)

La función miembro **ignore** pasa por alto un número designado de caracteres (un carácter por omisión) o da por terminado, al encontrar un delimitador designado (el delimitador por omisión es **EOF**).

La función miembro **putback** coloca el carácter previo obtenido por un **get** del flujo de entrada de regreso en dicho flujo. Esta función es útil para aquellas aplicaciones que rastrean un flujo de entrada buscando un campo que se inicie con un carácter específico. Cuando se introduce dicho carácter, la aplicación devuelve este carácter al flujo, a fin de que el carácter pueda ser incluido en los datos que serán introducidos.

La función miembro **peek** devuelve el carácter siguiente de un flujo de entrada, pero sin retirarlo del flujo.

```
// fig21_14.cpp
// Character input with member function getline.

#include <iostream.h>

const SIZE = 80;

main()
{
   char buffer[SIZE];

   cout << "Enter a sentence:\n";
   cin.getline(buffer, SIZE);

   cout << "\nThe sentence entered is:\n"
        << buffer << endl;

   return 0;
}
```

```
Enter a sentence:
Using the getline member function

The sentence entered is:
Using the getline member function
```

Fig. 21.14 Entrada de caracteres mediante la función miembro **getline**.

21.4.4 Entradas/salidas de tipo seguro

C++ ofrece *entradas/salidas de tipo seguro*. Se hace la homonimia de los operadores << y >> a fin de que acepten elementos de datos de tipos específicos. Si se procesan datos no esperados, se establecen varias banderas de error, mismas que el usuario puede probar, para determinar si una operación de entrada/salida ha tenido éxito o ha fracasado. De esta manera el programa "se conserva en control". Analizaremos estas banderas de error en la sección 21.8.

21.5 Entradas/salidas sin formato utilizando read, gcount, y write

Se lleva a cabo *entradas/salidas sin formato* mediante las funciones miembro **read** y **write**. Cada una de estas funciones introduce o extrae cierta cantidad de bytes de o desde un arreglo de caracteres existente en memoria. Estos bytes no contienen ningún tipo de formato. Sólo son introducidos o extraídos como bytes en bruto. Por ejemplo, la llamada

```
char buffer[] = "HAPPY BIRTHDAY";
cout.write(buffer, 10);
```

extrae los primeros 10 bytes de **buffer**. Dado que la cadena de caracteres se evalúa a la dirección del primero de los caracteres, la llamada

```
cout.write("ABCDEFGHIJKLMNOPQRSTUVWXYZ", 10);
```

exhibe los primeros 10 caracteres del alfabeto.

La función miembro **read** introduce un número designado de caracteres en un arreglo de caracteres. Si son leídos menos que el número designado de caracteres, se activa **failbit**. Pronto veremos cómo determinar si **failbit** ha sido activado (vea la sección 21.8).

En la figura 21.5 se demuestra el uso de las funciones miembro **read** y **gcount** de **istream**, y la función miembro **write** de **ostream**. El programa introduce 20 caracteres, (a partir de una secuencia de entrada más larga) en el arreglo de caracteres de nombre **buffer** mediante **read**, determina el número de caracteres introducidos mediante **gcount**, y extra los caracteres existentes en **buffer** mediante **write**

21.6 Manipuladores de flujo

C++ proporciona varios *manipuladores de flujo* que ejecutan tareas de formato. Los manipuladores de flujo ofrecen capacidades tales como definición de anchos de campo, definición de precisiones, establecimiento o eliminación de banderas de formato, establecimiento de caracteres de relleno en campos, vaciado de flujos, inserción de nueva línea en el flujo de salida y vaciado de flujo, inserción de un carácter nulo en el flujo de salida, y pasar por alto los espacios en blanco del flujo de entrada. Estas características se describen en las secciones siguientes.

21.6.1 Base de flujo integral: manipuladores de flujo dec, oct, hex, y setbase

Los enteros por lo general, se interpretan como valores decimales (de base 10). Para cambiar la base con la cual se interpretan los enteros dentro de un flujo, inserte el manipulador **hex** para definir la base a hexadecimal (de base 16), o el manipulador **oct** para definir a base octal (de base 8). Inserte el manipulador de flujo **dec** para devolver la base del flujo a decimal.


```
// fig21_15.cpp
// Unformatted I/O with the read,
// gcount and write member functions.

#include <iostream.h>

const int SIZE = 80;

main()
{
   char buffer[SIZE];

   cout << "Enter a sentence:\n";
   cin.read(buffer, 20);
   cout << "\nThe sentence entered was:\n";
   cout.write(buffer, cin.gcount());
   cout << endl;

   return 0;
}
```

```
Enter a sentence:
Using the read, write, and gcount member functions

The sentence entered was:
Using the read, writ
```

Fig. 21.15 Entradas/salidas sin formato mediante las funciones miembro **read**, **gcount** y **write**.

La base de un flujo también puede modificarse mediante el manipulador de flujo **setbase**, que toma un argumento entero **10**, **8** ó **16** para definir la base. Dado que **setbase** toma un argumento, se conoce como un *manipulador de flujo parametrizado*. El uso de **setbase** o de cualquier otro manipulador parametrizado requiere de la inclusión del archivo de cabecera **iomanip.h**. La base de flujo se conservará sin cambios hasta que sea modificada en forma explícita. En la figura 21.16 se muestra la utilización de los manipuladores de flujo **hex**, **oct**, **dec** y **setbase**.

21.6.2 Precisión de punto flotante (precision, setprecision)

Podemos controlar la *precisión* de los números en punto flotante, es decir, el número de dígitos a la derecha del punto decimal, ya sea mediante el manipulador de flujo **setprecision** o la función miembro **precision**. Una llamada a cualquiera de estas funciones define la precisión para todas las operaciones subsecuentes de salida, hasta la siguiente llamada definidora de precisión. La función miembro **precision** sin argumento, devuelve el ajuste actual de precisión. El programa de la figura 21.17 utiliza tanto la función miembro **precision**, como el manipulador **setprecision**, para imprimir una tabla mostrando la raíz cuadrada de **2** con precisiones variando desde **0** hasta **9**. Note que la precisión **0** tiene un significado especial. Restaura la *precisión por omisión* de valor **6**.

```
// fig21_16.cpp
// Using hex, oct, dec and setbase stream manipulators.
#include <iostream.h>
#include <iomanip.h>

main()
{
   int n;

   cout << "Enter a decimal number: ";
   cin >> n;

   cout << n << " in hexadecimal is: "
        << hex << n << '\n'
        << dec << n << " in octal is: "
        << oct << n << '\n'
        << setbase(10) << n << " in decimal is: "
        << n << endl;

   return 0;
}
```

```
Enter a decimal number: 20
20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20
```

Fig. 21.16 Cómo utilizar los manipuladores de flujo `hex`, `oct`, `dec`, y `setbase`.

21.6.3 Ancho de campo (Setw, Width)

La función miembro **`width`** define el ancho de campo y devuelve el ancho anterior. Si los valores procesados son menores que el ancho de campo, se insertan *caracteres de llenado* como *relleno*. Un valor más ancho que el ancho designando no será truncado —se imprimirá el número completo. El ajuste de ancho se aplica sólo a la siguiente inserción o extracción; después y de forma implícita el ancho se define a **0**, es decir, los valores de salida sencillamente serán tan anchos como lo requieran. La función **`width`** sin argumento devuelve el ajuste presente.

Error común de programación 21.4

Cuando no se de un campo lo suficiente amplio para manejar salidas, dichas salidas se imprimirán del ancho que requieran, posiblemente causando salidas erróneas o difíciles de leer.

En la figura 21.18 se demuestra el uso de la función miembro **`width`** tanto en la entrada como en la salida. Note que en el caso de la entrada, se leerá un máximo de un carácter menos que el ancho establecido, porque se deja lugar para el carácter nulo a colocarse en la cadena de entrada. Recuerde que la extracción de flujo se da por terminada cuando se encuentra un espacio en blanco a la derecha. El manipulador de flujo **`setw`** también puede ser utilizado para definir el ancho de campo.


```
// fig21_17.cpp
// Controlling precision of floating point values
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

main()
{
   double root2 = sqrt(2.0);

   cout << "Square root of 2 with precisions 0-9.\n"
        << "Precision set by the "
        << "precision member function:\n";

   for (int places = 0; places <= 9; places++) {
      cout.precision(places);
      cout << root2 << endl;
   }

   cout << "\nPrecision set by the "
        << "setprecision manipulator:\n";

   for (places = 0; places <= 9; places++)
      cout << setprecision(places) << root2 << endl;

   return 0;
}
```

```
Square root of 2 with precisions 0-9.
Precision set by the precision member function:
1.414214
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562

Precision set by the setprecision manipultor:
1.414214
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Fig. 21.17 Cómo controlar la precisión de valores de punto flotante.

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA
DEPARTAMENTO

```
// fig21_18.cpp
// Demonstrating the width member function

#include <iostream.h>

main()
{
   int w = 4;
   char string[10];

   cout << "Enter a sentence:\n";
   cin.width(5);

   while (cin >> string) {
      cout.width(w++);
      cout << string << '\n';
      cin.width(5);
   }

   return 0;
}
```

```
Enter a sentence:
This is a test of the width member function^Z
This
   is
    a
   test
      of
      the
      widt
         h
        memb
           er
          func
           tion
```

Fig. 21.18 Demostración de la función miembro **width**.

21.6.4 Manipuladores definidos por usuario

Los usuarios pueden crear su propios manipuladores de flujo. En la figura 21.19 se muestra la creación y el uso de los nuevos manipuladores de flujo **bell**, **ret** (retorno de carro), **tab** y **endLine**. Los usuarios también pueden crear sus propios manipuladores de flujo parametrizados —consulte sus manuales de instalación para encontrar las instrucciones de cómo llevar a cabo lo anterior.

21.7 Estados de formato de flujo

Varias *banderas de formato* especifican el tipo de formato a llevarse a cabo durante las operaciones de entrada/salida de flujo. Las funciones miembro **setf**, **unsetf** y **flags** controlan los ajustes de las banderas.


```
// fig21_19.cpp
// Testing user-defined, nonparameterized manipulators

#include <iostream.h>

// bell manipulator (using escape sequence \a)
ostream& bell(ostream& output)
{
   return output << '\a';
}

// ret manipulator (using escape sequence \r)
ostream& ret(ostream& output)
{
   return output << '\r';
}

// tab manipulator (using escape sequence \t)
ostream& tab(ostream& output)
{
   return output << '\t';
}

// endLine manipulator (using escape sequence \n
// and the flush member function)
ostream& endLine(ostream& output)
{
   return output << '\n' << flush;
}

main()
{
   cout << "Testing the tab manipulator:\n"
        << 'a' << tab << 'b' << tab << 'c' << endLine
        << "Testing the ret and bell manipulators:\n"
        << ".....";

   for (int i = 1; i <= 100; i++)
      cout << bell;

   cout << ret << "-----" << endLine;

   return 0;
}
```

```
Testing the tab manipulator:
a       b       c
Testing the ret and bell manipulators:
-----.....
```

Fig. 21.19 Cómo probar manipuladores definidos por usuario no parametrizados.

21.7.1 Banderas de estado de formato (setf, unsetf, flags)

Las banderas de estado de formato mostradas en la figura 21.20 se definen como una enumeración en la clase **ios** y son explicadas en la siguientes varias secciones.

Estas banderas pueden ser controladas por las funciones miembro **flags, setf** y **unsetf**, pero muchos programadores de C++ prefieren utilizar manipuladores de flujo (vea la sección 10.7.8). El programador puede utilizar la operación "OR a nivel de bits" **|**, para combinar varias opciones en un solo valor **int** (vea la figura 10.23). Llamar la función miembro **flags** para un flujo y especificar estas opciones de tipo "OR" define las opciones sobre dicho flujo y devuelve un valor que contiene las opciones anteriores. A menudo, este valor es guardado de tal forma que con este valor guardado pueda llamarse a **flags**, a fin de restaurar las opciones de flujo anteriores.

La función **flags** debe especificar un valor que represente los ajustes de todas las banderas. La función **setf** de un argumento, por otra parte, define una o más banderas "operadas con OR" o bien las opera con "OR" con los ajustes de banderas existentes a fin de formar un nuevo estado de formato.

El manipulador de flujo parametrizado **setiosflags** ejecuta las mismas funciones que la función **setf**. El manipulador de flujo **resetiosflags** lleva a cabo las mismas funciones que la función miembro **unsetf**. Para utilizar cualquiera de estos manipuladores de flujo, asegúrese de incluir **#include <iomanip.h>**.

La bandera **skipws** indica que **>>** deberá pasar por alto los espacios en blanco de un flujo de entrada. El comportamiento por omisión de **>>** es precisamente pasar por alto los espacios en blanco. Para modificar lo anterior, utilice la llamada **unsetf (ios::skipws)**. El manipulador de flujo **ws** pudiera también ser utilizado para definir que deben de saltarse los espacios en blanco.

21.7.2 Ceros a la derecha y puntos decimales (ios::showpoint)

La bandera **showpoint** se utiliza para obligar a un número de punto flotante a que se extraiga con punto decimal y ceros a la derecha. Sin la definición de **showpoint** un valor de punto flotante de **79.0** sería impreso como **79** y si **showpoint** estuviera definido sería impreso como **79.000000** (o con tantos ceros a la derecha como estuvieran especificados en la posición actual). El programa de la figura 21.21 muestra el uso de la función miembro **setf** para definir la bandera **showpoint** y controlar los ceros a la derecha y la impresión del punto decimal en el caso de valores de punto flotante.

```
ios::skipws
ios::left
ios::right
ios::internal
ios::dec
ios::oct
ios::hex
ios::showbase
ios::showpoint
ios::uppercase
ios::showpos
ios::scientific
ios::fixed
ios::unitbuf
ios::stdio
```

Fig. 21.20 Banderas de estado de formato.

21.7.3 Justificación (ios::left, los::right, ios::internal)

Las banderas **left** y **right** permiten que los campos sean justificados a la izquierda con caracteres de llenado a la derecha, o justificados a la derecha con caracteres de llenado a la izquierda, respectivamente. El carácter a utilizarse para el relleno se define mediante la función miembro **fill** o el manipulador de flujo parametrizado **setfill** (vea la sección 10.7.4). En la figura 21.22 se muestra el uso de los manipuladores **setw, setiosflags,** y **resetiosflags** así como las funciones miembro **setf** y **unsetf** para controlar la justificación a la derecha y a la izquierda de datos enteros dentro de un campo.

```
// fig21_21.cpp
// Controlling the printing of trailing
// zeros and decimal points when using
// floating point values with float values.

#include <iostream.h>
#include <iomanip.h>
#include <math.h>

main()
{
   cout << "cout prints 9.9900 as: " << 9.9900
        << "\ncout prints 9.9000 as: " << 9.9000
        << "\ncout prints 9.0000 as: " << 9.0000
        << "\n\nAfter setting the ios::showpoint flag\n";
   cout.setf(ios::showpoint);

   cout << "cout prints 9.9900 as: " << 9.9900
        << "\ncout prints 9.9000 as: " << 9.9000
        << "\ncout prints 9.0000 as: " << 9.0000 << endl;
   return 0;
}
```

```
cout prints 9.9900 as: 9.99
cout prints 9.9000 as: 9.9
cout prints 9.0000 as: 9

After setting the ios::showpoint flag
cout prints 9.9900 as: 9.990000
cout prints 9.9000 as: 9.900000
cout prints 9.0000 as: 9.000000
```

Fig. 21.21 Cómo controlar la impresión de ceros a la derecha y puntos decimales con valores de punto flotante.

```
// fig21_22.cpp
// Left-justification and right-justification.

#include <iostream.h>
#include <iomanip.h>

main()
{
   int x = 12345;

   cout << "Default is right justified:\n"
        << setw(10) << x
        << "\n\nUSING MEMBER FUNCTIONS\n"
        << "Use setf to set ios::left:\n" << setw(10);

   cout.setf(ios::left, ios::adjustfield);
   cout << x << "\nUse unsetf to restore default:\n";
   cout.unsetf(ios::left);
   cout << setw(10) << x
        << "\n\nUSING PARAMETERIZED STREAM MANIPULATORS"
        << "\nUse setiosflags to set ios::left:\n"
        << setw(10) << setiosflags(ios::left) << x
        << "\nUse resetiosflags to restore default:\n"
        << setw(10) << resetiosflags(ios::left)
        << x << endl;
   return 0;
}
```

```
Default is right justified:
    12345

USING MEMBER FUNCTIONS
Use setf to set ios::left:
12345
Use unsetf to restore default:
     12345

USING PARAMETERIZED STREAN MANIPULATORS
Use setiosflags to set ios::left:
12345
Use resetiosflags to restore default:
     12345
```

Fig. 21.22 Justificaciones a la izquierda y a la derecha.

La bandera **internal** indica que el signo de un número (o su base cuando la bandera **ios::showbase** está activa; vea la sección 21.7.5) deberá aparecer con justificación a la izquierda dentro de un campo, la magnitud del número deberá ser justificada a la derecha, y los espacios intermedios deberán ser rellenos con un carácter de llenado. Las banderas **left**, **right** e **internal** están contenidas en el miembro de datos estático **ios::adjustfield**. Al definir las banderas de justificación **left**, **right** o **internal**, el argumento **ios::adjustfield** deberá ser proporcionado como segundo argumento de **setf**. Esto activará a **setf** a fin de asegurarse que solamente una de las tres banderas de justificación esté activa (son mutuamente excluyentes). En la figura 21.23 se muestra el uso de los manipuladores de flujo **setiosflags** y **setw** para definir el espaciamiento interno. Note el uso de la bandera **ios::showpos** para obligar a la impresión del signo más.

21.7.4 Relleno (Fill, Setfill)

La *función miembro* **fill** especifica el carácter de llenado a utilizarse en campos ajustados; si no se especifica valor , se utilizarán espacios para el relleno. La función **fill** devuelve el carácter de relleno anterior. El *manipulador* **setfill** también define el carácter de relleno. En la figura 21.24 se demuestra el uso de la función miembro **fill** y del manipulador **setfill** para controlar la definición y redefinición del carácter de llenado.

21.7.5 Base de flujo integral (ios::dec, ios::oct, ios::hex, ios::showbase)

El *miembro estático* **ios::basefield** (que se usa de una manera similar que **ios::adjustfield** con **setf**), incluye los bits **oct**, **hex** y **dec** para especificar que los enteros deben ser tratados como valores octales, hexadecimales y decimales, en forma respectiva. Por omisión, las inserciones de flujo se hacen a valores decimales, si ninguno de estos bits está activo; el valor por omisión para las extracciones de flujo es procesar los datos en la misma forma en que han sido introducidos —los enteros que se inicien con **0** se tratan como valores octales, los enteros que se inicien con **0x** o **0X** se tratarán como valores hexadecimales, y todos los demás enteros serán tratados como valores decimales. Una vez especificada una base particular para un flujo, todos los enteros de dicho flujo serán procesados con dicha base, hasta que se especifique una nueva base, o hasta el final del programa.

```
// fig21_23.cpp
// Printing an integer with internal spacing and
// forcing the plus sign.

#include <iostream.h>
#include <iomanip.h>

main()
{
   cout << setiosflags(ios::internal | ios::showpos)
        << setw(10) << 123 << endl;

   return 0;
}
```

```
+       123
```

Fig. 21.23 Cómo imprimir un entero con espaciamiento interno y obligando a la impresión del signo más.

```
// fig21_24.cpp
// Using the fill member function and the setfill
// manipulator to change the padding character for
// fields larger than the values being printed.

#include <iostream.h>
#include <iomanip.h>

main()
{
   int x = 10000;

   cout << x
        << " printed as int right and left justified\n"
        << "and as hex with internal justification.\n"
        << "Using the default pad character (space):\n";
   cout.setf(ios::showbase);
   cout << setw(10) << x << '\n ;
   cout.setf(ios::left, ios::adjustfield);
   cout << setw(10) << x << '\n';
   cout.setf(ios::internal, ios::adjustfield);
   cout << setw(10) << hex << x << "\n\n";

   cout << "Using various padding characters:\n";
   cout.setf(ios::right, ios::adjustfield);
   cout.fill('*');
   cout << setw(10) << dec << x << '\n';
   cout.setf(ios::left, ios::adjustfield);
   cout << setw(10) << setfill('%') << x << '\n';
   cout.setf(ios::internal, ios::adjustfield);
   cout << setw(10) << setfill('^') << hex
        << x << endl;

   return 0;
}
```

```
10000 printed as int right and left justified
and as hex with internal justification.
Using the default pad character (space):
     10000
10000
0x   2710

Using various padding characters:
*****10000
10000%%%%%
0x****2710
```

Fig. 21.24 Cómo utilizar la función miembro **fill** y el manipulador **setfill** para modificar el carácter de relleno para campos mayores que los valores a imprimirse.

Active la bandera **showbase** para que se extraiga la base del valor completo. Los números decimales serán extraídos en forma normal, los números octales serán extraídos con un **0** a la izquierda, y los números hexadecimales serán extraídos ya sea con un **0x** a la izquierda o con un **0X** a la izquierda (la bandera **uppercase** determinará cuál de las opciones será seleccionada). En la figura 21.25 se demuestra el uso de la bandera **showbase** para obligar a que un entero se imprima en formatos decimal. octal y hexadecimales.

21.7.6 Números de punto flotante; notación científica (ios::scientific, ios::fixed)

La *bandera* ***ios::scientific*** y la *bandera* ***ios::fixed*** están incluidas en el *miembro de dato estático* ***ios::floatfield*** (que se usa en forma similar a **ios::adjustfield** e **ios::basefield** en **setf**). Se utilizan estas banderas para controlar el formato de salida de los números de punto flotante. La bandera **scientific** se activa para obligar a la salida de un número de punto flotante en formato científico. La bandera **fixed** obliga a un número de punto flotante a mostrar un número específico de dígitos (de acuerdo con lo especificado con la función miembro **precision**) a la derecha del punto decimal. Si estas banderas no están activadas, el valor del punto flotante determinará el formato de salida.

La llamada **cout.setf(0, ios::floatfield)** restaura el formato de sistema por omisión para la salida de número de punto flotante. En la figura 21.26 se demuestra la exhibición de los números de punto flotante, tanto en formatos fijos como científicos.

```
// fig21_25.cpp
// Using the ios::showbase flag

#include <iostream.h>
#include <iomanip.h>

main()
{
   int x = 100;

   cout << setiosflags(ios::showbase)
        << "Printing integers preceded by their base:\n"
        << x << '\n'
        << oct << x << '\n'
        << hex << x << endl;

   return 0;
}
```

```
Printing integers preceded by their base:
100
0144
0x64
```

Fig. 21.25 Cómo utilizar la bandera **ios::showbase**.

```
// fig21_26.cpp
// Displaying floating point values in system default,
// scientific, and fixed formats.

#include <iostream.h>

main()
{
   double x = .001234567, y = 1.946e9;

   cout << "Displayed in default format:\n"
        << x << '\t' << y << '\n';
   cout.setf(ios::scientific, ios::floatfield);
   cout << "Displayed in scientific format:\n"
        << x << '\t' << y << '\n';
   cout.unsetf(ios::scientific);
   cout << "Displayed in default format after unsetf:\n"
        << x << '\t' << y << '\n';
   cout.setf(ios::fixed, ios::floatfield);
   cout << "Displayed in fixed format:\n"
        << x << '\t' << y << endl;

   return 0;
}
```

```
Displayed in default format:
0.001235   1.946e+09
Displayed in scientific format:
1.234567e-03   1.946e+09
Displayed in default format after unsetf:
0.001235   1.946e+09
Displayed in fixed format:
0.001235   1946000000
```

Fig. 21.26 Cómo mostrar valores de punto flotante en formatos científicos, fijos y de sistema por omisión.

21.7.7 Control de mayúsculas/minúsculas (ios::uppercase)

La bandera **ios::uppercase** se activa para obligar la salida de una **X** o de una **E** en mayúsculas con enteros hexadecimales o con puntos con valores de punto flotante en notación científica, respectivamente (vea la figura 21.27). Cuando está activa, la bandera **ios::uppercase** hace que todas las letras en un valor hexadecimal aparezcan en mayúsculas.

21.7.8 Cómo activar y desactivar las banderas de formato (flags, setiosflags, resetiosflags)

La función miembro **flags**, sin argumento, sólo devuelve (como un valor **long**) los ajustes actuales de las banderas de formato. La función miembro **flags**, con un argumento **long**, define


```
// fig21_27.cpp
// Using the ios::uppercase flag
#include <iostream.h>
#include <iomanip.h>

main()
{
   cout << setiosflags(ios::uppercase)
        << "Printing uppercase letters in scientific\n"
        << "notation exponents and hexadecimal values:\n"
        << 4.345e10 << '\n'
        << hex << 123456789 << endl;

   return 0;
}
```

```
Printing uppercase letters in scientific
notation exponents and hexadecimal values:
4.345E+10
75BCD15
```

Fig. 21.27 Cómo utilizar la bandera **ios::uppercase**.

las banderas de formato según especificado en el argumento, y devuelve los ajustes de bandera anteriores. Cualquier otra bandera de formato que no se haya especificado en el argumento a **flags** será restablecida. El programa de la figura 21.28 demuestra el uso de la función miembro **flags** para definir un nuevo estado de formato, guardando el anterior, y a continuación restaurando los ajustes de formato original.

La función miembro **setf** define las banderas de formato incluidas en su argumento, y devuelve los ajustes previos de banderas como un valor **long**, como en

```
long previousFlagSettings =
   cout.setf(ios::showpoint | ios::showpos);
```

La función miembro **setf**, con dos argumentos **long**, como en

```
cout.setf(ios::left, ios::adjustfield);
```

primero desactiva los bits de **ios::adjustfield** y a continuación define la bandera **ios::left**. Esta versión de **setf** es utilizada con los campos de bits asociados con **ios::basefield** (representados por **ios::dec**, **ios::oct** e **ios::hex**), **ios::floatfield** (representado por **ios:scientific** e **ios::fixed**), e **ios::adjustfield** (representado por **ios::left**, **ios::right** e **ios::internal**).

La función miembro **unsetf** vuelve a activar las banderas designadas y devuelve el valor de las banderas existentes antes de que hayan sido reactivadas.

21.8 Estados de error de flujo

El estado de un flujo puede ser probado mediante bits en la clase **ios** —la clase base correspondiente a las clases **istream**, **ostream** e **iostream** que estamos utilizando para entradas/salidas.

```
// fig21_28.cpp
// Demonstrating the flags member function.
#include <iostream.h>

main()
{
   int i = 1000;
   double d = 0.0947628;

   cout << "The value of the flags variable is: "
        << cout.flags() << '\n'
        << "Print int and double in original format:\n"
        << i << '\t' << d << "\n\n";
   long originalFormat = cout.flags(ios::oct | ios::scientific);
   cout << "The value of the flags variable is: "
        << cout.flags() << '\n'
        << "Print int and double in a new format\n"
        << "specified using the flags member function:\n"
        << i << '\t' << d << "\n\n";
   cout.flags(originalFormat);
   cout << "The value of the flags variable is: "
        << cout.flags() << '\n'
        << "Print values in original format again:\n"
        << i << '\t' << d << endl;

   return 0;
}
```

```
The value of the flags variable is: 1
Print int and double in original format:
1000    0.094763

The value of the flags variable is: 4040
Print int and double in a new format
specified using the flags member function:
1750    9.47628e-02

The value of the flags variable is: 1
Print values in original format again:
1000    0.094763
```

Fig. 21.28 Demostración de la función miembro **flags**.

El **eofbit** queda automáticamente activado para un flujo de entrada cuando un fin de archivo es encontrado. Un programa puede utilizar la función miembro **eof** para determinar si en un flujo se ha encontrado un fin de archivo. La llamada

```
cin.eof()
```

devuelve verdadero si en **cin** se ha encontrado un fin de archivo, y falso de lo contrario.

El **failbit** queda activado para un flujo cuando en el mismo ocurre un error de formato, pero sin pérdida de caracteres. La función miembro **fail** determina si ha fallado la operación de un flujo; por lo regular es posible recuperarse de dichos errores.

El **badbit** se activa para un flujo cuando ocurre un error que resulte en pérdida de datos. La función miembro **bad** determina si ha fallado una operación de flujo. Estas fallas serias por lo regular no son recuperables.

El **goodbit** se activa en un flujo si para dicho flujo no se han activado ninguno de los bits **eofbit**, **failbit**, o **badbit**.

La función miembro **good** devuelve verdadero si las funciones **bad**, **fail** y **eof** devolvieran todas ellas falso. Las operaciones de entrada/salida deberían únicamente ser llevadas a cabo sobre flujos "buenos".

La función miembro **rdstate** devuelve el estado de error del flujo. Una llamada a **cout.rdstate**, por ejemplo, devolvería el estado del flujo que a continuación podría ser probado mediante un enunciado **switch** que examinase **ios::eofbit**, **ios::badbit**, **ios::failbit** e **ios::goodbit**. La manera preferida de probar el estado de un flujo es utilizando las funciones miembro **eof**, **bad**, **fail** y **good** —el uso de estas funciones no requiere que el programador esté familiarizado con bits de estado particulares.

La función miembro **clear** se utiliza por lo general, para restaurar el estado de un flujo a "bueno", de tal forma que puedan continuar las entradas/salidas sobre dicho flujo. El argumento por omisión para **clear** es **ios::goodbit**, por lo que el enunciado

```
cin.clear();
```

elimina o limpia **cin** y activa **goodbit** para el flujo. El enunciado

```
cin.clear(ios::failbit)
```

es el que verdaderamente activa **failbit**. El usuario quizás desease hacer esto al ejecutar una entrada en **cin** con un tipo definido por usuario y al encontrar un problema. En este contexto el nombre **clear** parece inapropiado, pero es correcto.

El programa de la figura 21.29 ilustra el uso de las funciones miembro **rdstate**, **eof**, **fail**, **bad**, **good**, y **clear**.

La función miembro **operator!** devuelve verdadero ya sea que **badbit** esté activo, **failbit** o ambos. La función miembro **operator void*** devuelve falso si **badbit**, **failbit** o ambos están activos. Estas funciones resultan útiles en el procesamiento de archivos, cuando se hacen pruebas de condiciones verdaderas/falsas dentro de la condición de una estructura de selección o de repetición.

21.9 Entradas/salidas de tipos definidos por usuario

C++ es capaz de introducir y extraer los tipos de datos estándar utilizando los operadores de extractor de flujo >> y de inserción de flujo <<. Estos operadores son homónimos, a fin de poder procesar cada tipo de datos estándar, incluyendo cadenas y direcciones en memoria. El programador puede hacer la homonimia de los operadores de inserción y de extracción de flujo, a fin de llevar a cabo entradas y salidas de tipos definidos por usuario. En la figura 21.30 se demuestra la homonimia de operadores de extracción y de inserción de flujo, a fin de manejar datos de una clase de número telefónicos definidos por usuario llamada **PhoneNumber**. Note que este programa supone que los números telefónicos están introducidos correctamente. Dejamos como ejercicio el incorporar la correspondiente verificación de errores.

```
// fig21_29.cpp
// Testing error states.

#include <iostream.h>

main()
{
   int x;
   cout << "Before a bad input operation:\n"
        << "cin.rdstate(): " << cin.rdstate()
        << "\n    cin.eof(): " << cin.eof()
        << "\n   cin.fail(): " << cin.fail()
        << "\n    cin.bad(): " << cin.bad()
        << "\n   cin.good(): " << cin.good()
        << "\n\nExpects an integer, but enter a character: ";
   cin >> x;

   cout << "\nAfter a bad input operation:\n"
        << "cin.rdstate(): " << cin.rdstate()
        << "\n    cin.eof(): " << cin.eof()
        << "\n   cin.fail(): " << cin.fail()
        << "\n    cin.bad(): " << cin.bad()
        << "\n   cin.good(): " << cin.good() << "\n\n";

   cin.clear();

   cout << "After cin.clear()\n"
        << "cin.fail(): " << cin.fail() << endl;

   return 0;
}
```

```
Before a bad input operation:
cin.rdstate(): 0
    cin.eof(): 0
   cin.fail(): 0
    cin.bad(): 0
   cin.good(): 1

Expects an integer, but enter a character: A

After a bad input operation:
cin.edstate(): 2
    cin.eof(): 0
   cin.fail(): 2
    cin.bad (): 0
   cin.good(): 0

After cin.clear()
cin.fail(): 0
```

Fig. 21.29 Cómo probar estados de error.

El operador de extracción de flujo toma como argumentos una referencia **istream** y una referencia a un tipo definido por usuario (en este caso **PhoneNumber**), y devuelve una referencia **istream**. En la figura 21.30, el operador de extracción de flujo homónimo es utilizado para introducir los números telefónicos de la forma

```
(800) 555-1212)
```

en objetos del tipo **PhoneNumber**. La función de operador lee las tres partes de un número telefónico en los miembros **areaCode**, **exchange** y **line** del objeto referenciado **PhoneNumber** (**num** en la función de operador y **phone** en **main**). Los caracteres de paréntesis, de espacio y de guiones serán descartados mediante el llamado a la función miembro **ignore** de **istream**. La función de operador devuelve la referencia **input** de **istream**. Al devolver la referencia de flujo, las operaciones de entrada en objetos **PhoneNumber** pueden ser concatenadas con operaciones de entrada sobre otros objetos **PhoneNumber** o con otros tipos de datos. Por ejemplo, dos objetos **PhoneNumber** podrían ser introducidos como sigue:

```
cin >> phone1 >> phone2;
```

El operador de inserción de flujo toma como argumentos una referencia **ostream** y una referencia a un tipo definido por usuario (en este caso **PhoneNumber**), y devuelve una referencia **ostream**. En la figura 21.30, el operador de inserción de flujo homónimo exhibe objetos del tipo **PhoneNumber**, de la misma forma en que fueron introducidos. La función **operator** exhibe las partes del número telefónico como cadenas, porque están almacenados en formato de cadena (recuerde que la función miembro **getline** de **istream** almacena un carácter nulo después de que termina su entrada).

Las funciones homónimas **operator** se declaran en **class PhoneNumber** como funciones **friend**. Los operadores de entrada y de salida homónimos deben de ser declarados como **friend**, para que puedan tener acceso a miembros de clase no públicos. Los amigos de una clase pueden tener acceso a los miembros de clase privados.

Observación de ingeniería de software 21.3

Se pueden añadir a C++ nuevas capacidades de entrada/salida para tipos definidos por usuario, sin modificar las declaraciones o los miembros de datos privados, ya sea para la clase **ostream** *o para la clase* **istream**. *Esto fomenta la extensibilidad del lenguaje de programación C++ —lo que es uno de los aspectos más atractivos de C++.*

21.10 Cómo ligar un flujo de salida con un flujo de entrada

Las aplicaciones interactivas generalmente involucran un **istream** como entrada y un **ostream** como salida. Cuando en la pantalla aparece un mensaje solicitando algo, el usuario responde introduciendo los datos apropiados. Obviamente, la solicitud debe aparecer antes de que se inicie la operación de entrada. Con almacenamiento temporal de salida, las salidas sólo aparecerán cuando el búfer esté lleno, cuando las salidas se vacíen en forma explícita por requerimientos del programa, o bien en forma automática al final del programa. C++ tiene la función miembro **tie** para sincronizar, es decir, "para ligar" la operación de un **istream** con un **ostream** para asegurar que las salidas aparezcan antes de sus entradas subsecuentes. Una llamada como

```
// fig21_30.cpp
// User-defined insertion and extraction operators

#include <iostream.h>

class PhoneNumber {
   friend ostream& operator<<(ostream&, PhoneNumber&);
   friend istream& operator>>(istream&, PhoneNumber&);
private:
   char areaCode[4];
   char exchange[4];
   char line[5];
};

ostream& operator<<(ostream& output, PhoneNumber& num)
{
   output << "(" << num.areaCode << ") "
          << num.exchange << "-" << num.line;

   return output;
}

istream& operator>>(istream& input, PhoneNumber& num)
{
   input.ignore();                    // skip (
   input.getline(num.areaCode, 4);
   input.ignore(2);                   // skip ) and space
   input.getline(num.exchange, 4);
   input.ignore();                    // skip -
   input.getline(num.line, 5);

   return input;
}

main()
{
   PhoneNumber phone;

   cout << "Enter a phone number in the "
        << "form (123) 456-7890:\n";
   cin >> phone;
   cout << "The phone number entered was:\n"
        << phone << endl;

   return 0;
}
```

```
Enter a phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was:
(800) 555-1212
```

Fig. 21.30 Operadores de inserción y de extracción de flujo definidos por usuario.


```
cin.tie(cout);
```

Liga a **cout** (un **ostream**) con **cin** (un **istream**). De hecho, esta llamada en particular resulta redundante, porque C++ lleva a cabo esta operación en forma automática para crear el entorno estándar de entrada/salida. El usuario podría, sin embargo, ligar en forma explícita otros pares **istream/ostream**. A fin de desligar un flujo de entrada, **inputStream**, de un flujo de salida, utilice la llamada

```
inputStream.tie(0);
```

Resumen

- Las operaciones de entrada/salida se ejecutan de una forma que depende o es sensible al tipo de los datos.
- Las entradas y salidas en C++ ocurren en flujos de bytes. Un flujo es sólo una secuencia de bytes.
- Los mecanismos de entrada/salida del sistema mueven bytes de dispositivos a la memoria y viceversa de una forma eficiente y confiable.
- C++ proporciona capacidades de entrada/salida de "bajo nivel" y de "alto nivel". Las capacidades de entrada/salida de bajo nivel especifica que cierto número de bytes deberán ser transferidos del dispositivo a la memoria o de la memoria al dispositivo. Las entradas/salidas de alto nivel se ejecutan con bytes agrupadas en unidades significativas como son enteros, puntos flotantes, caracteres, cadenas y tipos definidos por usuario.
- C++ proporciona operaciones de entradas/salidas tanto con como sin formato. Las entradas y salidas sin formato son rápidas, pero procesan datos en bruto difíciles de utilizar por las personas. Las entradas/salidas con formato procesan los datos en unidades significativas, pero requieren de tiempo de proceso adicional que puede tener un impacto negativo en transferencias de datos a altos volúmenes.
- La mayor parte de los programas de C++ incluyen un archivo de cabecera **iostream.h** que contiene la información básica requerida para todas las operaciones de entradas/salidas de flujo.
- El encabezado **iomanip.h** contiene información para entradas/salidas con formato mediante manipuladores de flujo parametrizados.
- El encabezado **fstream.h** contiene información para operaciones de procesamiento de archivos.
- El encabezado **strstream.h** contiene información para dar formato en memoria.
- El encabezado **stdiostream.h** contiene información para aquellos programas que mezclan los estilos de entradas/salidas de C con las de C++.
- La clase **istream** acepta operaciones de entrada de flujo.
- La clase **ostream** acepta operaciones de salida de flujo.
- La clase **iostream** acepta tanto operaciones de entrada como operaciones de salida de flujo.
- Las clases **istream** y **ostream** son ambas derivadas mediante herencia simple a partir de la clase base **ios**.
- La clase **iostream** es derivada mediante herencia múltiple, tanto de la clase **istream** como de la clase **ostream**.

- Se hace la homonimia del operador de desplazamiento a la izquierda (<<), para designar salida de flujo y se le conoce como el operador de inserción de flujo.
- Se hace la homonimia del operador de desplazamiento a la derecha (>>) para designar entrada de flujo y se le conoce como el operador de extracción de flujo.
- El objeto **cin** de la clase **istream** está ligado al dispositivo de entrada estándar, que por lo general, es el teclado.
- El objeto **cout** de la clase **ostream** está ligado al dispositivo de salida estándar, que por lo regular es la pantalla de exhibición.
- El objeto **cerr** de la clase **ostream** está ligado con el dispositivo de error estándar. Las salidas a **cerr** no pasan por búfer; cada inserción a **cerr** aparece de inmediato.
- El manipulador de flujo **endl** emite un carácter de nueva línea y vacía el búfer de salida.
- El compilador de C++ determina automáticamente los tipos de datos para la entrada y la salida.
- Por omisión las direcciones se exhiben en formato hexadecimal.
- Para imprimir la dirección de una variable del tipo **char***, convierta explícitamente (cast) el apuntador a **void***.
- La función miembro **put** extrae un carácter. Las llamadas a **put** pueden ser concatenadas.
- La salida de flujo se lleva a cabo mediante el operador de extracción de flujo >>. Este operador en forma automática pasa por alto caracteres de espacio en blanco existentes en el flujo de entrada.
- El operador >> devuelve falso cuando en un flujo encuentra la señal de fin de archivo.
- La extracción de flujo hace que se active **failbit** cuando se trate de una entrada incorrecta, y que se active **badbit** si la operación falla.
- Una serie de valores pueden ser introducidos mediante la operación de extracción de flujo en un encabezado de ciclo **while**. La extracción devolverá falso (cero) cuando se encuentre fin de archivo.
- La función miembro **get**, sin argumentos, introduce un carácter y devuelve el carácter; si encuentra en el flujo la señal de fin de archivo, se devolverá **EOF**.
- La función miembro **get**, con un argumento del tipo **char**, introduce un carácter. Se devolverá falso al encontrar el fin de archivo; de lo contrario será devuelto el objeto **istream** para el cual se está invocando la función miembro **get**.
- La función miembro **get**, con tres argumentos, un arreglo de caracteres, un límite de tamaño y un delimitador (con el valor de nueva línea por omisión) —lee caracteres del flujo de entrada hasta un máximo de límite - 1 caracteres y termina, o termina cuando se lea el delimitador. La cadena de entrada se terminará incluyendo un carácter nulo. El delimitador no se coloca en el arreglo de caracteres, sino que se conserva dentro del flujo de entrada.
- La función miembro **getline** opera en forma similar a la función miembro **get** con tres argumentos. La función **getline** elimina el delimitador del flujo de entrada.
- La función miembro **ignore** pasa por alto el número específico de caracteres (su valor por omisión es un carácter) dentro del flujo de entrada; termina si encuentra el delimitador especificado (el delimitador por omisión es **EOF**).
- La función miembro **putback** coloca el carácter previamente obtenido por un **get** de un flujo de regreso en dicho flujo.

- La función miembro **peek** devuelve el siguiente carácter de un flujo de entrada, pero sin eliminar dicho carácter de un flujo.
- C++ ofrece entrada/salida de tipo seguro. Si se procesan datos inesperados por los operadores << y >>, se definirán varias banderas de error, que el usuario podrá probar para determinar si una operación de entrada/salida ha tenido éxito o ha fallado.
- Las entradas/salidas sin formato se llevan a cabo con las funciones miembro **read** y **write**. Estas introducen o extraen un cierto número de bytes, hacia o de la memoria, empezando en una dirección designada de memoria. Se introducen o se extraen como bytes en bruto sin formato.
- La función miembro **gcount** devuelve el número de caracteres introducidos mediante la operación anterior **read** sobre dicho flujo.
- La función miembro **read** introduce un número especificado de caracteres en un arreglo de caracteres. Si se han leído menos que el número especificado de caracteres, se activa **failbit**.
- Para modificar la base numérica en la cual se extraen enteros, utilice el manipulador **hex** para definir la base a hexadecimal (de base 16) o bien **oct** para definir la base a octal (de base 8). Utilice el manipulador **dec** para restaurar la base a decimal. La base se conservará sin modificación en tanto no sea cambiada en forma explícita.
- El manipulador de flujo parametrizado **setbase** también define la base de la salida de enteros. **setbase** toma un argumento entero de **10**, **8**, ó **16** para definir la base.
- La precisión de punto flotante puede ser controlada utilizando o el manipulador de flujo **setprecision** o la función miembro **precision**. Ambos definen la precisión para todas las operaciones de salida subsecuentes hasta la siguiente llamada de ajuste de precisión. La función miembro **precision**, sin argumentos, devuelve el valor de precisión actual. Una precisión de **0** define el valor de precisión por omisión (**6**).
- Los manipuladores parametrizados requieren la inclusión del archivo de cabecera **iomanip.h**.
- La función miembro **width** define el ancho de campo y devuelve el ancho anterior. Los valores que sean menores que el campo serán llenados con caracteres de relleno. El ajuste de ancho de campo se aplica sólo para la siguiente inserción o extracción; después el ancho de campo se define en forma implícita a **0** (los valores subsecuentes serán extraídos del tamaño que necesiten tener). Los valores mayores que el tamaño del campo son impresos por completo. La función **width** sin argumentos, devuelve el ajuste actual de ancho. El manipulador **setw** también define el ancho.
- Para las entradas, el manipulador de flujo **setw** establece un tamaño máximo de cadena; si se introduce una cadena más grande, la línea más grande se divide en piezas no mayores que el tamaño designado.
- Los usuarios pueden crear sus propios manipuladores de flujo.
- Las funciones miembro **setf**, **unsetf**, y **flags** controlan los ajustes de bandera.
- La bandera **skipws** indica que en un flujo de entrada el operador >> deberá pasar por alto los espacios en blanco. El manipulador de flujo **ws** también pasa por alto los espacios en blanco a la izquierda en un flujo de entrada.
- Las banderas de formato se definen como una enumeración en la clase **ios**.
- Las banderas de formato están controladas por las funciones miembro **flags**, y **setf**, pero muchos programadores de C++ prefieren utilizar manipuladores de flujo. La operación OR a

nivel de bits, |, puede ser utilizada para combinar varias opciones en un valor único `long`. Llamar la función miembro `flags` para un flujo y especificar estas opciones con la función OR define las opciones de dicho flujo y devuelve un valor `long` que contiene las opciones precedentes. A menudo, este valor es guardado, por lo que `flags` puede ser llamada con este valor guardado, a fin de restaurar las opciones de flujo anteriores.

- La función `flags` debe definir un valor que represente los ajustes totales de todas las banderas. La función `setf` con un argumento, por otra parte, hace la operación a nivel de bit "OR" correspondiente a las banderas especificadas con los ajustes existentes de bandera, para formar un nuevo estado de formato.
- La bandera `showpoint` se define para obligar a un número de punto flotante a que sea extraído con un punto decimal y una cantidad de dígitos significativos según esté definido en la precisión.
- Las banderas `left` y `right` hacen que los campos queden justificados a la izquierda con caracteres de relleno a la derecha, o a la derecha con caracteres de relleno a la izquierda, respectivamente.
- La bandera `internal` indica que el signo de un número (o la base cuando se trate de la bandera `ios::showbase`) deberá quedar justificado a la izquierda dentro de un campo, la magnitud justificada a la derecha, y llenar los espacios intermedios con un carácter de relleno.
- `ios::adjustfield` contiene las banderas `left`, `right`, e `internal`.
- La función miembro `fill` define el carácter de relleno a utilizar con los campos ajustados `left`, `right`, e `internal` (un espacio en blanco es el valor por omisión); el carácter de relleno anterior es devuelto. El manipulador de flujo `setfill` también define el carácter de relleno.
- El miembro estático `ios::basefield` incluye los bits `oct`, `hex`, y `dec` a fin de especificar que los enteros deben ser tratados como valores octal, hexadecimal y decimales, respectivamente. Si no está activo ninguno de estos bits, la salida de enteros por omisión se hace en decimal; las extracciones de flujo procesan los datos en la misma forma en la cual fueron suministrados.
- Defina la bandera `showbase` para obligar la base de la extracción de un valor integral.
- El miembro de datos estático `ios::floatfield` contiene las banderas `scientific` y `fixed`. Defina la bandera `scientific` para extraer un número en punto flotante en formato científico. Defina la bandera `fixed` para extraer un número de punto flotante con la precisión especificada por la función miembro `precision`.
- La llamada `cout.setf(0, ios::floatfield)` restaura el formato por omisión para la exhibición de los números de punto flotante.
- Active la bandera `uppercase` para obligar a la extracción de una `X` o `E` mayúsculas con los enteros hexadecimales o en la notación científica de valores de punto flotante, respectivamente. Cuando esté activa, la bandera `ios::uppercase` hará que todas las letras en un valor hexadecimal aparezcan en mayúsculas.
- La función miembro `flags` sin argumento, devuelve el valor `long` de los ajustes actuales de las banderas de formato. La función miembro `flags` con el argumento `long` define las banderas de formato especificadas por el argumento y devuelve los ajustes anteriores de bandera.
- La función miembro `setf` define las banderas de formato en su argumento y devuelve los ajustes de banderas anteriores como un valor `long`.

- La función miembro `setf` con dos argumentos `long` —un bit `long` y un campo de bits `long`— elimina los bits en el campo de bits, y a continuación ajusta el bit del primer argumento.
- La función miembro `unsetf` restaura las banderas designadas y devuelve el valor de las banderas antes de su restauración.
- El manipulador de flujo parametrizado `setiosflags` ejecuta las mismas funciones que la función miembro `flags`.
- El manipulador de flujo parametrizado `resetiosflags` ejecuta las mismas funciones que la función miembro `unsetf`.
- El estado de un flujo puede ser probado mediante bits en la clase `ios`.
- El `iofbit` es activado para un flujo de entrada cuando se encuentra un fin de archivo durante una operación de entrada. La función miembro `eof` se utiliza para determinar si `eofbit` está activo.
- El `failbit` se activa para un flujo cuando en dicho flujo ocurrió un error de formato, pero sin pérdida de caracteres. La función miembro `fail` determina si una operación de flujo ha fallado; por lo general, es posible recuperarse de dichos errores.
- `badbit` se activa en un flujo cuando ha ocurrido un error cuyo resultado es pérdida de datos. La función miembro `bad` determina si un operación de flujo ha fallado. Estas fallas serias, por lo regular no son recuperables.
- La función miembro `good` devuelve verdadero si las funciones `bad`, `fail` y `eof` todas ellas devuelven falso. Las operaciones de entrada/salida únicamente deberían ser llevadas a cabo sobre flujos "buenos".
- La función miembro `rdstate` devuelve el estado de error del flujo.
- La función miembro `clear` se utiliza por lo regular para regresar el estado del flujo a "bueno", de tal forma que se pueda proceder con entradas y salidas sobre dicho flujo.
- El usuario puede hacer la homonimia de los operadores de inserción y de extracción de flujo para llevar a cabo entradas/salidas para tipos definidos por usuario.
- El operador de extracción de flujo homónimo toma como argumento una referencia a `istream` y una referencia a un tipo definido por usuario, y devuelve una referencia a `istream`.
- El operador de inserción de flujo homónimo toma como referencias una referencia a `ostream` y una referencia a un tipo definido por usuario, y devuelve una referencia `ostream`.
- A menudo, se declaran funciones homónimas `operator` como funciones `friend` a una clase; esto permite el acceso a miembros de clase no públicos.
- C++ proporciona la función miembro `tie` para sincronizar operaciones `istream` y `ostream` para asegurarse que las salidas aparezcan antes de entradas subsecuentes.

Terminología

función miembro `bad`
`badbit`
`cerr`
`cin`
función miembro `clear`
`clog`
`cout`
manipulador de carácter de flujo `dec`
lcarácter de llenado por omisión (espacio en blanco)
precisión por omisión

fin de archivo
endl
función miembro **eof**
eofbit
extensibilidad
función miembro **fail**
failbit
ancho de campo
carácter de llenado
función miembro **fill**
función miembro **flags**
función miembro **flush**
manipulador de flujo **flush**
banderas de formato
estados de formato
entrada/salida con formato
clase **fstream**
función miembro **gcount**
función miembro **get**
función miembro **getline**
función miembro **good**
manipulador de flujo **hex**
clase **ifstream**
función miembro **ignore**
formato en núcleo
formato en memoria
archivo de cabecera estándar **<iomanip.h>**
clase **ios**
ios::adjustfield
ios::basefield
ios::fixed
ios::floatfield
ios:internal
ios::scientific
ios::showbase
ios::showpoint
ios::showpos
clase iostream
clase istream
0 a la derecha (octal)
0x ó **0X** a la izquierda (hexadecimal)
left
justificado a la izquierda
manipulador de flujo oct
clase ofstream
función miembro **operator!**
función miembro operator void *
clase ostream
relleno
manipulador de flujo parametrizado
función miembro peek
función miembro precision
flujo predeterminado
función miembro put
función miembro **putback**
función miembro **rdstate**
función miembro **read**
manipulador de flujo resetiosflags
justificado a la derecha
manipulador de flujo setbase
función miembro setf
manipulador de flujo setfill
manipulador de flujo setiosflags
manipulador de flujo **setprecision**
manipulador de flujo setw
skipws
bibliotecas de clases de flujo
operador de extracción de flujo (**>>**)
entrada de flujo
operador de inserción de flujo (<<)
manipulador **stream**
salida de flujo
función miembro tie
entrada/salida de tipo seguro
entrada/salida sin formato
función miembro **unsetf**
uppercase
flujos definidos por usuario
caracteres de espacio en blanco
width
función miembro **write**
función miembro **ws**

Errores comunes de programación

21.1 Intentar leer de un **ostream** (o de cualquier otro flujo de sólo salida).
21.2 Intentar escribir a un **istream** (o a cualquier otro flujo de sólo entrada).
21.3 Omitir paréntesis para obligar a una correcta precedencia al utilizar los operadores de inserción de flujo **<<** o de extracción de flujo **>>** que tienen una relativamente alta precedencia.
21.4 Cuando no se de un campo lo suficiente amplio para manejar salidas, dichas salidas se imprimirán del ancho que requieran, posiblemente causando salidas erróneas o difíciles de leer.

Prácticas sanas de programación

21.1 En programas C++, utilice exclusivamente la forma de entradas/salidas de C++, a pesar del hecho que para los programadores C++ esté disponible el estilo de entradas/salidas de C.
21.2 Al extraer expresiones, colóquelas entre paréntesis, para evitar problemas de precedencia de operadores entre los operadores de la expresión y el operador <<.

Sugerencia de rendimiento

21.1 Utilice entradas/salidas sin formato para un rendimiento máximo en procesos de alto volumen de archivos.

Observaciones de ingeniería de software

21.1 El estilo de entradas/salidas de C++ es de tipo seguro.
21.2 C++ permite un tratamiento común de entradas/salidas de tipos predefinidos y de tipos definidos por usuario. Este tipo de estado común facilita el desarrollo de software en general y de la reutilización de software en particular.
21.3 Se pueden añadir a C++ nuevas capacidades de entrada/salida para tipos definidos por usuario, sin modificar las declaraciones o los miembros de datos privados, ya sea para la clase **ostream** o para la clase **istream**. Esto fomenta la extensibilidad del lenguaje de programación C++ —lo que es uno de los aspectos más atractivos de C++.

Ejercicios de autoevaluación

21.1 Llene cada uno de los siguientes espacios vacíos:
a) Las funciones de operador de flujo homónimas deben ser definidas como funciones _____ de una clase.
b) Los bits de formato de justificación que se pueden definir son _____, _____, y _____.
c) En C++ la entrada/salida ocurre como un _____ de bytes.
d) Los manipuladores de flujo parametrizados _____, y _____ se pueden utilizar para activar y desactivar banderas de estado de formato.
e) La mayor parte de los programas de C++ deben incluir el archivo de cabecera _____ que contiene información básica requerida para todas las operaciones de flujo de entrada/salida.
f) Las funciones miembro _____, y _____ pueden ser utilizadas para activar y reactivar banderas de estado de formato.
g) El archivo de cabecera _____ contiene información para llevar a cabo el formato "en memoria".
h) Al utilizar los manipuladores parametrizados, se debe incluir en el programa el archivo de cabecera _____.
i) El archivo de cabecera _____ contiene información para llevar a cabo procesamiento de archivos controlados por usuario.
j) El manipulador de flujo _____ inserta un carácter de nueva línea en el flujo de salida y vacía el flujo de salida.
k) El archivo de cabecera _____ contiene información para aquellos programas que mezclan entradas y salidas en estilo C y en estilo C++.
l) La función miembro **ostream** _____ se utiliza para llevar a cabo salidas sin formato.
m) Las operaciones de entrada son soportadas por la clase _____.
n) Las salidas al flujo de error estándar son dirigidas ya sea por el objeto de flujo _____ o _____.
o) Las operaciones de salida están soportadas por la clase _____.
p) El símbolo para el operador de inserción de flujo es _____.

q) Los cuatro objetos que corresponden a los dispositivos estándar en el sistema incluyen _____, _____, _____, y _____.
r) El símbolo para operador de extracción de flujo es _____.
s) Los manipuladores de flujo _____, _____, y _____ se utilizan para especificar que los enteros deben ser mostrados en formatos octal, hexadecimal y decimal, respectivamente.
t) La precisión por omisión para mostrar valores en punto flotante es _____.
u) Cuando está activa, la bandera _____ hace que los números positivos numéricos aparezcan con un signo más.

21.2 Indique si lo que sigue es verdadero o falso. Si la respuesta es falsa, explique por qué.
a) La función miembro de flujo `flags()` con un argumento long, define a su argumento a la variable de estado `flags` y devuelve su valor anterior.
b) El operador de inserción de flujo << y el operador de extracción de flujo >> son homónimos para manejar todos los tipos de datos estándar —incluyendo cadenas y direcciones de memoria, así como todos los tipos de datos definidos por usuario.
c) La función miembro de flujo `flags()` sin argumentos, restaura todos los bits de bandera en la variable de estado `flags`.
d) Se puede hacer la homonimia del operador de extracción de flujo >> con una función operador que toma como argumentos una referencia `istream` y una referencia a un tipo definido por usuario, y devuelve una referencia `istream`.
e) El manipulador de flujo `ws` pasa por alto el espacio en blanco a la izquierda de un flujo de entrada.
f) Se puede hacer la homonimia del operador de inserción de flujo << con una función operador que toma como referencia una referencia a `istream` y una referencia a un tipo definido por usuario y devuelve una referencia `istream`.
g) La entrada con el operador de extracción de flujo >> siempre pasará por alto los caracteres de espacio en blanco del flujo de entrada.
h) Las características de entrada y de salida son parte de C++.
i) La función miembro de flujo `rdstate()` devuelve el estado del flujo actual.
j) El flujo `cout` por lo general, está conectado a la pantalla de exhibición.
k) La función miembro de flujo `good()` devuelve verdadero si las funciones miembro `bad()`, `fail()` y `eof()` todas ellas devuelven falso.
l) El flujo `cin` por lo regular está conectado a la pantalla de exhibición.
m) Si ocurre un error no recuperable durante una operación de flujo, la función miembro `bad()` devolverá verdadero.
n) La salida a `cerr` no tiene búfer y la salida a `clog` sí lo tiene.
o) Cuando está activa la bandera `ios::showpoint`, los valores en punto flotante están forzados a imprimirse con los seis dígitos de precisión por omisión —siempre y cuando el valor de la precisión no haya sido modificado, en cuyo caso los valores de punto flotante se imprimirán con la precisión especificada.
p) La función miembro `ostream` de nombre `put` extrae el número especificado de caracteres.
q) Los manipuladores de flujo `dec`, `oct` y `hex` solamente afectan la siguiente operación de salida de enteros.
r) Al ser extraído, por omisión las direcciones de memoria se exhiben como enteros `long`.

21.3 Escriba un solo enunciado que ejecute la tarea indicada para cada uno de los siguientes.
a) Extraiga la cadena `"Enter your name: "`.
b) Active una bandera para que el exponente en notación científica y las letras en los valores hexadecimales se impriman en mayúsculas.
c) Extraiga la dirección de la variable `string` del tipo `char *`.
d) Active una bandera, de tal forma que los valores de punto flotante se impriman en notación científica.

e) Extraiga la dirección de la variable `integerPtr` del tipo `int *`.
f) Active una bandera, de tal forma que cuando valores enteros sean extraídos se exhiban la base entera para valores octal y hexadecimal.
g) Extraiga el valor al cual apunta `floatPtr` del tipo `float *`.
h) Utilice una función miembro de flujo para definir el carácter de llenado como `'*'` para imprimir en anchos de campo mayores que los valores siendo extraídos. Escriba un enunciado por separado para ello mediante un manipulador de flujo.
i) Extraiga los caracteres `'O'` y `'K'` en un enunciado mediante la función `put` de `ostream`.
j) Obtenga el siguiente carácter en el flujo de entrada sin extraerlo del mismo flujo.
k) Introduzca un solo carácter en la variable `c` de tipo `char` utilizando la función miembro `istream get` de dos formas distintas.
l) Introduzca y descarte los siguientes seis caracteres del flujo de entrada.
m) Utilice la función miembro `read` de `istream` para introducir 50 caracteres al arreglo `line` del tipo `char`.
n) Lea 10 caracteres en el arreglo de caracteres `name`. Si se encuentra el delimitador `'.'` detenga la lectura de caracteres. No retire el delimitador del flujo de entrada. Escriba otro enunciado que lleve a cabo esa tarea y elimine el delimitador del flujo de entrada.
o) Utilice la función miembro `gcount` de `istream` para determinar el número de caracteres introducidos en el arreglo de caracteres `line` por la última llamada a la función miembro `read` de `istream` y extraiga dicho número de caracteres utilizando la función miembro `write` de `ostream`.
p) Escriba enunciados por separado para vaciar el flujo de salida utilizando una función miembro y un manipulador de flujo.
q) Extraiga los siguientes valores: `124`, `18.376`, `'Z'`, `1000000`, y `"String"`.
r) Imprima el ajuste de precisión actual utilizando una función miembro.
s) Introduzca un valor entero en la variable `months` de `int` y un valor de punto flotante en la variable `percentageRate` de `float`.
t) Imprima `1.92`, `1.925`, y `1.9258` con `3` dígitos de precisión utilizando un manipulador.
u) Imprima el entero `100` en base octal, hexadecimal y decimal utilizando manipuladores de flujo.
v) Imprima el entero `100` en base decimal, octal y hexadecimal utilizando un solo manipulador de flujo para modificar la base.
w) Imprima `1234` justificado a la derecha en un campo de `10` dígitos.
x) Lea los caracteres al arreglo de caracteres `line` hasta que se encuentre con el carácter `'z'` hasta un límite de `20` caracteres (incluyendo el carácter de terminación null). No extraiga el carácter del delimitador del flujo.
y) Utilice las variables enteras `x` e `y` para especificar el ancho de campo y la precisión utilizada para mostrar el valor `87.4573` de tipo `double` y exhiban dicho valor.

21.4 Identifique el error en cada uno de los enunciados siguientes y explique cómo corregirlo.
a) `cout << "Value of x <= y is: " << x <= y;`
b) El siguiente enunciado debe de imprimir el valor entero de `'c'`
```
cout << 'c';
```
c) `cout << ""A string in quotes"";`

21.5 Muestre la salida para cada uno de los siguientes:
a)
```
cout << "12345\n";
cout.width(5);
cout.fill('*');
cout << 123 << '\n' << 123;
```
b) `cout << setw(10) << setfill('$') << 10000;`
c) `cout << setw(8) << setprecision(3) << 1024.987654;`

```
d) cout << setiosflags (ios::showbase) << oct << 99
   << '\n' << hex 99;
e) cout << 100000 << '\n'
   << setiosflags (ios::showpos) << 100000;
f) cout << setw(10) << setprecision(2) <<
   << setiosflags(ios::scientific) << 444.93738;
```

Respuestas a los ejercicios de autoevaluación

21.1 a) `friend`. b) `ios::left`, `ios::right`, y `ios::internal`. c) flujos. d) `setiosflags`, `resetiosflags`. e) `iostream.h`. f) `setf`, `unsetf`. g) `strstream.h`. h) `iomanip.h`. i) `fstream.h`. j) `endl`. k) `stdiostream.h`. l) `write`. m) `istream`. n) `cerr` o `clog`. o) `ostream`. p) `<<`. q) `cin`, `cout`, `cerr`, y `clog`. r) `>>`. s) `oct`, `hex`, `dec`. t) seis dígitos de precisión. u) `ios::showpos`.

21.2 a) Verdadero.

b) Falso. No se hace la homonimia de los operadores de extracción y de inserción de flujo para todos los tipos definidos por usuario. El programa de una clase debe proporcionar en forma específica las funciones de operador homónimas para hacer la homonimia de los operadores de flujo para uso con cada uno de los tipos definidos por usuario.

c) Falso. La función miembro de flujo `flags()` sin argumento, sólo devuelve el valor actual de la variable de estado `flags`.

d) Verdadero.

e) Verdadero.

f) Falso. Para hacer la homonimia del operador de inserción de flujo `<<`, la función de operador homónima debe tomar como argumentos una referencia `ostream` y una referencia a un tipo definido por usuario, y devolver una referencia `ostream`.

g) Verdadero. A menos de que `ios::skipws` esté desactivado.

h) Falso. Las características de entrada y de salida de C++ se proporcionan como parte de la biblioteca estándar de C++. El lenguaje de C++ no contiene capacidades para el procesamiento de entradas, salidas o de archivos.

i) Verdadero.

j) Verdadero.

k) Verdadero.

l) Falso. El flujo `cin` está conectado a la entrada estándar de la computadora, que por lo regular es el teclado.

m) Verdadero.

n) Verdadero.

o) Verdadero.

p) Falso. La función miembro `put` de `ostream` extrae su único argumento de carácter.

q) Falso. Los manipuladores de flujo `dec`, `oct` y `hex` establecen el estado de formato de salida para enteros a la base especificada, hasta que ésta sea otra vez específicamente modificada o el programa se termine.

r) Falso. Por omisión las direcciones de memoria se exhiben en formato hexadecimal. Para mostrar las direcciones como enteros `long`, la dirección deberá convertirse explícitamente (cast) a un valor `long`.

21.3 a)
```
cout << "Enter your name: ";
```
b)
```
cout.setf(ios::uppercase);
```
c)
```
cout << long(string);
```
d)
```
cout.setf(ios::scientific, ios::floatfield);
```
e)
```
cout << integerPtr;
```
f)
```
cout << setiosflags(ios::showbase);
```


g)
```
cout << *floatPtr;
```
h)
```
cout.fill('*');
    cout << setfill('*');
```
i)
```
cout.put('0').put('K');
```
j)
```
cin.peek();
```
k)
```
c = cin.get();
cin.get(c);
```
l)
```
cin.ignore(6);
```
m)
```
cin.read(line,50);
```
n)
```
cin.get(name, 10, '.');
    cin.getline(name, 10, '.');
```
o)
```
cout.write(line, cin.gcount());
```
p)
```
cout.flush();
    cout << flush;
```
q)
```
cout << 124 << 18.376 << 'Z' << 1000000 << "String";
```
r)
```
cout.precision();
```
s)
```
cin >> months >> percentageRate;
```
t)
```
cout << setprecision(3) << 1.92 << '\t'
    << 1.925 << '\t' << 1.9258;
```
u)
```
cout << oct << 100 << hex << 100 << dec << 100;
```
v)
```
cout << 100 << setbase(8) << 100 << setbase(16) << 100;
```
w)
```
cout << setw(10) << 1234;
```
x)
```
cin.get(line, 20, 'z');
```
y)
```
cout << setw(x) << setprecision(y) << 87.4573;
```

21.4 a) Error: es mayor la precedencia del operador `<<` que la del operador `<=`, lo que hace que se evalúe en forma incorrecta el enunciado y causará que se emita un error por el compilador.
Corrección: para corregir el enunciado, añada paréntesis encerrando la expresión `x <= y`. Este problema ocurrirá con cualquier expresión que utilice operadores de una precedencia menor que el operador `<<`, si la expresión no se encierra entre paréntesis.

b) Error: en C++, los caracteres no se tratan como pequeños enteros como es el caso en C.
Corrección: imprimir el valor numérico de un carácter en el conjunto de caracteres de la computadora, el carácter deberá ser convertido explícitamente (cast) a un valor entero como en el siguiente:

```
cout << int('c');
```

c) Error: los caracteres entre comillas no pueden ser impresos como una cadena, a menos de que se utilice una secuencia de escape.
Corrección: imprima la cadena en alguna de las siguientes formas:

```
cout << '"' << "A string in quotes" << '"';
    cout << "\"A string in quotes\"";
```

21.5 a)
```
12345
**123
123
```
b)
```
$$$$$10000
```
c)
```
1024.988
```
d)
```
0143
0x63
```
e)
```
100000
+100000
```
f)
```
4.45e+02
```

Ejercicios

21.6 Escriba un enunciado para cada uno de los siguientes:

a) Imprima el entero `40000` justificado a la izquierda en un campo de `15` dígitos.
b) Lea una cadena a la variable de arreglo de caracteres `state`.
c) Imprima `200` con y sin signo.
d) Imprima `100` en forma hexadecimal precedido por `0x`.
e) Lea caracteres al arreglo `s` hasta que se encuentre con el carácter `'p'` hasta un límite de 10 caracteres (incluyendo el carácter nulo de terminación). Extraiga el delimitador del flujo de entrada y descártelo.
f) Imprima `1.234` en un campo de `9` dígitos con ceros a la izquierda.
g) Lea un carácter de la forma `"characters"` de la entrada estándar. Almacene la cadena en el arreglo de caracteres `s`. Elimine las comillas del flujo de entrada. Lea un máximo de 50 caracteres (incluyendo el carácter nulo de terminación).

21.7 Escriba un programa para probar los valores enteros de entrada en formato decimal, octal y hexadecimal. Extraiga cada entero leído por el programa en los tres formatos. Pruebe el programa con los siguientes datos de entrada: 10, 010, 0x10.

21.8 Escriba un programa que imprima los valores de apuntador utilizando conversiones explícitas (cast) para todos los tipos de datos enteros. ¿Cuáles son los que imprimen valores raros? ¿Cuáles causan errores?

21.9 Escriba un programa para probar los resultados de imprimir el valor entero `12345` y el valor de punto flotante `1.2345` en campos de varios tamaños. ¿Qué ocurre cuando los valores se imprimen en campos que contienen menos dígitos que los valores?.

21.10 Escriba un programa que imprima el valor `100.453627` redondeado al siguiente dígito, décimo, centésimo, milésimo y diezmilésimo.

21.11 Escriba un programa que introduzca una cadena desde el teclado y que determine la longitud de dicha cadena. Imprima la cadena utilizando como ancho de campo el doble de su longitud.

21.12 Escriba un programa que convierta temperaturas enteras Fahrenheit desde `0` hasta `212` grados a temperaturas Celsius en punto flotante con `3` dígitos de precisión. Utilice la fórmula.

```
celsius = 5.0/9.0 * (fahrenheit - 32);
```

para efectuar el cálculo. La salida deberá ser impresa en dos columnas justificadas a la derecha, y las temperaturas Celsius deberán estar precedidas por un signo, tanto para valores positivos como negativos.

21.13 En algunos lenguajes de programación, las cadenas se introducen encerradas ya sea entre comillas sencillas o dobles. Escriba un programa que lea tres cadenas `suzy`, `"suzy"`, y `'suzy'`. Las comillas sencillas y dobles ¿son ignoradas o son leídas como parte de la cadena?

21.14 En la figura 21.30 los operadores de extracción y de inserción de flujo son homónimos para la entrada y salida de objetos `PhoneNumber`. Vuelva a escribir el operador de extracción de flujo para llevar a cabo las siguientes tareas de verificación de error a la entrada. Note que la función `operator>>` necesitará recodificarse totalmente.

a) Introduzca en un arreglo la totalidad del número telefónico. Pruebe que han sido introducidos todos los números telefónicos (debe haber un total de 14 caracteres para un número telefónico de la forma `(800) 555-1212`). Use la función miembro de flujo `clear` para definir el bit `ios::fail` para una entrada incorrecta.
b) En un número telefónico el código de área y la centralilla no empiezan con `0` o con `1`. Pruebe el primer dígito de las porciones del código de área y de centralilla del número telefónico, para estar seguro que ninguno empieza con `0` o con `1`. Utilice la función miembro de flujo `clear` para definir el bit `ios::fail` en caso de entrada incorrecta.

c) El dígito intermedio de un código de área es siempre `0` ó `1`. Pruebe el dígito intermedio de esta porción buscando un valor de `0` ó `1`. Utilice la función miembro de flujo `clear` para definir el bit `ios::fail` en caso de entrada incorrecta. Si ninguna de las anteriores operaciones resulta en la activación del bit `ios::fail` debido a entrada incorrecta, copie las tres porciones del número telefónico en los miembros `areaCode`, `exchange`, y `line` del objeto `PhoneNumber`. En el programa principal, si el bit `ios::fail` ha sido activado en la entrada, haga que el programa imprima un mensaje de error y termine antes de imprimir el número telefónico.

21.15 Escriba un programa que lleve a cabo cada uno de los siguientes:

a) Crear una clase definida por usuario de nombre `point` que contenga los miembros de dato enteros privados `xCoordinate` y `yCoordinate`, y declarar las funciones de operador homónimas de extracción y de inserción de flujo como `friend`s de la clase.
b) Defina las funciones de operador de inserción y de extracción de flujo. La función de operador de extracción de flujo debe determinar si los datos introducidos son válidos, de lo contrario, deberá activar `ios::failbit` para indicar entrada incorrecta. Una vez que haya ocurrido el error de entrada el operador de inserción de flujo no deberá ser capaz de mostrar el punto.
c) Escriba una función `main` que pruebe la entrada y la salida de la clase definida por usuario `point` utilizando los operadores de inserción y de extracción de flujo homónimos.

21.16 Escriba un programa que lleve a cabo cada uno de los siguientes:

a) Crear la clase definida por usuario `complex`, que contenga los miembros de dato enteros privados `real` e `imaginary`, y que declare las funciones de operador de extracción y de inserción de flujo homónimas como `friend`s de la clase.
b) Defina las funciones de operador o de extracción y de inserción de flujo. La función de operador de extracción de flujo deberá determinar si los datos introducidos son datos válidos, y de lo contrario, deberá activar `ios::failbit` para indicar entrada incorrecta. La entrada deberá ser de la forma

```
3 + 8i
```

Los valores pueden ser negativos o positivos, y es posible que alguno de los dos valores no sea proporcionado. Si uno de ellos no es proporcionado, el miembro de datos apropiado deberá ser definido como `0`. Si ha ocurrido un error de entrada el operador de inserción de flujo no debe ser capaz de mostrar el punto. El formato de salida debe ser idéntico al formato de entrada que se muestra arriba. Para los valores imaginarios negativos, deberá imprimirse un signo menos en vez de un signo más.
c) Escriba una función `main` que pruebe la entrada y la salida de la clase definida por usuario `complex` utilizando los operadores de inserción y de extracción de flujo homónimos.

21.17 Escriba un programa que utilice una estructura `for` para imprimir una tabla de valores ASCII para los caracteres en el conjunto de caracteres ASCII desde `33` hasta `126`. El programa deberá imprimir el valor decimal, el valor octal, el hexadecimal y el valor de carácter de cada uno de los caracteres. Utilice los manipuladores de flujo `dec`, `oct` y `hex` para imprimir los valores enteros.

21.18 Escriba un programa para demostrar que las funciones miembro istream de nombre `getline` y `get` de tres argumentos cada una de ellas termina la cadena de entrada con un carácter nulo de terminación de cadena. También demuestre que `get` deja el carácter delimitador en el flujo de entrada, en tanto que `getline` extrae el carácter delimitador y lo descarta . ¿Qué ocurre con los caracteres no leídos dentro del flujo?

21.19 Escriba un programa que cree el manipulador definido por usuario `skipwhite` para pasar por alto los caracteres de espacio en blanco a la izquierda en el flujo de entrada. El manipulador deberá utilizar la función `isspace` de la biblioteca `ctype.h` para probar si el carácter es un carácter de espacio en blanco. Cada carácter deberá ser introducido utilizando la función miembro `get` de `istream`. Cuando se encuentre

con un carácter distinto a un espacio en blanco, el manipulador **skipwhite** termina su trabajo colocando el carácter de regreso en el flujo de entrada y devolviendo una referencia **istream**.

Pruebe el manipulador definido por usuario creando una función **main** en el cual quede la bandera **ios::skipws** desactivada de tal forma que el operador de extracción de flujo no se salte en forma automática los espacios en blanco. A continuación pruebe el manipulador sobre el flujo de entrada introduciendo un carácter precedido por un espacio en blanco como entrada. Imprima el carácter que fue introducido a fin de confirmar que el carácter de espacio en blanco no fue introducido.

Bibliografía

(A193) Allison, C., "Code Capsules: A C++ Date Class, Parte I" *The C Users Journal,* Vol 11, No. 2, Febrero 1993, pp. 123-131.

(An92) Anderson, A. E., y W. J. Heinze, *C++ Programming and Fundamental Concepts,* Englewood Cliffs, NJ: Prentice Hall, 1992.

(Ba93) Bar-David, T., *Object-Oriented Design for C++*, Englewood Cliffs, NJ: Prentice Hall, 1993.

(Be93) Berard, E. V., Essays on Object-Oriented Software Engineering: Volumen I, Englewood Cliffs, NJ: Prentice Hall, 1993.

(Br91) Borland, *Borland C++ Programmer's Guide*, Parte No. 14MN-TCP04, Scotts Valley, CA: Borland International, Inc., 1991.

(Br91a) Borland, *Borland C++ Getting Started*, Parte No. 14MN-TCP02, Scotts Valley, CA: Borland International, Inc., 1991.

(Br91b) Borland, *Borland C++ 3.0 Programmers Guide*, Scotts Valley, CA: Borland International, Inc., 1991.

(By93) Byron, D., "The Case for Object Technology Standards," *CASE Trends,* Septiembre de 1993, pp. 22-26.

(Co93) Computerworld, "The CW Guide to Object-Oriented Programming" *Computerworld*, Junio 14, 1993, pp. 107-130.

(De90) Deitel, H. M., *Operating Systems*, Second Edition, Reading, MA: Addison-Wesley, 1990.

(El90) Ellis, M: A. y B. Stroustrup, *The Annotated C++ Reference Manual*, Reading, MA: Addison-Wesley, 1990.

(Fl93) Flaming, B., *Practical Data Structures in C++*, John Wiley & Sons, 1993.

(Ha93) Hagan, T., "C++ Class Libraries for GUIs," *Open Systems Today*, Febrero 15, 1993, pp. 54, 58.

(Ja93) Jacobson, I., "Is Object Technology Software's Industrial Platform?" *IEEE Software Magazine*, Vol. 10, No. 1, Enero de 1993, pp. 24-30.

(Ko93) Kozaczynski, W., y A. Kuntzmann-Combelles, "What It Takes to Make OO Work," *IEEE Software Magazine*, Vol.10, No.1, Enero 1993, pp. 20-23.

(Li91) Lippman, S. B., *C++ Primer* (Second Edition). Reading, MA: Addison-Wesley Publishing Company, 1991.

(Lo93) Lorenz, M., *Object-Oriented Software Development: A Practical Guide*, Englewood Cliffs, NJ: Prentice Hall, 1993.

(Lu92) Lucas, P. J., *The C++ Programmer's Handbook*, Englewood Cliffs, NJ: Prentice Hall, 1992.

(Ma93) Martin, J., *Principles of Object-Oriented Analysis and Design*, Englewood Cliffs, NJ: Prentice Hall, 1993.

(Me93) Matsche, J.J., "Object-oriented programming in Standard C," *Object Magazine*, Vol.2, No.5, Enero/Febrero 1993, pp. 71-74.

(Mi91) Microsoft, *Microsoft C/C++ Class Libraries Reference* (Versión 7.0), Redmond, WA: Microsoft Corporation, 1991.

(Mi91a) Microsoft, *Microsoft C/C++ C++ Language Reference* (Versión 7.0), Redmond, WA: Microsoft Corporation, 1991.

(Mi91b) Microsoft, *Microsoft C/C++ C++ Tutorial* (Versión 7.0), Redmond, WA: Microsoft Corporation, 1991.

(Pi93) Pittman, M., "Lessons Learned in Managing Object-Oriented Development," *IEEE Software Magazine*, Vol.10, No.1, Enero 1993, pp 43-53.

(Pr93) Prieto-Diaz, R., "Status Report: Software Reusability," *IEEE Software*, Vol.10, No.3, Mayo 1993, pp. 61-66.

(Ra92) Ranade, J., y S. Zamir, C++ *Primer for C Programmers,* New York, NY McGraw-Hill, Inc., 1992.

(Re91) Reed, D.R., "Moving from C to C++," *Object Magazine,* Vol.1, No.3, Septiembre/Octubre, 1991, pp. 46-60.

(Rl93) Rettig, M., G.Simmons, y J. Thompson, "Extended Objects," *Communications of the ACM,* Vol.36, No.8, Agosto 1993, pp. 19-24.

(Sa93) Saks, D., "Inheritance, Parte 2," *The C Users Journal,* Mayo 1993, pp.81-89.

(Sh91) Shiffman, H., "C++ Object-Oriented Extensions to C," *SunWorld,* Vol.4, No.5, Mayo de 1991, pp.63-70.

(Sk93) Skelly, C., "Pointer Power in C and C++, Parte 1," *The C Users Journal,* Vol.11, No.2, Febrero 1993, pp. 93-98.

(Sn93) Snyder, A., "The Essence of Objects: Concepts and Terms," *IEEE Software Magazine,* Vol.10, No.1, Enero 1993, pp. 31-42.

(St91) Stroustrup, B., *The C++ Programming Language* (Second Edition), Reading, MA: Addison-Wesley Series in Computer Science, 1991.

(St93) Stroustrup, B., "Why Consider Language Extensions?": Maintaining a Delicate Balance," *C++ Report,* Septiembre 1993, pp. 44-51.

(Vo93) Voss, G., "Objects and Messages," *Windows Tech Journal,* Febrero de 1993, pp. 15-16.

(Wi93) Wiebel, M., y S. Halladay, "Using OOP Techniques Instead of *switch* in C++," Vol.10, No.10, *The C Users Journal,* Octubre de 1992, pp. 105-106,

(Wl93) Wilde, N., P. Matthews, y R Huitt, "Maintaining Object-Oriented Software" *IEEE Software Magazine,* Vol.10, No.1, Enero 1993, pp. 75-80.

(Wt93) Wilt, N., "Templates in C++," *The C Users Journal,* Mayo de 1993, pp. 33-51.

Apéndice A *

Sintaxis de C

En la notación sintáctica utilizada, las categorías sintácticas (no terminales) se indican en *cursivas* y las palabras literales y los miembros de conjuntos de caracteres (terminales) en **negrillas**. Dos puntos después de una terminal introducen su definición. Las definiciones alternas son enlistadas en líneas por separado, excepto cuando estén antecedidas por las palabras "una de". Un símbolo opcional es presentado mediante el subíndice "opt", de tal forma que

{*expresión*$_{opt}$ }

indica una expresión opcional encerrada entre llaves.

Resumen de sintaxis de lenguaje

A.1 Gramática lexicográfica

A.1.1 Componentes léxicos (tokens)

Componente léxico:
- *palabra reservada*
- *identificador*
- *constante*
- *cadena literal*
- *operador*
- *puntuaciones*

componente léxico de preprocesamiento:
- *nombre de encabezado*
- *identificador*
- *número de preprocesador*
- *constante de carácter*
- *cadena literal*
- *operador*

* Certificación de autorización: Este material ha sido condensado y adaptado partiendo de American National Standard for International —Systems Programming Language— C, ANSI/ISO 9899:1990. Se pueden adquirir copias de esta norma de American National Standards Institute en 11 West 42nd. Street, New York, NY 10036.

puntuaciones
cada carácter distinto a un espacio en blanco que no pueda ser uno de los arriba citados

A.1.2 Palabras reservadas

palabra reservada: una de

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

A.1.3 Identificadores

identificador:
- *no dígito*
- *identificador no dígito*
- *identificador dígito*

no dígito: uno de:

`_ a b c d e f g h i j k l m`
`n o p q r s t u v w x y z`
`A B C D E F G H I J K L M`
`N O P Q R S T U V W X Y Z`

dígito: uno de
`0 1 2 3 4 5 6 7 8 9`

A.1.4 Constantes

constante:
- *constante flotante*
- *constante entero*
- *constante de enumeración*
- *constante de carácter*

constante flotante:
- *constante fraccionaria sufijo*$_{opt}$ *flotante parte*$_{opt}$ *exponencial*
- *secuencia de dígitos sufijo*$_{opt}$ *flotante parte exponencial*

constante fraccionaria:
- *secuencia*$_{opt}$ *de dígitos* . *secuencia de dígitos*
- *secuencia de dígitos*.

parte exponencial:
- *signo*$_{opt}$ **e** *secuencia de dígitos*
- *signo*$_{opt}$ **E** *secuencia*$_{opt}$ *de dígitos*

signo: uno de
`+ -`

secuencia de dígitos
    dígito
    secuencia de dígitos dígito

sufijo flotante: uno de
    `f l F L`

constante entera
    constante decimal sufijo$_{opt}$ *entero*
    constante octal sufijo$_{opt}$ *entero*
    constante hexadecimal sufijo$_{opt}$ *entero*

constante decimal:
    dígito no cero
    constante decimal dígito

constante octal:
    `0`
    constante octal dígito octal

constante hexadecimal:
    dígito *hexadecimal* `0x`
    dígito *hexadecimal* `0X`
    constante hexadecimal dígito hexadecimal

dígito no cero: uno de
    `1 2 3 4 5 6 7 8 9`

dígito octal: uno de
    `0 1 2 3 4 5 6 7`

dígito hexadecimal: uno de
    `0 1 2 3 4 5 6 7 8 9`
    `a b c d e f`
    `A B C D E F`

sufijo entero:
    sufijo unsignado sufijo$_{opt}$ *long*
    sufijo long sufijo$_{opt}$ *unsignado*

sufijo unsignado: uno de
    `u U`

sufijo long: uno de
    `l L`

constante de enumeración:
    identificador

constante de carácter:
    'secuencia c-char'
    'secuencia c-char' `L`

secuencia c-char:
    c-char
    secuencia c-char c-char

c-char:
    cualquier miembro del conjunto de caracteres fuente, a excepción de la comilla sencilla `'`, la diagonal invertida \, o el carácter de nueva línea
    secuencia de escape

secuencia de escape:
    secuencia de escape simple
    secuencia de escape octal
    *secuencia de escape hexadecim*al

secuencia de escape simple: una de
    `\'  \"  \?  \\`
    `\a  \b  \f  \n  \r  \t  \v`

secuencia de escape octal:
    dígito octal \
    dígito octal dígito octal \
    dígito octal dígito octal dígito octal \

secuencia de escape hexadecimal:
    dígito hexadecimal `\x`
    secuencia de escape hexadecimal dígito hexadecimal

A.1.5 Cadenas literales

cadena literal:
    "secuencia$_{opt}$ *s-char"*
    "secuencia$_{opt}$ *s-char"* `L`

secuencia s-char:
    s-char
    secuencia s-char s-char

s-char:
    cualquier miembro del conjunto de caracteres fuente, a excepción de las dobles comillas `"`, la diagonal invertida \ o el carácter de nueva línea
    secuencia de escape

A.1.6 Operadores

operador: uno de

```
[ ]   ( )   .     ->
++    --    &     *     +     -     ~     !     sizeof
/     %     <<    >>    <     >     <=    >=    ==    !=    ^     |     &&    ||
?     :
=     *=    /=    %=    +=    -=    <<=   >>=   &=    ^=    |=
,     #     ##
```

A.1.7 Puntuaciones

puntuaciones: uno de

```
[ ]   ( )   { }   *   ,   :   =     ...   #
```

A.1.8 Nombres de encabezados

nombre de encabezado:
<h-char-sequence>
"secuencia q-char"

secuencia h-char:
h-char
secuencia h-char h-char

h-char:
cualquier miembro del conjunto de caracteres fuente, a excepción del carácter de nueva línea, y de >

secuencia q-char:
q-char
secuencia q-char q-char

q-char:
cualquier miembro del conjunto de caracteres fuente, a excepción del carácter de nueva línea, y de "

A.1.9 Números de preprocesador

número de preprocesador:
dígito
dígito .
dígito de número de preprocesador
no dígito de número de preprocesador
signo e *de número de preprocesador*
signo E *de número de preprocesador*
. de número de preprocesador

A.2. Gramática estructural de frases

A.2.1 Expresiones

expresión primaria:
identificador
constante
cadena literal
(*expresión*)

expresión posfija:
expresión primaria
expresión posfija [*expresión*]
expresión posfija (*lista*$_{opt}$ *de expresión de argumentos*)
expresión posfija identificador .
expresión posfija identificador ->
expresión posfija ++
expresión posfija --

lista de expresión de argumentos:
expresión de asignación
lista de expresión de argumentos , *expresión de asignación*

expresión unaria:
expresión posfija
expresión unaria ++
expresión unaria --
expresión de operador unario de conversión explícita (cast)
expresión unaria `sizeof`
`sizeof` (*nombre de tipo*)

operador unario: uno de
`&` `*` `+` `-` `~` `!`

expresión de conversión explícita (cast):
expresión unaria
expresión de conversión explícita (cast) (*nombre de tipo*)

expresión multiplicativa:
expresión de conversión explícita (cast)
expresión multiplicativa expresión de conversión explícita (cast) *
expresión multiplicativa expresión de conversión explícita (cast) /
expresión multiplicativa expresión de conversión explícita (cast) %

expresión aditiva:
expresión multiplicativa
expresión aditiva expresión multiplicativa +
expresión aditiva expresión multiplicativa -

expresión de desplazamiento:
expresión aditiva
expresión de desplazamiento expresión aditiva <<
expresión de desplazamiento expresión aditiva >>

expresión relacional:
expresión de desplazamiento
expresión de desplazamiento expresión relacional <
expresión de desplazamiento expresión relacional >

expresión relacional expresión de desplazamiento <=
expresión relacional expresión de desplazamiento >=

expresión de igualdad:
expresión relacional
expresión de igualdad expresión relacional ==
expresión de igualdad expresión relacional !=

expresión AND:
expresión de igualdad
expresión de igualdad &

expresión OR exclusivo:
expresión AND
expresión OR exclusivo expresión AND ^

expresión OR inclusivo:
expresión OR exclusivo
expresión OR inclusivo expresión OR exclusivo |

expresión AND lógica:
expresión OR inclusivo
expresión AND lógica expresión OR inclusivo &&

expresión OR lógica:
expresión AND lógica
expresión OR lógica expresión AND lógica ||

expresión condicional:
expresión OR lógica
expresión OR lógica expresión ? *expresión condicional* :

expresión de asignación:
expresión condicional
expresión de asignación unaria expresión de asignación operador

operador de asignación: uno de
= *= /= %= += -= <<= >>= &= ^= |=

expresión:
expresión condicional
expresión expresión condicional ,

expresión constante:
expresión condicional

A.2.2 Declaraciones

declaración:
declaración de especificadores de inicialización $lista_{opt}$;

declaración de especificadores:
declaración de especificadores de clase de almacenamiento $especificadores_{opt}$
declaración de especificadores de tipo $especificadores_{opt}$
declaración de calificadores de tipo de $especificadores_{opt}$

lista de declarador de inicialización:
declarador de inicialización
lista de declarador de inicialización declarador de inicialización ,

declarador Init:
declarador
declarador = *inicializador*

especificador de clase de almacenamiento:
typedef
extern
static
auto
register

especificador de tipo:
void
char
short
int
long
float
double
signed
unsigned
especificador de estructuras o uniones
especificador de enumeraciones
nombre typedef

especificador de estructura o unión:
identificador de estructura o unión$_{opt}$ {*lista de declaración de estructura*}
identificador de estructura o unión

estructura o unión:
struct
union

lista de declaración de estructura:
declaración de estructura
lista de declaración de estructura declaración de estructura

declaración de estructura:
lista de calificadores de especificación lista de declaración de estructura ;

lista de calificadores de especificación:
especificador de especificadores de tipo lista de calificadores$_{opt}$
especificador de calificadores de tipo lista de calificadores$_{opt}$

lista de declarador de estructura:
declarador de estructura
lista de declarador de estructura declarador de estructura ,

declarador de estructura:
declarador
declarador$_{opt}$: *expresión constante*

especificador de enumeraciones:
identificador$_{opt}$ **enum** { *lista de enumerador* }
identificador **enum**

lista de enumerador:
enumerador
lista de enumerador enumerador ,

enumerador:
constante de enumeración
constante de enumeración expresión constante =

calificador de tipo:
const
volatile

declarador:
apuntador$_{opt}$ declarador directo

declarador directo:
identificador
(*declarador*)
declarador directo [*expresión$_{opt}$constante*]
declarador directo (*lista de parámetros de tipo*)
declarador directo (*lista$_{opt}$ de identificadores*)

apuntador:
lista$_{opt}$ *de calificadores de tipo* *
*lista$_{opt}$ de calificadores de tipo * apuntador*

lista de calificadores de tipo
calificadores de tipo
lista de calificadores de tipo calificador de tipo

lista de tipo de parámetros:
lista de parámetros
lista de parámetros , ...

lista de parámetros:
declaración de parámetros
lista de parámetros , *declaración de parámetros*

declaración de parámetros:
especificador de declaración declarador
especificador de declaración declarador abstracto$_{opt}$

lista de identificadores:
identificador
lista de identificadores , *identificador*

nombre de tipo:
lista de calificador de especificador declarador$_{opt}$ abstracto

declarador abstracto:
apuntador
apuntador$_{opt}$ declarador abstracto directo

declarador abstracto directo:
(declarador abstracto)
declarador abstracto directo$_{opt}$ [*expresión$_{opt}$constante*]
declarador abstracto directo$_{opt}$ (*lista de tipo de parámetros$_{opt}$*)

nombre typedef:
identificador

inicializador:
expresión de asignación
{ *lista de inicializador* }
{ *lista de inicializador* , }

lista de inicializador:
inicializador
lista de inicializador , *inicializador*

A.2.3 Enunciados

enunciado:
enunciado etiquetado
enunciado compuesto
enunciado de expresión
enunciado de selección
enunciado de iteración
enunciado de salto

enunciado etiquetado:
identificador : *enunciado*
expresión constante **case** *expresión* :
enunciado: **default**

enunciado compuesto:
`{` *lista*$_{opt}$ *de declaración lista*$_{opt}$ *de enunciado* `}`

lista de declaración:
declaración
declaración lista de declaración

lista de enunciado:
enunciado
enunciado lista de enunciado

enunciado de expresión:
expresión$_{opt}$ `;`

enunciado de selección:
enunciado `(` *expresión* `)` `if`
enunciado `(` *expresión* `)` `if` *enunciado* `else`
enunciado `(` *expresión* `)` `switch`

enunciado de iteración:
enunciado `(` *expresión* `)` `while`
enunciado `do` `(` *expresión* `)` `while` `;`
enunciado `(`*expresión*$_{opt}$ `;` *expresión*$_{opt}$ `;` *expresión*$_{opt}$`)` `for`

enunciado de salto:
identificador `goto` `;`
`continue` `;`
`break` `;`
expresión$_{opt}$ `return` `;`

A.2.4 Definiciones externas

unidad de traducción:
declaración externa
unidad de traducción declaración externa

declaración externa:
definición de función
declaración

definición de función:
declarador de especificadores$_{opt}$ *de declaración enunciado compuesto de lista*$_{opt}$ *de declaración*

A.3 Directrices de preprocesador

archivo de preprocesador:
grupo$_{opt}$

grupo:
parte de grupo
parte de grupo grupo

parte de grupo:
componentes léxicos$_{opt}$ *de preprocesador nueva línea*
sección if
línea de control

sección if:
grupo if grupos$_{opt}$ *elif grupo*$_{opt}$ *línea endif*

grupo if:
`#` `if` *expresión constante grupo*$_{opt}$ *nueva línea*
`#` `ifdef` *identificador grupo*$_{opt}$ *nueva línea*
`#` `ifndef` *identificador grupo*$_{opt}$ *nueva línea*

grupos elif:
grupos elif
grupos elif grupo elif

grupo elif:
`#` `elif` *expresión constante grupo*$_{opt}$ *nueva línea*

grupo else:
`#` `else` *grupo*$_{opt}$ *nueva línea*

línea endif:
`#` `endif` *nueva línea*

línea de control:
`#` `include` *componentes léxicos nueva línea*
`#` `define` *lista de remplazo identificador nueva línea*
`#` `define` *identificador paréntesis izquierdo (lparen) lista*$_{opt}$ *identificador* `)` *lista de remplazo nueva línea*
`#` `undef` *identificador nueva línea*
`#` `line` *componentes léxicos nueva línea*
`#` `error` *componentes léxicos*$_{opt}$ *nueva línea*
`#` `pragma` *componentes léxicos*$_{opt}$ *nueva línea*
`#` *nueva línea*

lparen:
el carácter de paréntesis izquierdo sin espacio en blanco previo

lista de remplazo:
componentes léxicos$_{opt}$

componentes léxicos:
componente léxico de preprocesador
componentes léxicos de preprocesador componente léxico

nueva línea:
el carácter de nueva línea

UNIVERSIDAD DE LA SALLE
FACULTAD DE INGENIERIA
DEPARTAMENTO DE DOCUMENTACION Y BIBLIOTECA
CENTRO DE INFORMACION - BIBLIOTECA

Apéndice B*

Biblioteca estándar

B.1 Errores <errno.h>

EDOM

ERANGE

Estas se expanden a expresiones constantes integrales con valores precisos no cero, utilizables para uso en directrices de preprocesador `#if`.

`errno`

Un valor de tipo `int` que es definido por varias funciones de biblioteca como un número positivo de error. Al arranque del programa el valor `errno` es cero, pero jamás es definido como cero por ninguna función de biblioteca. Un programa que utilice `errno` para verificación de errores debe definirlo a cero antes de una llamada a una función de biblioteca, y leerlo antes de una subsiguiente llamada a una función de biblioteca. Una función de biblioteca puede guardar el valor de `errno` al entrar y después definirlo como cero, siempre y cuando el valor original sea restaurado, si justo antes del regreso, el valor de `errno` siga siendo cero. El valor de `errno` puede ser definido por una llamada de función de biblioteca como un valor no cero, exista o no exista error, siempre que el uso de `errno` no haya sido documentado en la descripción de función en el estándar.

B.2 Definiciones comunes <stddef.h>

NULL

Una constante de apuntador nula definida por la puesta en práctica.

`offsetof` (*tipo*, *designador de miembro*)

Se expande a una expresión constante íntegra de tipo `size_t`, el valor de la cual es el desplazamiento en bytes al miembro de estructura (especificado por el *designador de miembro*) a partir del principio de su tipo de estructura (designado por *tipo*). El *designador de miembro* debe ser de tal forma que, dado

```
static tipo t;
```

* Reconocimiento de autorización: Este material ha sido condensado y adaptado a partir del American National Standard for Information Systems —Programming Language— C, ANSI/ISO 9899:1990. Se pueden adquirir copias de esta norma de la American National Standard Institute en 11 West 42nd Street, New York, NY 10036.

entonces, la expresión `&` `(t.` *designador de miembro*) se evalúa a una constante de dirección. (Si el miembro especificado es un campo de bit, el comportamiento queda indefinido).

`ptrdiff_t`

El tipo entero con signo del resultado de la sustracción de dos apuntadores.

`size_t`

El tipo entero sin signo del resultado del operador `sizeof`.

`wchar_t`

Un tipo entero cuyo rango de valores puede representar códigos diferentes para todos los miembros del conjunto extendido de caracteres especificado entre los escenarios soportados; el carácter nulo tendrá el valor de código cero y cada miembro del conjunto básico de caracteres tendrá un valor de código igual a su valor cuando se use como un carácter único en una constante de caracteres íntegra.

B.3 Diagnósticos <assert.h>

`void assert(int expression);`

La macro `assert` efectúa diagnósticos dentro de programas. Cuando se ejecuta, si `expression` es falsa, la macro `assert` escribe información relativa a la llamada particular que falló (incluyendo el texto del argumento, el nombre del archivo fuente, y el número de línea fuente —éstos últimos son respectivamente los valores de los macros de preprocesador `__FILE__` y `__LINE__`) en el archivo de error estándar en formato definido por la puesta en práctica. El mensaje escrito pudiera aparecer de la forma

```
Assertion failed: expression, file xyz, line nnn
```

La macro `assert` a continuación llama a la función `abort`. Si la directiva de preprocesador

```
#define NDEBUG
```

aparece en el archivo fuente donde `assert.h` esté incluido, cualquier verificación sobre el archivo será ignorada.

B.4 Manejo de caracteres <ctype.h>

Las funciones en esta sección devuelven no cero (verdadero) si, y sólo si, el valor del argumento c está conforme con el incluido en la descripción de la función.

`int isalnum(int c);`

Prueba buscando cualquier carácter para el cual `isalpha` o `isdigit` es verdadero.

`int isalpha(int c);`

Prueba buscando cualquier carácter para el cual `isupper` o `islower` es verdadero.

`int iscntrl(int c),`

Prueba la existencia de cualquier carácter de control.

`int isdigit(int c);`

Prueba buscando cualquier carácter de dígito decimal.

`int isgraph(int c);`

Busca cualquier carácter de impresión, excepto el espacio (`' '`).

`int islower(int c);`

Busca cualquier carácter que sea una letra minúscula.

`int isprint(int c);`

Busca cualquier carácter de impresión incluyendo el espacio (`' '`).

`int ispunct(int c);`

Prueba buscando cualquier carácter de impresión que no sea ni espacio (`' '`) ni ningún carácter para el cual **isalnum** resulte verdadero.

int isspace(int c);

Prueba buscando cualquier carácter que sea un carácter estándar de espacio en blanco. Los caracteres estándar de espacio en blanco son: espacio (`' '`), alimentación de forma (`'\f'`), nueva línea (`'\n'`), retorno de carro (`'\r'`), tabulador horizontal (`'\t'`) y tabulador vertical (`'\v'`).

in isupper(int c);

Busca cualquier carácter que esté en mayúsculas.

in isxdigit(int c);

Busca cualquier carácter en dígitos hexadecimales.

int tolower(int c);

Convierte una letra mayúscula en la letra minúscula correspondiente. Si el argumento es un carácter para el cual **isupper** resulta verdadero y existe un carácter correspondiente para el cual **islower** resulta verdadero, la función **tolower** devuelve el carácter correspondiente; de lo contrario, el argumento se conserva sin modificación.

int toupper(int c);

Convierte una letra minúscula a la letra mayúscula correspondiente. Si el argumento a un carácter para el cual **islower** resulta verdadero y existe un carácter correspondiente para el cual **isupper** es verdadero, la función **toupper** devuelve el carácter correspondiente; de lo contrario, el argumento se regresa sin modificación.

B.5 Localización <locale.h>

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME

Estas se expanden a expresiones constantes íntegras con valores precisos, utilizables como primer argumento de la función **setlocale**.

NULL

Una constante de apuntador nula definida por la puesta en práctica.

struct lconv

Contiene miembros relacionados con el formato de valores numéricos. La estructura deberá contener por lo menos los siguientes miembros en cualquier orden. En el escenario "C" los miembros tendrán los valores especificados en los comentarios.

```
char *decimal_point;        /* "." */
char *thousands-sep;        /* "" */
char *grouping;             /* "" */
char *int_curr_symbol;      /* "" */
char *currency_symbol;      /* "" */
char *mon_decimal_point;    /* "" */
char *mon_thousands_sep;    /* "" */
char *mon_grouping;         /* "" */
char *positive_sign;        /* "" */
char *negative_sign;        /* "" */
char int_frac_digits;       /* CHAR_MAX */
char frac_digits;           /* CHAR_MAX */
```



```
char p_cs_precedes;         /* CHAR_MAX */
char p_sep_by_space;        /* CHAR_MAX */
char n_cs_precedes;         /* CHAR_MAX */
char n_sep_by_space;        /* CHAR_MAX */
char p_sign_posn;           /* CHAR_MAX */
char n_sign_posn;           /* CHAR_MAX */
```

char *setlocale(int category, const char *locale);

La función **setlocale** selecciona la porción apropiada del escenario del programa, tal y como se especifica por los argumentos **category** y **locale**. La función **setlocale** puede ser utilizada para modificar o consultar el escenario actual total del programa o partes del mismo. El valor **LC_ALL** para **category** nombra todo el escenario del programa: los otros valores para **category** nombran sólo una porción del escenario del programa. **LC_COLLATE** afecta el comportamiento de las funciones **strcoll** y **strxfrm**. **LC_CTYPE** afecta el comportamiento de las funciones de manejo de caracteres y de las funciones multibyte. **LC_MONETARY** afecta la información de formato de moneda devuelta por la función **localeconv**. **LC_NUMERIC** afecta el carácter de punto decimal para las funciones de entrada/salida con formato, para las funciones de conversión de cadenas, y para la información de formato no monetario devuelta por **localeconv**. **LC_TIME** afecta el comportamiento de **strftime**.

Un valor de **"C"** para el **locale** especifica el entorno mínimo para traducción en C; un valor de **""** para **locale** especifica el entorno nativo definido por la puesta en práctica. Se pueden pasar a **setlocale** otras cadenas definidas por la puesta en práctica. Al arranque del programa, se ejecuta el equivalente de

```
setlocale(LC_ALL, "C");
```

Si un apuntador a una cadena se da para **locale** y dicha selección puede ser aceptada, la función **setlocale** devuelve un apuntador a la cadena asociada con la **category** especificada para el nuevo escenario. Si la selección no puede ser aceptada, la función **setlocale** devuelve un apuntador nulo y el escenario del programa no es modificado.

Un apuntador nulo para **locale** hace que el función **setlocale** devuelva un apuntador a la cadena asociada con la **category** correspondiente al escenario actual del programa; el escenario del programa no se modifica.

El apuntador a la cadena devuelta por la función **setlocale** es tal que una llamada subsecuente a ese valor de cadena y a su categoría asociada restaurará dicha parte de la escenario del programa. La cadena a la cual se señala será modificada por el programa, pero podría ser sobrescrita por una llamada subsecuente a la función **setlocale**.

struct lconv *localeconv(void);

La función **localeconv** define los componentes de un objeto con tipo **struct lconv** con valores apropiados para el formato de cantidades numéricas (de moneda y otras) de acuerdo con las reglas del escenario actual.

Los miembros de la estructura con el tipo **char *** son apuntadores a cadenas, cualquiera de los cuales (a excepción de **decimal_point**) puede señalar a **""**, para indicar que el valor no está disponible en el escenario actual o tiene una longitud cero. Los miembros con tipo **char** son números no negativos, cualquiera de los cuales puede ser **CHAR_MAX** para indicar que el valor no está disponible en el escenario actual.

Los miembros incluyen lo siguiente:

char *decimal_point

El carácter de punto decimal utilizado para darle formato a cantidades no monetarias.

char *thousands_sep

El carácter utilizado para separar grupos de dígitos antes del carácter de punto decimal en cantidades no monetarias con formato.

char *grouping
Una cadena cuyos elementos indican el tamaño de cada grupo de dígitos en cantidades no monetarias con formato.

char *int_curr_symbol
El símbolo de moneda internacional aplicable al escenario actual. Los tres primeros caracteres contienen el símbolo de moneda internacional alfabético de acuerdo con lo especificado en ISO 4217:1987. El cuarto carácter (que precede al carácter nulo) es el carácter utilizado para separar el símbolo internacional de moneda de la cantidad monetaria.

char *currency_symbol
El símbolo local de moneda aplicable al escenario actual.

char *mon_decimal_point
El punto decimal utilizado para darle formato a cantidades monetarias.

char *mon_thousands_sep
El separador para grupos de dígitos antes del punto decimal en cantidades monetarias con formato.

char *mon_grouping
Una cadena cuyos elementos indican el tamaño de cada grupo de dígitos en cantidades monetarias con formato.

char *positive_sign
Una cadena utilizada para indicar una cantidad monetaria con formato con valores no negativos.

char *negative_sign
La cadena que se utiliza para indicar una cantidad monetaria con formato de valor negativo.

char int_frac_digits
El número de dígitos fraccionarios (aquellos que aparecen después del punto decimal) al mostrarse en una cantidad monetaria internacional con formato.

char frac_digits
El número de dígitos fraccionarios (los que aparecen después del punto decimal) a mostrarse en una cantidad monetaria con formato.

char p_cs_precedes
Se define como 1 ó como 0 el **currency_symbol** respectivamente antecederá o seguirá al valor en una cantidad monetaria no negativa con formato.

char p_sep_by_space
Definido como 1 ó como 0 el **currency_symbol** respectivamente está o no está separado mediante un espacio de la cantidad monetaria no negativa con formato.

char n_cs_precedes
Definido a 1 ó como 0 el **currency_symbol** respectivamente antecede o sigue al valor en una cantidad monetaria negativa con formato.

char n_sep_by_space
Definido a 1 ó como 0 el **currency_symbol** respectivamente está o no está separado mediante un espacio del valor en una cantidad monetaria negativa con formato.

char p_sign_posn
Definido a un valor indicado en la posición del **positive_sign** para una cantidad monetaria no negativa con formato.

char n_sign_posn
Definido a un valor indicado en la posición del **negative_sign** en cantidad monetaria negativa con formato.

Los elementos de **grouping** y de **mon_grouping** se interpretan de acuerdo con lo siguiente:

CHAR_MAX No se ejecutará más agrupamiento

0 El elemento precedente debe ser utilizado de forma repetida para el resto de los dígitos.

other El valor entero es el número de dígitos que comprenden el grupo actual. Se examinará el siguiente elemento para determinar el tamaño del siguiente grupo de dígitos antes del grupo actual.

Los valores de **p_sign_posn** y de **n_sign_posn** se interpretan de acuerdo con lo siguiente.

0 Paréntesis encerrando la cantidad y **currency_symbol**.

1 La cadena de signos antecede a la cantidad y **currency_symbol**.

2 La cadena de signos sigue a la cantidad y **currency_symbol**.

3 La cadena de signos precede de inmediato antes de **currency_symbol**.

4 La cadena de signos sigue de inmediato a **currency_symbol**.

La función **localeconv** devuelve un apuntador al objeto rellenado. La estructura a la cual se señala mediante el valor de regreso no deberá ser modificado por el programa, pero puede ser sobreescrito mediante una llamada subsecuente a la función **localeconv**. En adición, llamadas a la función **setlocale** con las categorías **LC_ALL, LC_MONETARY,** o bien **LC_NUMERIC** pueden sobreescribir el contenido de la estructura.

B.6 Matemáticas <math.h>

HUGE_VAL
Una constante simbólica que representa una expresión positiva **double**.

double acos(double x);
Calcula el valor principal del arco cuyo coseno es **x**. Para argumentos que no estén en el rango [-1, +1] ocurrirá un error de dominio. La función **acos** devuelve el arco del coseno en el rango de [0, π] radianes.

double asin(double x);
Calcula el valor principal del arco cuyo seno es **x**. Para argumentos que no estén en el rango [-1, +1] ocurrirá un error de dominio. La función **asin** devuelve el seno arco en el rango de [-π/2, +π/2] radianes.

double atan(double x);
Calcula el valor principal del arco cuya tangente es **x**. La función **atan** devuelve el arco tangente en el rango de [-π/2, +π/2] radianes.

double atan2(double y, double x);
La función **atan2** calcula el valor principal del arco tangente **y/x**, utilizando los signos de ambos argumentos para determinar el cuadrante del valor devuelto. Puede ocurrir un error de dominio si ambos argumentos son cero. La función **atan2** devuelve el arco tangente de **y/x**, en el rango [-π, +π] radianes.

double cos(double x);
Calcula el coseno de **x** (medido en radianes).

double sin(double x);
Calcula el seno de **x** (medido en radianes).

`double tan(double x);`
Devuelve la tangente de `x` (medida en radianes).

`double cosh(double x);`
Calcula el coseno hiperbólico de `x`. Puede ocurrir un error de rango si la magnitud de `x` es demasiado grande.

`double sinh(double x);`
Calcula el seno hiperbólico de `x`. Ocurre un error de rango si la magnitud de `x` es demasiado grande.

`double tanh(double x);`
La función `tanh` calcula la tangente hiperbólica de `x`.

`double exp(double x);`
Calcula la función exponencial de `x`. Ocurre un error de rango si la magnitud de `x` es muy extensa.

`double frexp(double value, int *exp);`
Divide el número de punto flotante en una fracción normalizada y una potencia entera de 2. Almacena el entero en el objeto `int` apuntado por `exp`. La función `frexp` devuelve el valor `x`, tal que `x` es un `double` con magnitud en el intervalo [1/2, 1] o cero, y `value` es igual a `x` multiplicado por 2 elevado a la potencia `*exp`. Si `value` es igual a cero, ambas partes del resultado son cero.

`double ldexp(double x, int exp);`
Multiplica un número de punto flotante por una potencia entera de 2. Puede ocurrir un error de rango. La función `ldexp` devuelve el valor de `x` multiplicado por 2 elevado a la potencia `exp`.

`double log(double x);`
Calcula el logaritmo natural de `x`. Ocurre un error de dominio si el argumento es negativo. Pudiera ocurrir un error de rango si el argumento es cero.

`double log10(double x);`
Calcula el logaritmo en base diez de `x`. Ocurre un error de dominio si el argumento es negativo. Pudiera ocurrir un error de rango si el argumento es cero.

`double modf(double value, double *iptr);`
Divide el argumento `value` en partes enteras y fraccionarias, cada una de las cuales tiene el mismo signo que el argumento. Almacena en la parte entera como un `double` en el objeto al cual apunta `iptr`. La función `modf` devuelve la parte fraccionaria signada de `value`.

`double pow(double x, double y);`
Calcula `x` elevado a la potencia `y`. Ocurre un error de dominio si `x` es negativo e `y` no es un valor entero. Ocurre un error de dominio si el resultado no puede ser representado cuando `x` es cero e `y` es menor o igual a cero. Puede ocurrir un error de rango.

`double sqrt (double x);`
Calcula la raíz cuadrada no negativa de `x`. Ocurre un error de dominio si el argumento es negativo.

`double ceil(double x);`
Calcula el valor integral más pequeño no menor que `x`.

`double fabs(double x);`
Calcula el valor absoluto del número de punto flotante `x`.

`double floor(double x);`
Calcula el valor integral más grande no mayor que `x`.

`double fmod(double x, double y);`
Calcula el residuo en punto flotante de `x` dividido entre `y`.

B.7 Saltos no locales `<setjmp.h>`

`jmp_buf`
Un tipo de arreglo adecuado para contener la información necesaria para restaurar un entorno llamador.

`int setjmp(jmp_buf env);`
Guarda su entorno llamador en el argumento `jmp_buf` para uso posterior por la función `longjmp`.

Si el regreso es debido a una invocación directa, la macro `setjmp` devuelve el valor cero. Si el regreso es debido a una llamada de la función `longjmp`, la macro `setjmp` devuelve un valor no cero.

Una invocación a la macro `setjmp` debe de aparecer sólo en alguno de los siguientes contextos:

- en la totalidad de la expresión de control de un enunciado de selección o de iteración;
- un operando de un operador relacional o de igualdad siendo el otro operando una expresión constante entera, con la expresión resultante siendo la expresión de control total de un enunciado de selección o de iteración;
- el operando del operador unario `!` con la expresión resultante siendo la expresión de control total de un enunciado de selección o de iteración; o bien
- la expresión total de un enunciado de expresión.

`void longjmp(jmp_buf env, int val);`
Devuelve el entorno guardado por la invocación más reciente de la macro `setjmp` en la misma invocación del programa, con el argumento correspondiente `jmp_buf`. Si dicha invocación no ha ocurrido o si en el ínterin la función que contiene la invocación de la macro `setjmp` ha terminado su ejecución, el comportamiento queda indefinido.

Todos los objetos accesibles tienen valores correspondientes al momento en que fue llamado `longjmp`, excepto que los valores de objeto de persistencia automática, que son locales a la función que contiene la invocación de la macro `setjmp` correspondiente que no son del tipo volátil y que han sido modificados entre la invocación `setjmp` y la invocación `longjmp`, quedan indeterminados.

Como pasa por alto los mecanismos usuales de llamada de función y de regreso, `longjmp` ejecutará de forma correcta en el contexto de interrupciones, señales y cualquiera de sus funciones asociadas. Sin embargo, si la función `longjmp` es invocada a partir de un manejador de señales anidado (esto es, desde una función invocada como resultado de una señal iniciada durante el manejo de otra señal), el comportamiento queda indefinido.

Después de que `longjmp` se haya completado, la ejecución del programa continuará como si la invocación correspondiente de la macro `setjmp` hubiera devuelto el valor especificado por `val`. La función `longjmp` no puede hacer que la macro `setjmp` devuelva el valor 0; si `val` es 0, la macro `setjmp` devuelve el valor 1.

B.8 Manejo de señales `<signal.h>`

`sig_atomic_t`
El tipo entero de un objeto al cual se puede tener acceso como entidad atómica, inclusive en presencia de interrupciones asincrónicas.

`SIG_DFL`
`SIG_ERR`
`SIG_IGN`
Estas se expanden a expresiones constantes con valores precisos que tienen tipos compatibles con el segundo argumento y con el valor de regreso a la función `signal`, y cuyos valores se comparan en forma desigual con la dirección de cualquier función declarable; en lo que sigue, cada uno de los cuales se expande a una expresión constante entera positiva que es el número de señal para la condición especifica:

`SIGABRT`	terminación anormal, como la que es iniciada por la función `abort`
`SIGFPE`	una operación aritmética errónea, como división por cero o una operación que resulte en desbordamiento
`SIGILL`	detección de una imagen de función inválida, como sería una instrucción ilegal
`SIGINT`	recepción de una señal de atención interactiva
`SIGSEGV`	un acceso inválido a almacenamiento
`SIGTERM`	una solicitud de terminación enviada al programa.

Una puesta en práctica no necesita generar ninguna de estas señales, excepto como resultado de llamadas explícitas a la función `raise`.

```
void (*signal(int sig, void (*func)(int)))(int);
```

Escoge una de tres formas en la cual será subsecuentemente manejado el número de señal `sig`. Si el valor de `func` es `SIG_DEF`, ocurrirá el manejo por omisión de la señal. Si el valor de `func` es `SIG_IGN`, la señal será ignorada. De lo contrario, cuando dicha señal ocurra `func` apuntará a la función a llamarse. Dicha función es un *manejador de señal.*

Cuando ocurre una señal, si `func` apunta a una función, primero se ejecuta el equivalente de `signal (sig, SIG_DFL)` o se ejecuta un bloqueo de la señal, definido por la puesta en práctica. (Si el valor de `sig` es `SIGILL`, la restauración de `SIG_DFL` dependerá de la puesta en práctica). A continuación se ejecuta el equivalente de `(*func) (sig)` La función `func` puede terminar ejecutando un enunciado `return` o llamando a la función `abort`, `exit` o `longjmp`. Si `func` ejecuta un enunciado `return` y el valor de `sig` era `SIGFPE` o cualquier otro valor definido por la puesta en práctica correspondiente a una excepción de cómputo, el comportamiento queda indefinido. De lo contrario, el programa continuará en su ejecución desde el punto en que fue interrumpido.

Si la señal ocurre de forma distinta que como resultado a la llamada de la función `abort` o `raise`, el comportamiento queda indefinido si el manejador de señal llama a cualquier función de la biblioteca estándar distinta a la función `signal` misma (con un primer argumento del número de señal correspondiente a la señal que causó la invocación del manejador) o se refiere a cualquier otro objeto con persistencia estática, distinto que mediante la asignación de un valor a una variable de persistencia estática del tipo `volatile sig_atomic_t`. Además, si dicha llamada a una función `signal` resulta en un regreso `SIG_ERR`, el valor de `errno` queda indeterminado.

Al arranque del programa, el equivalente de

```
signal(sig, SIG_IGN);
```

puede ser ejecutado para algunas señales seleccionadas de una forma definida por la puesta en práctica; el equivalente de

```
signal(sig, SIG_DFL);
```

es ejecutado para todas las demás señales definidas por la puesta en práctica.
Si la solicitud puede ser aceptada, la función `signal` devuelve el valor de `func` para la llamada más reciente a `signal` para la señal especificada `sig`. De lo contrario, se devuelve un valor de `SIG_ERR` y en `errno` queda almacenado un valor positivo.

```
int raise(int sig);
```

La función `raise` envía la señal `sig` al programa en ejecución. Si tiene éxito la función `raise` devuelve cero, y no cero si no lo tiene.

B.9 Argumentos variables <stdarg.h>

`va_list`

Un tipo adecuado para contener información necesaria para las macros `va_start`, `va_arg`, y `va_end`. Si se desea tener acceso a los distintos argumentos, la función llamada declarará un objeto

(llamada en esta sección `ap`) con el tipo `va_list`. El objeto `ap` puede ser pasado como argumento a otra función; si dicha función invoca la macro `va_arg` con el parámetro `ap`, el valor de `ap` en la función llamada es determinada y será pasada a la macro `va_end` antes de cualquier otra referencia `ap`.

```
void va_start(va_list ap, parmN);
```

Será invocada antes de cualquier acceso a argumentos sin nombre. La macro `va_start` inicializa `apuntador` para uso subsecuente por las macros `va_arg` y `va_end`. El parámetro *parmN* es el identificador del parámetro más a la derecha de la lista de parámetros variables en la definición de función (una justo antes de la coma , ...). Si se declara el parámetro *parmN* con la clase de almacenamiento `register`, con un tipo de función o de arreglo, o con un tipo que no sea compatible con el tipo que resulte después de la aplicación de las promociones de argumento por omisión, el comportamiento quedará indefinido.

tipo `va_arg(va_list, tipo);`

Se expande a una expresión que tiene el tipo y el valor del siguiente argumento dentro de la llamada. El parámetro `ap` será el mismo que el de `va_list ap` inicializado por `va_start`. Cada invocación de `va_arg` modifica `ap` de tal forma, que los valores de argumentos sucesivos sean regresados en orden. El parámetro *type* es el nombre de tipo definido de tal forma, que el tipo de un apuntador a un objeto que tiene el tipo especificado se pueda obtener mediante la colocación postfija de un * a *type*. Si no existe un argumento siguiente o si *type* no es compatible con el tipo del siguiente argumento (como promovido de acuerdo con las promociones de argumento por omisión), el comportamiento queda indefinido. La primera invocación de la macro `va_arg` después de la macro `va_start` devuelve el valor del argumento después del especificado por *parmN*. Invocaciones subsiguientes devuelven en sucesión los valores de los argumentos siguientes.

```
void va_end(va_list ap);
```

Facilita el regreso normal de una función cuya lista de argumentos variables fue referida mediante la expansión de `va_start` que inicializó `va_list ap`. La macro `va_end` puede modificar de tal forma `ap`, que ya no sea utilizable (sin una invocación intermedia de `va_start`). Si no existe una invocación correspondiente de la macro `va_start` o si la macro `va_end` no se invoca antes del regreso, el comportamiento queda indefinido.

B.10 Entrada/salida <stdio.h>

`_IOFBF`
`_IOLBF`
`_IONBF`

Expresiones constantes enteras con valores distintos, adecuadas para uso como el tercer argumento a la función `setvbuf`.

`BUFSIZ`

Una expresión constante entera, que representa el tamaño del búfer utilizado por la función `setvbuf`.

`EOF`

Una expresión constante entera negativa, devuelta por varias funciones a fin de indicar fin de archivo, esto es, no más entradas a partir de un flujo.

`FILE`

Un tipo de objeto capaz de registrar toda la información necesaria para controlar un flujo, incluye el indicador de posición de archivo, un apuntador a su búfer asociado (si es que existe alguno), un indicador de error que registre si ha ocurrido algún error de escritura/lectura y un indicador de fin de archivo que registra si se ha llegado al fin del archivo.

FILENAME_MAX
Una expresión constante entera que tiene el tamaño necesario para un arreglo de **char** lo suficiente extensa para contener la cadena de nombre de archivo más larga posible que la puesta en práctica garantiza es posible abrir.

FOPEN_MAX
Una expresión constante entera que es el número mínimo de archivos que la puesta en práctica garantiza se abrirán en forma simultánea.

fpos_t
Un tipo de objeto capaz de registrar toda la información necesaria para especificar todas las posiciones únicas dentro de un archivo.

L_tmpnam
Una expresión constante integral que es del tamaño necesario para un arreglo **char** lo suficiente extensa para contener una cadena de nombre de archivo temporal generado por una función **tmpnam**.

NULL
Una constante de apuntador nulo definido por la puesta en práctica.

SEEK_CUR
SEEK_END
SEEK_SET
Expresiones constantes integrales con valores precisos, adecuadas para uso como el tercer argumento de la función **fseek**.

size_t
El tipo entero sin signo del resultado del operador **sizeof**.

stderr
Expresión del tipo "apuntador a **FILE**" que señala al objeto **FILE** asociado con el flujo estándar de error.

stdin
Expresión del tipo "apuntador a **FILE**" que señala al objeto **FILE** asociado con el flujo de entrada estándar.

stdout
Expresión del tipo "apuntador a **FILE**" que señala al objeto **FILE** asociado con el flujo de salida estándar.

TMP_MAX
Una expresión constante entera que es el número mínimo de nombres únicos de archivo que se generarán mediante la función **tmpnam**. El valor de la macro **TMP_MAX** será de por lo menos 25.

int remove(const char *filename);
Hace que el archivo cuyo nombre es la cadena a la cual señala **filename** ya no sea accesible para dicho nombre. Cualquier intento subsiguiente para abrir dicho archivo utilizando este nombre fallará, a menos que sea creado de nuevo. Si el archivo está abierto, el comportamiento de la función **remove** queda definido por la puesta en práctica. La función **remove** devuelve cero si la operación tiene éxito, y no cero si falla.

int rename(const char *old, const char *new);
Hace que el archivo cuyo nombre es la cadena a la cual señala **old** que de ahí en adelante se conozca por el nombre dado por la cadena a la cual señala **new**. El archivo de nombre **old** ya no es accesible utilizando ese nombre. Si antes de la llamada a la función **rename** existe un archivo llamado mediante la cadena a la cual señala **new**, el comportamiento queda definido por la puesta en práctica. La función **rename** devuelve cero si la operación tiene éxito, no cero si falla, en cuyo caso si el archivo ya existía aún se llamará con su nombre original.

FILE *tmpfile(void);
Genera un archivo temporal binario que será eliminado de forma automática cuando se cierre o al final del programa. Si el programa termina en forma anormal, si este programa temporal abierta se elimina o no, depende de la puesta en práctica. El archivo se abre para actualización mediante el modo "**wb+**". La función **tmpfile** devuelve un apuntador al flujo del archivo que ha sido creado. Si el archivo no puede ser creado, la función **tmpfile** devuelve un apuntador nulo.

char *tmpnam(char *s)
La función **tmpnam** genera una cadena que es un nombre válido de archivo y que no es el mismo que el nombre de un archivo existente. La función **tmpnam** genera una cadena distinta cada vez que es llamada, hasta **TMP_MAX** veces. Si es llamada más de **TMP_MAX** veces, el comportamiento queda definido por la puesta en práctica.

Si el argumento es un apuntador nulo, la función **tmpnam** deja su resultado en un objeto estático interno y devuelve un apuntador a dicho objeto. Llamadas subsecuentes a la función **tmpnam** pueden modificar el mismo objeto. Si el argumento no es un apuntador nulo, se supone que debe señalar a un arreglo de por lo menos **L_tmpnam char**; la función **tmpnam** escribe su resultado en dicho arreglo y devuelve el argumento como su valor.

int fclose(FILE *stream);
La función **fclose** hace que el flujo al cual señala **stream** sea vaciado y el archivo asociado se cierre. Cualquier dato en archivo temporal o búfer aún no escrito para el flujo es entregado al entorno huésped para ser escrito al archivo; cualesquiera datos en archivos temporales no leídos serán descartados. El flujo queda desasociado del archivo. Si el búfer asociado fue asignado en forma automática, queda desasignado. La función **fclose** devuelve cero si el flujo fue cerrado de forma exitosa o bien **EOF** si se detectaron algunos errores.

int fflush(FILE *stream);
Si **stream** señala a un flujo de salida o a uno actualizado en el cual la operación más reciente no ha sido introducida, la función **fflush** hace que todos los datos aún no escritos para dicho flujo sean entregados al entorno huésped o se escriban al archivo; de lo contrario, el comportamiento queda indefinido.

Si **stream** es un apuntador nulo, la función **fflush** lleva a cabo su acción de vaciado sobre todos los flujo para los cuales el comportamiento se ha definido arriba. Si ocurre algún error de escritura la función **fflush** devuelve **EOF**, de lo contrario devuelve cero.

FILE *fopen(const char *filename, const char *mode);
La función **fopen** abre el archivo cuyo nombre es la cadena a la cual señala **filename** y asocia un flujo al mismo. El argumento **mode** señala una cadena que empieza con alguna de las secuencias siguientes:

r	abre archivo de texto para lectura.
w	trunca a cero longitud o crea un archivo de texto para escritura.
a	agrega; abre o crea un archivo de texto para escritura al fin de archivo.
rb	abre archivo binario para lectura.
wb	trunca a longitud cero o crea archivo binario para escritura.
ab	agrega; abre o crea un archivo binario para escritura al fin de archivo.
r+	abre archivo de texto para actualizar (lectura y escritura).
w+	trunca a longitud cero o crea un archivo de texto para actualizar.

a+	agrega; abre o crea un archivo de texto para actualizar, escribiendo al fin de archivo.
r+b o rb+	abre archivo binario para actualizar (lectura y escritura).
w+b o wb+	trunca a longitud cero o crea un archivo binario para actualizar.
a+b o ab+	agrega; abre o crea un archivo binario para actualizar, escribiendo al fin de archivo.

Abrir un archivo en modo de lectura (**'r'** como el primer carácter en el argumento **mode**) fallará si el archivo no existe o no puede ser leído. Abrir el archivo con el modo de agregar (**'a'** como el primer carácter en el argumento **mode**) hace que todas las escrituras subsiguientes al archivo se obliguen a lo que en ese momento sea el fin del archivo, independiente de llamadas a la función **fseek**. En algunas puestas en práctica, la apertura de un archivo binario en modo de agregar (**'b'** como segundo y tercer caracteres en la lista anterior de los valores de argumento **mode**) puede colocar en forma inicial el indicador de posición de archivo para el flujo más allá de los últimos datos escritos, debido al relleno de carácter nulo.

Cuando se abre un archivo con el modo de actualizar (**'+'** como segundo y tercer carácter en la lista anterior de valores de argumento **mode**), se puede llevar a cabo tanto entrada como salida sobre el flujo asociado. Sin embargo, la salida no puede ser seguida en forma directa por entrada, sin una llamada intermedia a la función **fflush** o a la función de posicionamiento de archivo (**fseek**, **fsetpos** o **rewind**), y la entrada no puede ser de manera directa seguida por salida sin una llamada intermedia a una función de posicionamiento de archivo, a menos de que la operación de entrada se encuentre con fin de archivo. La apertura (o creación) de un archivo de texto en el modo de actualizar, puede quizás abrir (o crear) un flujo binario en algunas puestas en práctica particulares.

Una vez abierto, un flujo está por completo equipado con memoria temporal (búfer) si y sólo si se puede determinar que no se refiere a un dispositivo interactivo. Los indicadores de error y de fin de archivo para el flujo estarán desactivados. La función **fopen** devuelve un apuntador al objeto que controla el flujo. Si la operación de apertura falla, **fopen** devuelve un apuntador nulo.

```
FILE *freopen(const char *filename, const char *mode,
              FILE *stream);
```

La función **freopen** abre el archivo cuyo nombre es la cadena a la cual señala **filename** y asocia el flujo al cual apunta **stream**. El argumento **mode** se utiliza de la misma forma que en la función **fopen**.

La función **freopen** intenta primero cerrar cualquier archivo que esté asociado con el flujo especificado. Se ignorará cualquier falla para cerrar el archivo con éxito. Los indicadores de error y de fin de archivo serán desactivados. Si falla la operación de apertura la función **freopen** devuelve un apuntador nulo. De lo contrario, **freopen** devuelve el valor de **stream**.

```
void setbuf(FILE *stream, char *buf);
```

La función **setbuf** es equivalente a la función **setvbuf** que se invoca con los valores **_IOFBF** para **mode** y **BUFSIZ** para **size**, o (**buf** es un apuntador nulo) con el valor **_IONBF** para **mode**. La función **setbuf** no devuelve valor.

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

La función **setvbuf** puede ser utilizada sólo cuando el flujo al cual señala **stream** ha sido asociado con un archivo abierto y antes de cualquier otra operación se haya ejecutado sobre el flujo. El argumento **mode** determina como se harán los búfers de **stream**, como sigue: **_IOFBF** hará que existan búfers completos para entradas/salidas; **_IOLBF** creará búfer de línea para entrada/salida; **_IONBF** hará que las entradas y salidas no tengan búfer. Si **buf** no es un apuntador nulo, el arreglo al cual señala podrá ser utilizado en vez de un búfer asignado por la función **setvbuf**. El argumento **size** especifica el tamaño del arreglo. Los contenidos del arreglo en cualquier momento quedan indeterminados. La función **setvbuf** devuelve cero si tiene éxito o algún valor no cero si se ha dado un valor inválido para **mode** o si la solicitud no puede ser aceptada.


```
int fprintf(FILE *stream, const char *format, ...);
```

La función **fprintf** escribe salida al flujo al cual señala **stream**, bajo control de la cadena al cual señala **format** que define como los argumentos subsecuentes son convertidos para la salida. Si no hay suficientes argumentos para el formato, el comportamiento queda indefinido. Si el formato se termina aunque queden argumentos, los argumentos en exceso se evalúan (como siempre), pero serán ignorados. La función **fprintf** devuelve cuando se encuentra con el fin de la cadena de formato. Vea el capítulo 9, "Entradas/salidas con formato", para una descripción detallada de las especificaciones de conversión de salida. La función **fprintf** devuelve el número de caracteres transmitidos, o un valor negativo si ha ocurrido un error de salida.

```
int fscanf(FILE *stream, const char *format, ...);
```

La función **fscanf** lee la entrada desde el flujo al cual señala **stream**, bajo control de la cadena a la cual apunta **format** que define las secuencias de entrada admisibles y cómo deben de ser convertidas para asignación, utilizando los argumentos subsecuentes como apuntadores a los objetos que han de recibir la entrada convertida. Si para el formato existen argumentos insuficientes, el comportamiento queda indefinido. Si el formato se termina aunque sobren argumentos, los argumentos excedentes se evalúan (como siempre), pero por lo demás se ignorarán. Vea el capítulo 9, "Entradas/salidas con formato", para una descripción detallada de las especificaciones de conversión de entrada.

La función **fscanf** devuelve el valor de la macro **EOF** si ocurre una falla de entrada antes de cualquier conversión. De lo contrario, la función **fscanf** devuelve el número de elementos de entrada asignados, mismos que pueden resultar menos de los proveídos inclusive cero, en caso de una falla temprana de coincidencias.

```
int printf(const char *format, ...);
```

La función **printf** es equivalente a **fprintf** con la interposición del argumento **stdout** antes de los argumentos a **printf**. La función **printf** devuelve el número de caracteres transmitidos o un valor negativo si ocurrió un error de salida.

```
int scanf(const char *format, ...);
```

La función **scanf** es equivalente a **fscanf** con la interposición del argumento **stdin** antes de los argumentos a **scanf**. Si ocurre una falla de entrada antes de cualquier conversión la función **scanf** devuelve el valor de la macro **EOF**. De lo contrario, **scanf** devuelve el número de elementos de entrada asignados, mismos que pueden ser menos de los proveídos o inclusive cero, en el caso de una falla temprana de coincidencia.

```
int sprintf(char *s, const char *format, ...);
```

La función **sprintf** es equivalente a **fprintf**, excepto que el argumento **s** especifica un arreglo al cual deberá de ser escrita la salida generada, en vez de a un flujo. Se escribe un carácter nulo al final de los caracteres escritos; no se cuenta como una parte de la suma devuelta. El comportamiento de copia entre objetos que se superponen queda indefinido. La función **sprintf** devuelve el número de caracteres escritos por el arreglo, sin contar el carácter nulo de terminación.

```
int sscanf(const char *s, const char *format, ...);
```

La función **sscanf** es equivalente a **fscanf**, excepto que el argumento **s** especifica una cadena a partir de la cual se obtendrá la entrada, más bien que a partir de un flujo. El llegar al final de la cadena es equivalente a encontrar fin de archivo para la función **fscanf**. Si la copia ocurre entre objetos que se superponen, el comportamiento queda indefinido.

La función **sscanf** devuelve el valor de la macro **EOF** si ocurre una falla de entrada antes de cualquier conversión. Lo contrario, la función **sscanf** devuelve el número de elementos de entrada asignados, mismos que pueden ser menos de los proveídos inclusive cero, en el caso de una falla temprana de coincidencia.

```
int vfprintf(FILE *stream, const char *format, va_list arg);
```

La función **vfprintf** es equivalente a **fprintf**, con la lista de argumentos variables remplazados por **arg**, que se incializa mediante la macro **va_start** (y mediante posibles llamadas subsecuentes

va_arg). La función vfprintf no invoca a la macro va_end. La función vfprintf devuelve el número de caracteres transmitidos o un valor negativo si ocurrió un error de salida.

```
int vprintf(const char *format, va_list arg);
```

La función vprintf es equivalente a printf, con la lista de argumentos variables remplazadas por arg, que debe haber sido inicializada por la macro va_start (y llamadas subsecuentes posibles va_arg). La función vprintf no invoca a la macro va_end. La función vprintf devuelve el número de caracteres transmitidos o un valor negativo si ocurrió un error de salida.

```
int vsprintf(char *s, const char *format, va_list arg);
```

La función vsprintf es equivalente a sprintf, con la lista de argumentos variables remplazadas por arg, misma que deberá haber sido inicializada por la macro va_start (y subsecuentes posibles llamadas va_arg). La función vsprintf no invoca a la macro va_end. Si la copia se lleva a cabo entre objetos que se superponen, el comportamiento queda indefinido. La función vsprintf devuelve el número de caracteres escritos en el arreglo, sin contar el carácter nulo de terminación.

```
int fgetc(FILE *stream);
```

La función fgetc obtiene el carácter siguiente (si está presente) como un unsigned char convertido a un int, del flujo de entrada al cual señala stream y avanza el indicador de posición de archivo asociado correspondiente al flujo (si está definido). La función fgetc devuelve el siguiente carácter del flujo de entrada al cual señala stream. Si el flujo está al fin de archivo, se activa el indicador de fin de archivo para el flujo y fgetc devuelve EOF. Si ocurre un error de lectura, se activa el indicador de error para el flujo y fgetc devuelve EOF.

```
char *fgets(char *s, int n, FILE *stream);
```

La función fgets lee por lo menos uno menos que el número de caracteres definido por n del flujo al cual señala stream al arreglo al cual señala s. No se leen caracteres adicionales después del carácter de nueva línea (mismos que se conserva) o después de fin de archivo. Se escribe un carácter nulo inmediatamente después del último carácter leído en el arreglo.

Si tiene éxito la función fgets devuelve s. Si se encuentra con un fin de archivo y no se han leído caracteres al arreglo, el contenido del arreglo se conserva sin modificación y se devuelve un apuntador nulo. Si durante la operación ocurre un error de lectura, el contenido del arreglo queda indeterminado y se devuelve un apuntador nulo.

```
int fputc(int c, FILE *stream);
```

La función fputc escribe el carácter definido por c (convertido a unsigned char) al flujo de salida al cual apunta stream, en la posición indicada por el indicador de posición de archivo asociado para el flujo (si está definido), y adelanta en forma apropiada dicho indicador. Si el archivo no puede aceptar solicitudes de posicionamiento o si el flujo fue abierto en modo de agregar, el carácter será agregado al flujo de salida. La función fputc devuelve el carácter escrito. Si ocurre un error de escritura, se activará el indicador de error para el flujo y fputc devolverá EOF.

```
int fputs(const char *s, FILE *stream);
```

La función fputs escribe la cadena a la cual señala s en el flujo al cual señala stream. No es escrito el carácter nulo de terminación. Si ocurre un error de escritura la función fputs devuelve EOF; de lo contrario devolverá un valor no negativo.

```
int getc(FILE *stream);
```

La función getc es equivalente a fgetc, excepto que cuando es puesta en práctica como una macro, pudiera evaluar más de una vez stream —el argumento deberá ser una expresión sin efectos colaterales.

La función getc devuelve el carácter siguiente del flujo de entrada al cual señala stream. Si el flujo está al fin de archivo, se activará el indicador de fin de archivo para el flujo y getc devuelve EOF. Si ocurre un error de lectura, el indicador de error para el flujo se activará y getc devolverá EOF.


```
int getchar(void);
```

La función getchar es equivalente a getc, con el argumento stdin. La función getchar devuelve el siguiente carácter del flujo de entrada al cual señala stdin. Si el flujo está a fin de archivo, se activará el indicador de fin de archivo para el flujo y getchar devolverá EOF. Si ocurre un error de lectura, se activará el indicador de error para el flujo y getchar devolverá EOF.

```
char *gets(char *s);
```

La función gets lee caracteres del flujo de entrada al cual señala stdin, al arreglo apuntado por s, hasta que se encuentra fin de archivo o se lee un carácter de nueva línea. Cualquier carácter de nueva línea será descartado o si no se escribirá un carácter nulo después del último carácter leído al arreglo. La función gets devuelve s si tiene éxito. Si se encuentra un fin de archivo y no se han leído caracteres al arreglo, el contenido del arreglo se conservará sin modificación y se devolverá un apuntador nulo. Si durante la operación ocurre un error de lectura, el contenido del arreglo queda indeterminado y se devuelve un apuntador nulo.

```
int putc(int c, FILE *stream);
```

La función putc es equivalente a fputc, excepto que cuando se pone en práctica como una macro, pudiera evaluar stream más de una vez, por lo que el argumento no debería nunca ser una expresión con efectos colaterales. La función putc devuelve el carácter escrito. Si ocurre un error de escritura, se activa el indicador de error correspondiente al flujo y putc devuelve EOF.

```
int putchar(int c);
```

La función putchar es equivalente a putc con stdout como segundo argumento. La función putchar devuelve el carácter escrito. Si ocurre un error de escritura, se activa el indicador de error para el flujo y putchar devuelve EOF.

```
int puts(const char *s);
```

La función puts escribe la cadena a la cual señala s en el flujo al cual señala stdout, y agrega un carácter de nueva línea a la salida. El carácter nulo de terminación no es escrito. La función puts devuelve EOF si ocurre un error de escritura; de lo contrario, devuelve un valor no negativo.

```
int ungetc(int c, FILE *stream);
```

La función ungetc coloca el carácter especificado por c (convertido a un unsigned char) de regreso en el flujo de entrada al cual señala stream. Los caracteres devueltos serán regresados en lecturas subsecuentes de dicho flujo en orden inverso a su entrada. Una llamada intermedia exitosa (con el flujo al cual apunta stream) a un función de posicionamiento de archivo (fseek, fsetpos, o rewind) descartará todos los caracteres vueltos a colocar para el flujo. El almacenamiento externo correspondiente al flujo se mantiene sin modificación.

Un carácter de los regresados queda garantizado. Si una función ungetc es llamada demasiadas veces sobre el mismo flujo sin una operación intermedia de posicionamiento de lectura o de archivo sobre dicho flujo, la operación pudiera fallar. Si el valor de c se iguala al de la macro EOF, la operación fallará y el flujo de entrada queda sin modificar.

Una llamada exitosa a la función ungetc desactiva el indicador de fin de archivo correspondiente al flujo. El valor del indicador de posición de archivo para el flujo después de leer o descartar todos los caracteres introducidos y regresados será el mismo que era antes de que los caracteres fueran retornados. Para un flujo de texto, el valor de este indicador de posición de archivo después de una llamada a la función ungetc queda sin especificar en tanto todos los caracteres devueltos al flujo sean leídos o descartados. En el caso de un flujo binario, su indicador de posición de archivo queda determinado mediante cada llamada exitosa a la función ungetc; si su valor antes de la llamada era cero, quedará después de la llamada indeterminado. La función ungetc devuelve el carácter devuelto al flujo después de la conversión o si no en caso de fallo de la operación EOF.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

La función fread lee, al arreglo al cual señala ptr, hasta nmemb elementos cuyos tamaños están definidos por size, del flujo al cual señala stream. El indicador de posición de archivo para el flujo

(si está definido) se avanza en el número de caracteres con éxito leídos. Si ocurre algún error, queda indeterminado el valor resultante del indicador de posición de archivo para el flujo. Si se lee un elemento en forma parcial, su valor también queda indeterminado.

La función **fread** devuelve el número de elementos leídos, mismos que pudieran ser menos de **nmemb** si se encuentra con un error de lectura o con el fin de archivo. Si **size** o **nmemb** es cero, **fread** devuelve cero y tanto el contenido del arreglo como el estado del flujo se conservan sin modificar.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

La función **fwrite** escribe, desde el arreglo al cual señala **ptr** hasta **nmemb** elementos cuyo tamaño está definido por **size**, al flujo al cual señala **stream**. El indicador de posición de archivo para el flujo (si está definido) se avanza en el número de carácter escritos con éxito. Si ocurre un error, el valor resultante de la posición de archivo para el flujo queda indeterminado. La función **fwrite** devuelve el número de elementos escritos, mismo que sólo en el caso de que se encuentre un error de escritura será menor que **nmemb**.

```
int fgetpos(FILE *stream, fpos_t *pos);
```

La función **fgetpos** almacena el valor actual del indicador de posición de archivo para el flujo al cual señala **stream** en el objeto al cual apunta **pos**. El valor almacenado contiene información no especificada utilizable por la función **fsetpos** para recolocar el flujo a su posición en el momento de la llamada de la función **fgetpos**. En caso de éxito, la función **fgetpos** devuelve cero; en caso de falla, la función **fgetpos** devuelve no cero y almacena un valor positivo definido según la puesta en práctica en **errno**.

```
int fseek(FILE *stream, long int offset, int whence);
```

La función **fseek** define el indicador de posición de archivo para el flujo al cual señala **stream**. En el caso de un flujo binario, la nueva posición, medida en caracteres a partir del principio del archivo, se obtiene añadiendo **offset** a la posición especificada por **whence**. La posición especificada es el principio del archivo en el caso de que **whence** sea **SEEK_SET**, el valor actual del indicador de posición de archivo si **whence** es **SEEK_CUR**, o fin de archivo si es **SEEK_END**. Un flujo binario no necesariamente debe aceptar en forma significativa llamadas **fseek** con un valor **whence** de **SEEK_END**. En el caso de un flujo de texto, u **offset** deberá de ser cero, o un valor devuelto por una llamada anterior a la función **ftell** sobre el mismo flujo y **whence** deberá ser **SEEK_SET**.

Una llamada con éxito a la función **fseek** desactiva el indicador de fin de archivo para el flujo y deshace cualquier efecto de la función **ungetc** sobre el mismo archivo. Después de una llamada a **fseek**, la siguiente operación en un flujo de actualizar puede ser entrada o salida. La función **fseek** devuelve no cero sólo en el caso de una solicitud que no puede ser satisfecha.

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

La función **fsetpos** define el indicador de posición de archivo para el flujo al cual señala **stream** de acuerdo al valor del objeto al cual señala **pos**, mismo que debe de ser un valor obtenido de una llamada anterior a la función **fgetpos** sobre el mismo flujo. Una llamada con éxito a la función **fsetpos** desactiva el indicador de fin de archivo para el flujo y deshace cualquier efecto de la función **ungetc** del mismo flujo. Después de una llamada a **fsetpos**, la siguiente operación sobre un flujo de actualizar puede ser introducida o extraída. Si tiene éxito, la función **fsetpos** devuelve cero; en caso de falla, la función **fsetpos** devuelve no cero y almacena un valor positivo definido según la puesta en práctica en **errno**.

```
long int ftell(FILE *stream);
```

La función **ftell** obtiene el valor actual del indicador de posición de archivo del flujo al cual señala **stream**. En el caso de un archivo binario, el valor es el número de caracteres a partir del principio del archivo. En el caso de un archivo de texto, el indicador de posición de archivo contiene información no especifica, utilizable por la función **fseek** para devolver el indicador de posición de archivo para el flujo a su posición en el momento de la llamada **ftell**; la diferencia entre estos dos valores devueltos no es necesariamente una medida significativa del número de caracteres escritos o leídos. En caso de éxito, una función **ftell** devuelve el valor actual del indicador de posición de archivo para el flujo. En caso de falla, la función **ftell** devuelve **-1L** y almacena un valor positivo definido por la puesta en práctica en **errno**.


```
void rewind(FILE *stream);
```

La función **rewind** define el indicador de posición de archivo para el flujo al cual señala **stream** al principio del mismo. Es equivalente a

```
(void) fseek(stream, 0L, SEEK_SET)
```

excepto que el indicador de error del flujo también es desactivado.

```
void clearerr(FILE *stream);
```

La función **clearerr** desactiva los indicadores de fin de archivo y de error para el flujo al cual señala **stream**.

```
int feof(FILE *stream);
```

La función **feof** prueba el indicador de fin de archivo en el flujo al cual señala **stream**. La función **feof** devuelve no cero si y sólo si el indicador de fin de archivo está activo para **stream**.

```
int ferror(FILE *stream);
```

La función **ferror** prueba el indicador de error para el flujo al cual señala **stream**. La función **ferror** devuelve no cero si y sólo si el indicador de error está activo para **stream**

```
void perror(const char *s);
```

La función **perror** traduce el número de error en la expresión entera **errno** en un mensaje de error. Escribe una secuencia de caracteres al flujo de error estándar, de tal forma que: primero (si **s** no es un apuntador nulo y el carácter al cual señala **s** no es un carácter nulo), escribe la cadena a la cual señala **s** seguida por (:) y un espacio; a continuación escribe un mensaje de error apropiado seguido por un carácter de nueva línea. El contenido de las cadenas de mensaje de error son las mismas que devuelve la función **strerror** mediante el argumento **errno**, que están definidas por la puesta en práctica.

B.11 Utilerías generales <stdlib.h>

EXIT_FAILURE
EXIT_SUCCESS

Expresiones enteras que pueden ser utilizadas como argumentos de la función **exit** para devolver estados de terminación con o sin éxito, al entorno huésped respectivamente.

MB_CUR_MAX

Una expresión entera positiva cuyo valor es el número máximo de bytes en un carácter de multibyte para el conjunto de caracteres extendido definido por el escenario actual (categoría **LC_CTYPE**), y cuyo valor nunca es mayor de **MB_LEN_MAX**.

NULL

Una constante de apuntador nulo definida por la puesta en práctica.

RAND_MAX

Una expresión constante de intervalo, cuyo valor es el valor máximo devuelto por la función **rand**. El valor de la macro **RAND_MAX** deberá ser por lo menos 32767.

div_t

Un tipo de estructura que es el tipo de valor devuelto por la función **div**.

ldiv_t

Un tipo de estructura que es el tipo de valor devuelto por la función **ldiv**.

size_t
El tipo entero no signado del resultado del operador **sizeof**.

wchar_t
Un tipo entero cuyo rango de valores puede representar códigos precisos para todos los miembros del conjunto de caracteres extendido más especifico entre escenarios soportados; el carácter nulo deberá tener el valor de código cero y cada miembro del conjunto de caracteres básico deberá tener un valor de código igual a su valor cuando se utilice como un carácter solo en una constante de caracteres entera.

```
double atof(const char *nptr);
```
Convierte la porción inicial de la cadena a la cual señala **nptr** a representación **double**. La función **atof** devuelve el valor convertido.

```
int atoi(const char *nptr);
```
Convierte la porción inicial de la cadena a la cual señala **nptr** a representación **int**. La función **atoi** devuelve el valor convertido.

```
long int atol(const char *nptr);
```
Convierte la porción inicial de la cadena a la cual señala **nptr** a representación **long**. La función **atol** devuelve el valor convertido.

```
double strtod(const char *nptr, char **endptr);
```
Convierte la porción inicial de la cadena a la cual señala **nptr** a representación **double**. Primero, divide en tres partes la cadena de entrada: una secuencia inicial, posiblemente vacía, de caracteres en blanco (tal y como se define en la función **isspace**), una secuencia sujeto, parecida a una constante de punto flotante; y una cadena final de uno o más caracteres no reconocidos, que incluyen el carácter nulo de terminación de la cadena de entrada. A continuación, intenta convertir la secuencia sujeto a un número de punto flotante y devuelve el resultado.

La forma expandida de la secuencia sujeto es un signo más o un signo menos opcional, a continuación una secuencia de dígitos no vacía, que de forma opcional contenga un carácter de punto decimal, a continuación una parte exponencial opcional, pero sin sufijo flotante. La secuencia sujeto se define como la subsecuencia inicial más extensa de la cadena de entrada, empezando con el primer carácter de espacio no en blanco, esto es de la forma esperada. La secuencia sujeto no contiene ningún carácter, si la cadena de entrada está vacía o está solo formada por espacios en blanco o si el carácter de no espacio en blanco, que está en primer término, es distinto de un signo, de un dígito, o de un carácter de punto decimal.

Si la secuencia sujeto tiene la forma esperada, la secuencia de caracteres que se inicia con el primer dígito o con el carácter de punto decimal (lo que ocurra primero) se interpreta como una constante flotante, excepto que el carácter de punto decimal se utilizará en lugar de un periodo, y que si no aparecen ni una parte exponencial ni un carácter de punto decimal, se supondrá un punto decimal que deberá seguir al último dígito en la cadena. Si la secuencia sujeto empieza con un signo menos, el valor que resulte de la conversión se hace negativa. Un apuntador a la cadena final se almacena en el objeto al cual señala **endptr**, siempre y cuando **endptr** no sea un apuntador nulo.

Si la secuencia sujeto está vacía o no tiene la forma esperada, no se llevará a cabo la conversión; el valor **nptr** se almacenará en el objeto al cual señala **endptr**, siempre y cuando que **endptr** no sea un apuntador nulo.

La función **strtod** devuelve el valor convertido si es que hubiera alguno. Si no se ha ejecutado conversión, devuelve cero. Si el valor correcto queda fuera del rango de los valores representables, se devuelve más o menos **HUGE_VAL** (según el signo del valor), y se almacena el valor de la macro **ERANGE** en **errno**. Si el valor correcto pudiera generar desbordamiento, se devuelve cero y el valor de la macro **ERANGE** queda almacenado en **errno**.


```
long int strtol(const char *nptr, char **endptr, int base);
```
Convierte la porción inicial de la cadena a la cual señala **nptr** a **long int**. Primero, descompone la cadena de entrada en tres partes: una secuencia inicial, posiblemente vacía, de caracteres de espacio en blanco (según queda especificado por la función **isspace**), una secuencia sujeto, que se parece a un entero representado en alguna raíz o base determinada por el valor de **base**, y una cadena final, de uno o más caracteres reconocibles, incluyendo el carácter nulo de terminación de la cadena de entrada. A continuación, intenta convertir la secuencia sujeto a un entero, y devuelve el resultado.

Si el valor de **base** es cero, la forma esperada de la secuencia sujeto es la de una constante entera, precedida en forma opcional de un signo más o menos, pero no incluyendo un sufijo entero. Si el valor de **base** es entre 2 y 36, la forma esperada de la secuencia sujeto es una secuencia de letras y dígitos que representan un entero con la raíz o base especificada por **base**, precedida en forma opcional por un signo más o menos, pero sin incluir un sufijo entero. Las letras desde **a** (o **A**) hasta **z** (o **Z**) reciben los valores 10 a 35; sólo se permiten aquellas letras cuyos valores adscritos sean menores que el de la **base**. Si el valor de la **base** es 16, los caracteres **0x** ó **0X** pudieran opcionalmente preceder a la secuencia de letras y de dígitos, a continuación del signo, si es que está presente.

La secuencia sujeto se define como la subsecuencia inicial más larga de la cadena de entrada, empezando a partir del primer carácter que no sea de espacio en blanco, esto es, de la forma esperada.

La secuencia sujeto no contiene ningún carácter, si la cadena de entrada está vacía o está formada sólo de espacios en blanco, o si el primer carácter distinto a espacio en blanco es diferente a un signo o a una letra o dígito permisible.

Si la secuencia sujeto tiene la forma esperada y el valor **base** es cero, la secuencia de caracteres que empieza a partir del primer dígito se interpreta como una constante entera. Si la secuencia sujeto tiene la forma esperada y el valor de la **base** es entre 2 y 36, se utiliza como base para la conversión, dándole a cada letra su valor como se especifica arriba. Si la secuencia sujeto empieza con un signo menos, se colocará un signo menos al valor resultante de la conversión. Un apuntador a la cadena final queda almacenado en el objeto al cual señala **endptr**, siempre y cuando **endptr** no sea nulo.

Si la secuencia sujeto está vacía o no tiene la forma esperada, la conversión no se llevará a cabo; el valor **nptr** se almacenará en el objeto al cual señala **endptr**, siempre y cuando **endptr** no sea un apuntador nulo.

La función **strtol** devuelve el valor convertido, si es que hubiera alguno. Si no se pudo ejecutar ninguna conversión, se devuelve cero. Si el valor correcto queda fuera del rango de valores representables, se devuelve **LONG_MAX** o **LONG_MIN** (de acuerdo al signo del valor), y el valor de la macro **ERANGE** queda almacenado en **errno**.

```
unsigned long int strtoul(const char *nptr, char **endptr, int base);
```
Convierte la porción inicial de la cadena a la cual señala **nptr** a una representación **unsigned long int**. La función **strtoul** funciona de forma idéntica a la función **strtol**. La función **strtoul** devuelve el valor convertido, si es que hay alguno. Se devuelve cero, si no se pudo llevar a cabo la conversión. Si el valor correcto queda por fuera del rango de los valores representables, se devuelve **ULONG_MAX**, y se almacena el valor de la macro **ERANGE** en **errno**.

```
int rand(void);
```
La función **rand** calcula una secuencia de enteros pseudo aleatorios de rango 0 hasta **RAND_MAX**. La función **rand** devuelve un entero pseudo aleatorio.

```
void srand(unsigned int seed);
```
Utiliza el argumento como semilla para una nueva secuencia de números pseudo aleatorios, a devolverse en subsecuentes llamadas a **rand**. Si entonces es llamada a **srand** con el mismo valor de semilla, la secuencia de los números pseudo aleatorios se repetirá. Si se llama a **rand** antes de cualquier llamada a **srand**, la misma secuencia será generada como si se hubiera llamado primero a **srand** con un valor de semilla de 1. La siguientes funciones definen una puesta en práctica portátil de **rand** y de **srand**.

```
static unsigned long int next = 1;
```

```
int rand(void) /* RAND_MAX assumed to be 32767 */
{
   next = next * 1103515245 + 12345;
   return (unsigned int) (next/65536) % 32768;
}

void srand(unsigned int seed)
{
   next = seed;
}
```

void *calloc(size_t nmemb, size_t size);

Asigna espacio para un arreglo de objetos **nmemb**, cada uno de los cuales tiene un tamaño de **size**. El espacio es inicializado para todos los bits cero. La función **calloc** devuelve al espacio asignado un apuntador nulo o un apuntador.

void free(void *ptr);

Hace que sea desasignado el espacio al cual señala **ptr**, esto es, se deje disponible para otra asignación. Si **ptr** es un apuntador nulo, no ocurrirá ninguna acción. De lo contrario, si el argumento no coincide con un apuntador ya devuelto por las funciones **calloc**, **malloc** o **realloc**, o si el espacio ya ha sido desasignado mediante una llamada a **free** o **realloc**, el comportamiento queda indefinido.

void *malloc(size_t size);

Asigna espacio para un objeto cuyo tamaño queda especificado por **size** y cuyo valor es indeterminado. La función **malloc** devuelve un apuntador nulo o un apuntador al espacio asignado.

void *realloc(void *ptr, size_t size);

Modifica el tamaño del objeto al cual señala **ptr** al tamaño definido por **size**. El contenido del objeto deberá conservarse sin modificar hasta el menor de entre los tamaños nuevo y viejo. Si el nuevo tamaño es mayor, el valor de la porción recién asignada del objeto queda indeterminada. Si **ptr** es un apuntador nulo, la función **realloc** se comporta como la función **malloc** para el tamaño especificado. De lo contrario, si **ptr** no encuentra un apuntador devuelto por las funciones **calloc**. **malloc**, o **realloc**, o si el espacio ha sido ya desasignado mediante una llamada a las funciones **free** o **realloc**, el comportamiento queda indefinido. Si el espacio no puede ser asignado, el objeto al cual apunta **ptr** queda sin modificación. Si **size** es cero y **ptr** no es nulo, el objeto al cual señala queda liberado. La función **realloc** devuelve o un apuntador nulo o un apuntador al espacio asignado posiblemente relocalizado.

void abort(void);

Hace que ocurra una terminación anormal de programa, a menos de que se detecte una señal **SIGABRT** y el manejador de señal no se devuelva. Queda dependiente de la puesta en marcha si se vaciarán los flujos de salida abiertos o si se cerrarán los flujos abiertos y serán eliminados los archivos temporales. Mediante la llamada de función **raise(SIGABRT** se devuelve al entorno huésped una forma dependiente de la puesta en práctica del estado de *terminación no exitoso*). La función **abort** no puede devolver a su llamador.

int atexit(void (*func)(void));

Registra la función a la cual apunta **func**, para que a la terminación normal de un programa ésta sea llamada sin argumentos. La puesta en práctica debe aceptar el registro de por lo menos 32 funciones. La función **atexit** devuelve cero si el registro tiene éxito, y no cero si falla.

void exit(int status);

Hace que ocurra una terminación normal del programa. Si el programa ejecuta más de una llamada a la función **exit**, el comportamiento queda indefinido. Primero, serán llamadas todas las funciones registradas por la función **atexit**, en orden inverso a la de su registro. Cada función será llamada

tantas veces como fue registrada. A continuación, serán vaciados todos los flujos abiertos con datos en búfer no escritos, serán cerrados todos los flujos abiertos, y serán eliminados todos los archivos creados mediante la función **tempfile**.

Por último, el control será devuelto al entorno huésped. Si el valor de **status** es cero o **EXIT_SUCCESS**, será devuelta una forma, definida por la puesta en práctica, del estado de *terminación con éxito*. Si el valor de **status** es **EXIT_FAILURE**, será devuelta una forma, definida por la puesta en práctica, del estado *terminación sin éxito*. De lo contrario, el estado devuelto queda definido por la puesta en práctica. La función **exit** no puede devolver a su llamador.

char *getenv(const char *name);

Busca en una *lista de entorno*, proporcionada por el entorno huésped, una cadena que coincida con la cadena a la cual apunta **name**. El conjunto de nombres de entorno y el método para modificar la lista de entorno, quedan definidas por la puesta en práctica. Devuelve un apuntador a una cadena asociada con el miembro de lista coincidente. La cadena a la cual apunta no será modificada por el programa, pero puede ser sobrescrita por una llamada subsiguiente a la función **getenv**. Si no puede ser encontrado el **name** especificado, devuelve un apuntador nulo.

int system(const char *string);

Pasa la cadena a la cual apunta **string** al entorno huésped para que sea ejecutada por un *procesador de comandos* de una forma definida por la puesta en práctica. Se puede utilizar un apuntador nulo en lugar de **string** para indagar si existe el procesador de comandos. Si el argumento es un apuntador nulo, la función **system** devolverá no cero sólo en el caso de que el procesador de comandos esté disponible. Si el argumento no es un apuntador nulo, la función **system** devolverá un valor definido por la puesta en práctica.

void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));

Busca en un arreglo de objetos **nmemb**, el elemento inicial al cual señala **base**, en busca de un elemento que coincida con el objeto al cual señala **key**. El tamaño de cada elemento del arreglo se especifica mediante **size**. Se llama a la función de comparación a la cual señala **compar**, con dos argumentos que apuntan al objeto **key** y a un elemento del arreglo, en este orden. La función devolverá un entero menor que, igual que o mayor que cero si el objeto **key** se considera, respectivamente, ser menor que, coincidir o ser mayor que el elemento del arreglo. El arreglo debe consistir de: todos los elementos que se comparan como menores que, todos los elementos que se comparan como iguales y todos los elementos que se comparan como mayores que el objeto **key**, en ese orden.

La función **bsearch** devuelve un apuntador a un elemento coincidente del arreglo, o un apuntador nulo, si no encuentra coincidencia. Si dos elementos son comparados como iguales, el elemento coincidente queda indeterminado.

void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));

Ordena un arreglo de **nmemb** objetos. El elemento inicial es el que señala a **base**. El tamaño de cada objeto está especificado por **size**. El contenido del arreglo es clasificado u ordenado en orden ascendente, de acuerdo con la función de comparación a la cual señala **compar**, misma que es llamada con dos argumentos, que apuntan a los objetos bajo comparación. La función devuelve un entero menor que o igual, o mayor que cero si el primer argumento se considera ser respectivamente menor que, igual a o mayor que el segundo. Si los dos elementos que se comparan son iguales, su orden en el arreglo clasificado queda indefinido.

int abs(int j);

Calcula el valor absoluto de un entero **j**. Si el resultado no puede ser representado, el comportamiento queda indefinido. La función **abs** devuelve el valor absoluto.

div_t div(int numer, int denom);

Calcula el cociente y residuo o módulo de la división del numerador **numer** entre el denominador **denom**. Si la división es inexacta, el cociente resultante es el entero de menor magnitud que resulte más cercano al cociente algebraico. Si el resultado no puede ser representado, el comportamiento queda indefinido; de lo contrario, **quot * denom + rem** debe ser igual a **numer**. La función **div** devuelve una estructura del tipo **div_t**, que comprende tanto el cociente como el residuo. La estructura debe contener los miembros siguientes, en cualquier orden:

```
int quot;    /* quotient */
int rem;     /* remainder */
```

```
long int labs(long int j);
```

Similar a la función **abs**, excepto que el argumento y el valor devuelto cada uno de ellos tiene el tipo **long int**.

```
ldiv_t ldiv(long int numer, long int denom);
```

Similar a la función **div**, excepto que el argumento y los miembros de la estructura devuelta (que tiene el tipo **ldiv_t**) todos ellos tienen el tipo **long int**.

```
int mblen(const char *s, size_t n);
```

Si **s** no es un apuntador nulo, la función **mblen** determina el número de bytes contenidos en el carácter multibyte al cual señala **s**. Si **s** es un apuntador nulo, la función **mblen** devuelve un valor no cero o cero, si las codificaciones de caracteres de multibyte, respectivamente tienen o no, codificaciones dependientes del estado. Si **s** no es un apuntador nulo, la función **mblen** devuelve cero (si **s** apunta al carácter nulo) o devuelve el número de bytes contenida en el carácter de multibyte (si los siguientes **n** o menos bytes forman un carácter de multibyte válidos), o devuelve -1 (si no forman un carácter de multibyte válido).

```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Si **s** no es un apuntador nulo, la función **mbtowc** determina el número de bytes contenidos en el carácter de multibyte al cual señala **s**. A continuación determina el código para el valor del tipo **wchar_t** que corresponda a dicho carácter de multibyte. (El valor del código que corresponda al carácter nulo es cero). Si el carácter de multibyte es válido y **pwc** no es un apuntador nulo, la función **mbtowc** almacena el código en el objeto al cual apunta **pwc**. Por lo menos n bytes del arreglo al cual apunta **s** serán examinados.

Si **s** es un apuntador nulo, la función **mbtowc** devuelve un valor no cero o cero, si las codificaciones de caracteres de multibyte, en forma respectiva tienen o no tienen codificaciones que dependan del estado. Si **s** no es un apuntador nulo, la función **mbtowc** devuelve cero (si **s** señala al carácter nulo) o devuelve el número de bytes que están contenidas en el carácter de multibyte convertido (si la **n** siguiente o menos bytes forman un carácter de varios bytes válidos), o devuelve -1 (si no forman un carácter de multibyte válido). En ninguno de los casos el valor devuelto será mayor que n o el valor de la macro **MB_CUR_MAX**.

```
int wctomb(char *s, wchar_t wchar);
```

La función **wctomb** determina el número de bytes necesarios para representar un carácter de multibyte correspondiendo al código cuyo valor es **wchar** (incluyendo cualquier modificación en estado de desplazamiento). Almacena la representación de caracteres de multibyte en el objeto de arreglo al cual apunta **s** (si **s** no es un apuntador nulo). Como máximo se almacenan **MB_CUR_MAX** caracteres. Si el valor de **wchar** es cero, la función **wctomb** se deja en el estado de desplazamiento inicial.

Si **s** es un apuntador nulo, la función **wctomb** devuelve un valor no cero o cero, en el caso que las codificaciones de caracteres de multibyte, respectivamente tengan o no tengan codificaciones dependiendo del estado. Si **s** no es un apuntador nulo, la función **wctomb** devuelve -1 si el valor de **wchar** no corresponde a un carácter de multibyte válido o devuelve el número de bytes contenidos en el carácter de multibyte correspondiente al valor de **wchar**. En ningún caso el valor devuelto será mayor que el valor de la macro **MB_CUR_MAX**.


```
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

La función **mbstowcs** convierte una secuencia de caracteres multibyte que empieza en el estado de desplazamiento inicial del arreglo al cual apunta **s** en una secuencia de códigos correspondientes y almacena no más de **n** códigos en el arreglo al cual señala **pwcs**. No se examinarán o convertirán caracteres de multibyte que sigan después de un carácter nulo (mismo que se convierte a un código con valor cero). Cada carácter multibyte se convierte igual que en el caso se una función **mbtowc**, excepto de que no se afecta el estado de desplazamiento de la función **mbtowc**.

No se modificarán más de **n** elementos en el arreglo al cual señala **pwcs**. El comportamiento de copiar entre objetos que se superpongan queda indefinido. Si se encuentra un carácter de multibyte inválido, la función **mbstowcs** devuelve (**size_t**)-1. De no ser así, la función **mbstowcs** devuelve el número de elementos de arreglos modificados, sin incluir el código cero de terminación, si es que hay alguno.

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

La función **wcstombs** convierte una secuencia de códigos que corresponden a caracteres de multibyte del arreglo al cual apunta **pwcs** en una secuencia de caracteres de multibyte que empieza en el estado inicial de desplazamiento y almacena estos caracteres de multibyte en el arreglo al cual señala **s**, deteniéndose si un carácter multibyte se excediese del límite total de bytes igual a **n** o si se almacena un carácter nulo. Cada código se convierte igual que en el caso de una llamada a una función **wctomb**, a excepción de que el estado de desplazamiento de la función **wctomb** no se afecta.

No se modificarán más de **n** bytes en el arreglo al cual señala **s**. Si la copia ocurre entre objetos que se superponen, el comportamiento queda indefinido. Si se encuentra con un código que no corresponda con un carácter válido de multibyte, la función **wcstombs** devuelve (**size_t**) -1. De lo contrario, la función **wcstombs** devuelve el número de bytes modificados, sin incluir el carácter nulo de terminación, si existiese.

B.12 Manejo de cadenas <string.h>

NULL

Una constante de apuntador nula, definida por la puesta en práctica.

size_t

El tipo integral no signado resultante del operador **sizeof**.

```
void *memcpy(void *s1, const void *s2, size_t n);
```

La función **memcpy** copia n caracteres del objeto al cual señala **s2** al objeto al cual señala **s1**. Si la copia tiene lugar entre objetos que se superponen, el comportamiento queda indefinido. La función **memcpy** devuelve el valor de **s1**.

```
void *memmove(void *s1, const void *s2, size_t n);
```

La función **memmove** copia **n** caracteres del objeto al cual señala **s2** en el objeto al cual señala **s1**. La copia se lleva a cabo como si los **n** caracteres del objeto al cual señala **s2** se copian primero a un arreglo temporal de **n** caracteres, que no se superponen sobre los objetos al cual señala **s1** y **s2**, y a continuación los **n** caracteres del arreglo temporal se copian al objeto al que señala **s1**. La función **memmove** devuelve el valor de **s1**.

```
char *strcpy(char *s1, const char *s2);
```

La función **strcpy** copia la cadena a la cual señala **s2** (incluyendo el carácter nulo de terminación) al arreglo al cual señala **s1**. Si la copia se lleva a cabo entre objetos que se superponen, el comportamiento queda indefinido. La función **strcpy** devuelve el valor de **s1**.

```
char *strncpy(char *s1, const char *s2, size_t n);
```

La función **strncpy** copia no más de **n** caracteres, (caracteres que sigan a un carácter nulo ya no son copiados) del arreglo al cual señala **s2** al arreglo al cual señala **s1**. Si la copia se lleva a cabo entre objetos que se superponen, el comportamiento queda indefinido. Si el arreglo al cual señala **s2** es una cadena más corta de **n** caracteres, se agregarán caracteres nulos a la copia en el arreglo al cual señala **s1**, hasta que en total se hayan escrito **n** caracteres. La función **strncpy** devuelve el valor de **s1**.

```
char *strcat(char *s1, const char *s2);
```

La función **strcat** agrega una copia de la cadena a la cual señala **s2** (incluye el carácter nulo de terminación) al final de la cadena a la cual señala **s1**. El carácter inicial de **s2** sobreescribe el carácter nulo existente al final de **s1**. Si la copia se lleva a cabo entre objetos que se superponen, el comportamiento queda indefinido. La función **strcat** devuelve el valor de **s1**.

```
char *strncat(char *s1, const char *s2, size_t n);
```

La función **strncat** agrega no más de **n** caracteres (el carácter nulo y los caracteres que le sigan no serán agregados) del arreglo al cual apunta **s2** al final de la cadena a la cual apunta **s1**. El carácter inicial de **s2** sobreescribe el carácter nulo al final de **s1**. Siempre se agregará un carácter nulo de terminación al resultado. Si la copia se lleva a cabo entre objetos que se superponen, el comportamiento queda indefinido. La función **strncat** devuelve el valor de **s1**.

```
int memcmp(const void *s1, const void *s2, size_t n);
```

La función **memcmp** compara los primeros **n** caracteres del objeto al cual señala **s1** con los primeros **n** caracteres del objeto al cual señala **s2**. La función **memcmp** devuelve un entero mayor que, igual a o menor que cero, según si el objeto al cual señala **s1** es mayor que, igual a o menor que el objeto al cual señala **s2**.

```
int strcmp(const char *s1, const char *s2);
```

La función **strcmp** compara la cadena a la cual señala **s1** con la cadena a la cual señala **s2**. La función **strcmp** devuelve un entero mayor que, igual a o menor que cero, según que la cadena a la cual señala **s1** sea mayor que, igual a o menor que la cadena a la cual señala **s2**.

```
int strcoll(const char *s1, const char *s2);
```

La función **strcoll** compara la cadena a la cual señala **s1** con la cadena a la cual señala **s2**, ambas interpretadas como apropiadas a la categoría **LC_COLLATE** del escenario actual. La función **strcoll** devuelve un entero mayor que, igual a o menor que cero, según que la cadena a la cual señala **s1** sea mayor que, igual a o menor que la cadena a la cual señala **s2** cuando ambos están interpretadas como apropiadas para el escenario actual.

```
int strncmp(const char *s1, const char *s2, size_t n);
```

La función **strncmp** compara no más de **n** caracteres (los caracteres que sigan a un carácter nulo no son comparados) del arreglo al cual señala **s1** con el arreglo al cual señala **s2**. La función **strncmp** devuelve un entero mayor que, igual a o menor que cero, según si el arreglo, posiblemente terminado en nulo, al cual señala **s1**, es mayor que, igual a o menor que el arreglo, posiblemente terminado en nulo, al cual señala **s2**.

```
size_t strxfrm(char *s1, const char *s2, size_t n);
```

La función **strxfrm** transforma la cadena a la cual señala **s2** y coloca la cadena resultante en el arreglo al cual señala **s1**. La transformación es tal que si la función **strcmp** se aplica a las dos cadenas transformadas, devuelve un valor mayor que igual a o menor que cero, en concordancia con el resultado de la función **strcoll** aplicable a las dos cadenas originales. No se colocan más de **n** caracteres en el arreglo resultante al cual señala **s1**, incluyendo el carácter nulo de terminación. Si **n** es cero, **s1** es posible que sea un apuntador nulo. Si la copia se lleva acabo entre objetos que se superponen, el comportamiento queda indefinido. La función **strxfrm** devuelve la longitud de la cadena transformada (sin incluir el carácter nulo de terminación). Si el valor es **n** o más, el contenido del arreglo al cual señala **s1** queda indeterminado.


```
void *memchr(const void *s, int c, size_t n);
```

La función **memchr** localiza la primera instancia de **c** (convertida a un **unsigned char**) en los **n** caracteres iniciales (cada uno de ellos interpretados como un **unsigned char**) del objeto al cual señala **s**. La función **memchr** devuelve un apuntador al carácter localizado, o un apuntador nulo si el carácter no existe en el objeto.

```
char *strchr(const char *s, int c );
```

La función **strchr** localiza la primera instancia de **c** (convertido a un **char**) en la cadena a la cual señala **s**. El carácter nulo de terminación se considera como parte de la cadena. La función **strchr** devuelve un apuntador al carácter localizado, o un apuntador nulo si el carácter no está incluido en la cadena.

```
size_t strcspn(const char *s1, const char *s2);
```

La función **strcspn** calcula la longitud del segmento inicial máximo de la cadena a la cual señala **s1** que está formada por completo de caracteres que no sean de la cadena a la cual señala **s2**. La función **strcspn** devuelve la longitud del segmento.

```
char *strpbrk(const char *s1, const char *s2);
```

La función **strpbrk** localiza la primera instancia de la cadena a la cual señala **s1** de cualquier carácter de la cadena a la cual señala **s2**. La función **strpbrk** devuelve un apuntador al carácter, o un apuntador nulo si ningún carácter de **s2** ocurre en **s1**.

```
char *strrchr(const char *s, int c);
```

La función **strrchr** localiza la última instancia de **c** (convertido a **char**) en la cadena a la cual señala **s**. El carácter nulo de terminación se considera como parte de la cadena. La función **strrchr** devuelve un apuntador al carácter, o un apuntador nulo si **c** no está incluido en la cadena.

```
size_t strspn(const char *s1, const char *s2);
```

La función **strspn** calcula la longitud del segmento inicial máximo de la cadena a la cual señala **s1**, que consista en totalidad de caracteres de la cadena a la cual señala **s2**. La función **strspn** devuelve la longitud del segmento.

```
char *strstr(const char *s1, const char *s2);
```

La función **strstr** localiza la primera instancia en la cadena a la cual señala **s1** de la secuencia de caracteres (excluyendo el carácter nulo de terminación) en la cadena a la cual señala **s2**. La función **strstr** devuelve un apuntador a la cadena localizada, o un apuntador nulo si la cadena no se encuentra. Si **s2** señala a una cadena de longitud cero, la función devuelve **s1**.

```
char *strtok(char *s1, const char *s2);
```

Una secuencia de llamadas a función **strtok** divide la cadena a la cual señala **s1** en una secuencia de componentes léxicos, cada uno de los cuales queda delimitado por un carácter de la cadena a la cual señala **s2**. La primera llamada en la secuencia tiene como argumento **s1**, y es seguida por llamadas con un apuntador nulo como primer argumento. La cadena separadora a la cual señala **s2** puede cambiar de llamada a llamada.

La primera llamada en la secuencia busca en la cadena a la cual señala **s1** el primer carácter que no esté contenido en la cadena separadora actual a la cual señala **s2**. Si dicho carácter no se encuentra, entonces no existen componentes léxicos en la cadena a la cual señala **s1** y la función **strtok** devuelve un apuntador nulo. Si se encuentra dicho carácter, ese es el inicio del primer componente léxico.

La función **strtok** a continuación busca a partir de ahí un carácter que esté contenido en la cadena separadora actual. Si no encuentra dicho carácter, el componente léxico actual se extiende hasta el fin de la cadena a la cual señala **s1**, y las búsquedas subsecuentes para un componente léxico devolverán un apuntador nulo. Si se encuentra dicho carácter, queda sobreescrito por un carácter nulo, lo cual da por terminado el componente léxico actual. La función **strtok** guarda un apuntador al carácter siguiente, a partir del cual se iniciará el siguiente componente léxico.

Cada llamada subsiguiente, con un apuntador nulo como valor de su primer argumento, empieza a buscar a partir del apuntador guardado y se comporta como se describió anteriormente. La puesta en práctica debe comportarse como si ninguna función de biblioteca llama a la función `strtok`. La función `strtok` devuelve un apuntador al primer carácter de un componente léxico, o a un apuntador nulo si tal componente no existe.

`void *memset(void *s, int c, size_t n);`

La función `memset` copia el valor de `c` (convertido a un `unsigned char`) a cada uno de los primeros `n` caracteres en el objeto al cual señala `s`. La función `memset` devuelve el valor de `s`.

`char *strerror(int errnum);`

La función `strerror` relocaliza el número de error en `errnum` a una cadena de mensaje de error. La puesta en práctica deberá comportarse como si ninguna función de biblioteca llama a la función `strerror`. La función `strerror` devuelve un apuntador a la cadena, el contenido de la cual está definido por la puesta en práctica. El arreglo al cual se apunta no debe modificarse por el programa, que pudiera ser sobreescrito debido a una llamada subsiguiente a la función `strerror`.

`size_t strlen(const char *s);`

La función `strlen` computa la longitud de la cadena a la cual señala `s`. La función `strlen` devuelve el número de caracteres que anteceden al carácter nulo de terminación.

B.13 Fecha y hora <time.h>

`CLOCKS_PER_SEC`

El número por segundo del valor devuelto por la función `clock`.

`NULL`

Constante de apuntador nula, definida por la puesta en práctica.

`clock_t`

Un tipo aritmético capaz de representar la fecha.

`time_t`

Un tipo aritmético capaz de representar la hora.

`size_t`

Un tipo entero no signado del resultado del operador `sizeof`.

`struct tm`

Contiene los componentes de una hora de calendario, conocida como *hora desglosada*. La estructura deberá contener por lo menos los siguientes miembros, en cualquier orden. La semántica de los miembros y sus rangos normales se expresan en los comentarios.

```
int tm_sec;     /* seconds after the minute—[0,61] */
int tm_min;     /* minutes after the hour—[0,59] */
int tm_hour;    /* hours since midnight—[0,23] */
int tm_mday;    /* day of the month—[1,31] */
int tm_mon;     /* months since January—[0,11]*/
int tm_year;    /* year since 1900 *
int tm_wday;    /* days since Sunday—[0,6] */
int tm_yday;    /* days since January 1—[0,365] */
int tm_isdst;   /* Daylight Saving Time flag */
```


El valor `tm_isdst` es positivo si está en efecto Daylight Saving Time, es cero si no está en efecto Daylight Saving Time, y negativo si la información no está disponible.

`clock_t clock(void)`

La función `clock` determina la hora de procesador utilizada. La función `clock` devuelve la mejor aproximación de la puesta en práctica de la hora del procesador utilizada por el programa desde el principio de una era, definida por la puesta en práctica, únicamente relacionada con la invocación del programa. Para determinar la hora en segundos, el valor devuelto por la función `clock` deberá ser dividido por el valor de la macro `CLOCKS_PER_SEC`. Si la hora del procesador utilizada no está disponible o su valor no puede ser representado, la función devuelve el valor `(clock_t)-1`.

`double difftime(time_t time1, time_t time0);`

La función `difftime` calcula la diferencia entre dos horas de calendario: `time1 -time0`. La función `difftime` devuelve la diferencia, expresada en segundos, como un `double`.

`time_t mktime(struct tm *timeptr);`

La función `mktime` convierte la hora desglosada, expresada como hora local, en la estructura a la cual señala `timeptr` en un valor de hora de calendario, con la misma codificación que la de los valores devueltos por la función `time`. Serán ignorados los valores originales de los componentes `tm_wday` y `tm_yday` de la estructura, y los valores originales de los demás componentes no quedarán restringidos a los rangos arriba indicados. A la terminación exitosa, se ajustarán apropiadamente los valores de los componentes `tm_wday` y `tm_yday` y los demás componentes se ajustarán para representar la hora de calendario específico, pero con sus valores forzados a los rangos arriba indicados; el valor final de `tm_mday` no se define hasta que estén determinados `tm_mon` y `tm_year`. La función `mktime` devuelve la hora especificada de calendario codificada como un valor del tipo `time_t`. Si la hora de calendario no puede representarse, la función devuelve el valor `(time_t)-1`.

`time_t time(time_t *timer);`

La función `time` determina la hora de calendario actual. La función `time` devuelve la mejor aproximación según la puesta en práctica de la hora de calendario actual. El valor `(time_t)-1` será devuelto si dicha hora de calendario no está disponible. Si `timer` no es un apuntador nulo, el valor devuelto también se asignará al objeto al cual apunta.

`char *asctime(const struct tm *timeptr);`

La función `asctime` convierte la hora desglosada en la estructura a la cual señala `timeptr` en una cadena de la forma

```
Sun Sep 16 01:03:52 1973\n\0
```

La función `asctime` devuelve un apuntador a la cadena.

`char *ctime(const time_t *timer);`

La función `ctime` convierte la hora de calendario al cual señala `timer` a hora local en forma de una cadena. Es equivalente a

```
asctime(localtime(timer))
```

La función `ctime` devuelve el apuntador devuelto por la función `asctime` con dicha hora desglosada como argumento.

`struct tm *gmtime(const time_t *timer);`

La función `gmtime` convierte la hora de calendario al cual señala `timer` en una hora desglosada, expresada en forma de Coordinated Universal Time (UTC) La función `gmtime` devuelve un apuntador a dicho objeto, o un apuntador nulo si UTC no está disponible.

`struct tm *localtime(const time_t *timer);`

La función `localtime` convierte la hora de calendario al cual señala `timer` en una hora desglosada, expresada en forma de hora local. La función `localtime` devuelve un apuntador a dicho objeto.

```
size_t strftime(char *s, size-t maxsize, const char *format, const
                struct tm *timeptr);
```

La función **strftime** coloca caracteres en el arreglo al cual señala **s** controlados por la cadena a la cual señala **format**. La cadena **format** consiste de un cero o más especificadores de conversión y caracteres ordinarios de varios bytes. Todos los caracteres ordinarios (incluyendo el carácter nulo de terminación) serán copiados al arreglo sin modificarse. Si la copia se hace entre objetos que se superponen, el comportamiento quedará indefinido. No se colocarán más caracteres que los especificados en **maxsize** en el arreglo. Cada especificador de conversión será remplazado por los caracteres apropiados tal y como se describe en la lista siguiente. Los caracteres apropiados quedan determinados por la categoría **LC_TIME** del escenario actual y por los valores contenidos en la estructura a la cual señala **timeptr**.

- **%a** es reemplazado por la abreviatura del nombre del día de la semana del escenario.
- **%A** es reemplazado por el nombre completo del día de la semana del escenario.
- **%b** es reemplazado por la abreviatura del nombre del mes del escenario.
- **%B** es reemplazado por el nombre completo del mes del escenario.
- **%c** es reemplazado por la representación apropiada de fecha y de hora del escenario.
- **%d** es reemplazado por el día del mes como número decimal (**01-31**).
- **%H** es reemplazado por la hora (reloj de 24 horas) como número decimal (**00-23**).
- **%I** es reemplazado por la hora (reloj de 12 horas) como número decimal (**01-12**).
- **%j** es reemplazado por el día del año como número decimal (**001-366**).
- **%m** es reemplazado por el mes como número decimal (**01-12**).
- **%M** es reemplazado por el minuto como número decimal (**00-59**).
- **%p** es reemplazado por el equivalente para el escenario de las designaciones AM/PM asociadas con un reloj de 12 horas.
- **%S** es reemplazado como segundos como número decimal (**00-61**).
- **%U** es reemplazado por el número de la semana correspondiente del año (siendo el primer domingo el primer día de la semana 1) como número decimal (**00-53**).
- **%w** es reemplazado por el día de la semana como número decimal (**0-6**), donde domingo es **0**.
- **%W** es reemplazado por el número de la semana correspondiente del año (el primer lunes es el primer día de la semana 1) como número decimal (**00-53**).
- **%x** es reemplazado por la representación apropiada de la fecha, de acuerdo con el escenario.
- **%X** es reemplazado por la representación apropiada de la hora para el escenario.
- **%y** es reemplazado por el año, sin siglos, como un número decimal (**00-99**).
- **%Y** es reemplazado por el año, incluyendo siglos, como un número decimal.
- **%Z** es reemplazado por el nombre o abreviatura de la zona de tiempo, y mediante ningún carácter, si no hay determinable ninguna zona de tiempo.
- **%%** es reemplazado por **%**.

Si el especificador de conversión no es ninguno de los arriba citados, el comportamiento queda indefinido. Si el número total de caracteres resultantes, incluyendo el carácter nulo de terminación no es más de **max-size**, la función **strftime** devuelve el número de carácter colocados en el arreglo al cual señala **s**, sin incluir el carácter nulo de terminación. De lo contrario, se devuelve cero y el contenido del arreglo queda indeterminado.

B.14 Límites de puesta en práctica

<limits.h>

Los siguientes deben ser definidos en magnitud (valor absoluto) como igual a o mayor que los valores siguientes.

#define CHAR_BIT **8**
El número de bits para el objeto más pequeño que no sea un campo de bits (byte).

#define SCHAR_MIN **-127**
El valor mínimo para un objeto del tipo **signed char**.

#define SCHAR_MAX **+127**
El valor máximo para un objeto del tipo **signed char**.

#define UCHAR_MAX **255**
El valor máximo para un objeto del tipo **unsigned char**.

#define CHAR_MIN **0 ó SCHAR_MIN**
El valor mínimo para un objeto del tipo **char**.

#define CHAR_MAX **UCHAR_MAX ó SCHAR_MAX**
El valor máximo para un objeto del tipo **char**.

#define MB_LEN_MAX **1**
El número máximo de bytes de un carácter multibyte, para cualquier escenario soportado.

#define SHRT_MIN **-32767**
El valor mínimo para un objeto del tipo **short int**.

#define SHRT_MAX **+32767**
El valor máximo para un objeto del tipo **short int**.

#define USHRT_MAX **65535**
El valor máximo para un objeto del tipo **unsigned short int**.

#define INT_MIN **-32767**
El valor mínimo para un objeto del tipo **int**.

#define INT_MAX **+32767**
El valor máximo para un objeto del tipo **int**.

#define UINT_MAX **65535**
El valor máximo para un objeto del tipo **unsigned int**

#define LONG_MIN **-2147483647**
El valor mínimo para un objeto del tipo **long int**.

#define LONG_MAX **+2147483647**
El valor máximo para un objeto del tipo **long int**

#define ULONG_MAX **4294967295**
El valor máximo para un objeto del tipo **unsigned long int**

<float.h>

#define FLT_ROUNDS
El modo de redondeo para la suma en punto flotante

- **-1** indeterminable
- **0** hacia cero

1 hacia el más cercano

2 hacia infinito positivo

3 hacia infinito negativo

Lo siguiente debe ser definido en magnitud (valor absoluto) igual a o mayor que a los valores siguientes.

```
#define FLT_RADIX                    2
```

La raíz de la representación exponencial, *b*.

```
#define FLT_MANT_DIG
#define LDBL_MANT_DIG
#define DBL_MANT_DIG
```

El número de base **FLT_RADIX** dígitos en el significando del punto flotante *p*.

```
#define FLT_DIG                      6
#define DBL_DIG                     10
#define _LDB_DIG                    10
```

El número de dígitos decimales *q*, tal que cualquier número de punto flotante con *q* dígitos decimales pueda ser redondeado a un número de punto flotante con *p* raíz *b* dígitos y de regreso otra vez sin modificación a los dígitos decimales *q*.

```
#define FLT_MIN_EXP
#define DBL_MIN_EXP
#define LDBL_MIN_EXP
```

El entero negativo mínimo tal que **FLT_RADIX** elevado a dicha potencia menos 1 es un número de punto flotante normalizado.

```
#define FLT_MIN_10_EXP             -37
#define DBL_MIN_10_EXP             -37
#define LDBL_MIN_10_EXP            -37
```

El entero negativo mínimo tal que 10 elevado a dicha potencia está en el rango de los números de punto flotante normalizados.

```
#define FLT_MAX_EXP
#define DBL_MAX_EXP
#define LDBL_MAX_EXP
```

El entero máximo tal que **FLT_RADIX** elevado a dicha potencia menos 1 es un número finito de punto flotante representable.

```
#define FLT_MIN_10_EXP             +37
#define DBL_MIN_10_EXP             +37
#define LDBL_MIN_10_EXP            +37
```

El entero máximo tal que 10 elevado a esa potencia está en el rango de los números de punto flotante finitos y representables.

Lo siguiente deberá ser definido igual a o mayor que los valores mostrados a continuación.

```
#define FLT_MAX                    1E+37
#define DBL_MAX                    1E+37
#define LDBL_MAX                   1E+37
```

El número de punto flotante finito representable máximo.

Lo siguiente debe ser definido igual a o menor que los valores mostrados a continuación.

```
#define FLT_EPSILON                1E-5
```



```
#define DBL_EPSILON                1E-9
#define LDBL_EPSILON               1E-9
```

La diferencia entre 1.0 y el valor mínimo mayor que 1.0 que sea representable en el tipo dado de punto flotante.

```
#define FLT_MIN                    1E-37
#define DBL_MIN                    1E-37
#define LDBL_MIN                   1E-37
```

El número de punto flotante positivo normalizado mínimo.

Apéndice C

Precedencia y asociatividad de operadores

Operador	Asociatividad
() [] -> .	izquierda a derecha
++ — + - ! ~ (tipo) * & sizeof	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
<< >>	izquierda a derecha
< <= > >=	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha
^	izquierda a derecha
|	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
?:	derecha a izquierda
= += -= *= /= %= &= ^= |= <<= >>=	derecha a izquierda
,	izquierda a derecha

Los operadores aparecen de la parte superior a la inferior en orden decreciente de precedencia.

Apéndice D

Conjunto de caracteres ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Los dígitos a la izquierda de la tabla son los dígitos izquierdos del equivalente decimal del código de caracteres, y los dígitos en la parte superior de la tabla son los dígitos derechos del código de caracteres. Por ejemplo, el código de carácter para 'F' es 70, y el correspondiente para & es 38.

APÉNDICE E

Sistemas numéricos

Objetivos

- Comprender los conceptos básicos de los sistemas numéricos como base, valor posicional y valor simbólico.
- Comprender cómo trabajar con números representados en sistemas numéricos binario, octal, y hexadecimal.
- Ser capaz de reducir los números binarios a números octales o a números hexadecimales.
- Ser capaz de convertir los números octales y hexadecimales a números binarios.
- Ser capaz de convertir en ambos sentidos entre números decimales y sus equivalentes binario, octal y hexadecimal.
- Comprender la aritmética binaria y cómo se representan los números binarios negativos mediante la notación de complemento a dos.

He aquí sólo números ratificados.
William Shakespeare

La naturaleza tiene cierta clase de sistema de coordenadas geométrico aritméticas, porque la naturaleza tiene todo tipo de modelos. Lo que nosotros concebimos de la naturaleza está en los modelos y todos los modelos de la naturaleza son tan bellos. Se me ocurre que el sistema de la naturaleza debe ser una verdadera belleza, porque en química encontramos que las asociaciones se presentan siempre en bellos números enteros —no existen fracciones.
Richard Buckminster Fuller

Sinopsis

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios.

E.1 Introducción

En este apéndice, presentamos los sistemas numéricos clave utilizados por los programadores de C, en especial cuando están trabajando sobre proyectos de software que requieren una cercana interacción a "nivel máquina". Con el hardware proyectos como éstos incluyen sistemas operativos, software de redes de cómputo, compiladores, sistemas de base de datos y aplicaciones que requieran de alto rendimiento.

Cuando en un programa en C escribimos un entero como 227 o -63, se supone que el número está escrito en *sistema numérico decimal (de base 10)*. En el sistema numérico decimal los dígitos son 0, 1, 2, 3, 4, 5, 6, 7, 8, y 9. El dígito menor es 0 y el mayor es 9 —uno menos que la *base*, que es 10. En forma interna, las computadoras utilizan el *sistema numérico binario (de base 2)*. El sistema numérico binario utiliza únicamente dos dígitos, es decir 0 y 1. Su dígito menor es 0 y su dígito mayor es 1 —uno menos que la base de 2.

Como veremos, los números binarios tienden a ser mucho más largos que sus equivalentes decimales. Los programadores que tienen que trabajar en lenguajes de ensambladores, y en lenguajes de alto nivel como C que permiten a los programadores llegar hasta el "nivel máquina", por lo que otros dos sistemas numéricos se han hecho populares el *sistema numérico octal (de base 8) y el sistema numérico hexadecimal (de base 16)* primordialmente debido a que resultan convenientes para abreviar números binarios.

En el sistema numérico octal, los dígitos tienen un rango del 0 al 7. Dado que tanto el sistema numérico binario como el sistema numérico octal tienen menos dígitos que el sistema numérico decimal, sus dígitos son los mismos que los correspondientes dígitos en decimal.

El sistema numérico hexadecimal presenta un problema, porque requiere de dieciséis dígitos —un dígito menor 0 y un dígito mayor, con un valor equivalente al 15 decimal (uno menos que la base de 16). Por regla convencional, utilizamos las letras A hasta F para representar los dígitos hexadecimales que corresponden a los valores decimales del 10 hasta el 15. Por lo tanto en hexadecimal podemos tener números como 876, que están formados únicamente por dígitos parecidos a los decimales, números como 8A55F, que están formados de dígitos y de letras, y números como FFE, que sólo están formados por letras. Ocasionalmente, un número hexadecimal deletrea una palabra común, como FACE, o bien FEED y esto pudiera parecer extraño a aquellos programadores acostumbrados a trabajar solo con números.

Cada uno de estos sistemas numéricos utilizan *notación posicional* —la posición en la cual se escribe un dígito tiene un *valor posicional diferente*. Por ejemplo, en el número decimal 937 (el 9, el 3 y el 7 se conocen como *valores simbólicos*), decimos que el 7 está escrito en la *posición de las unidades*, el 3 está escrito en la *posición de las decenas*, y el 9 está escrito en la *posición de las centenas*. Note que cada una de estas posiciones es una potencia de la base (base 10), y estas potencias empiezan en 0 y se incrementan en 1 conforme nos movemos hacia la izquierda dentro del número (figura E.3).

Dígito binario	Dígito octal	Dígito decimal	Dígito hexadecimal
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9
			A (valor decimal 10)
			B (valor decimal 11)
			C (valor decimal 12)
			D (valor decimal 13)
			E (valor decimal 14)
			F (valor decimal 15)

Fig. E.1 Dígitos de los sistemas numéricos binario, octal, decimal y hexadecimal.

Atributo	Binario	Octal	Decimal	Hexadecimal
Base	2	8	10	16
Dígito menor	0	0	0	0
Dígito mayor	1	7	9	F

Fig. E.2 Comparación de los sistemas numéricos binario, octal, decimal y hexadecimal.

Valores posiciones en el sistema numérico decimal			
Dígito decimal	9	3	7
Nombre de la posición	Centenas	Decenas	Unidades
Valor posicional	100	10	1
Valor posicional como una potencia de la base (10)	10^2	10^1	10^0

Fig. E.3 Valores posicionales en el sistema numérico decimal.

Para números decimales más grandes, las siguientes posiciones a la izquierda serían la *posición de los millares* (10 a la tercera potencia), la *posición de los diez millares* (10 a la cuarta potencia), la *posición de los cien millares* (10 a la quinta potencia), la *posición de los millones* (10 a la sexta potencia), la *posición de los diez millones* (10 a la séptima potencia), y así en lo sucesivo.

En el número binario 101, decimos que el 1 más a la derecha está escrito en la *posición de las unidades*, el 0 está escrito en la *posición de los dos*, y el 1 más a la izquierda está escrito en la *posición de los cuatros*. Note que cada una de estas posiciones es una potencia de la base (base 2), y que estas potencias empiezan en 0 y se incrementan por 1 conforme nos movemos a la izquierda en el número (figura E.4).

Para números binarios más grandes, la siguiente posición a la izquierda sería la *posición de los ochos* (2 a la tercera potencia), la *posición de los diez y seis* (2 a la cuarta potencia), la *posición de los treinta y dos* (2 a la quinta potencia), la *posición de los sesenta y cuatro* (2 a la sexta potencia), y así en lo sucesivo.

En el número octal 425, decimos que el 5 está escrito en la *posición de las unidades*, el 2 está escrito en la *posición de los ochos*, y el 4 está escrito en la *posición de los sesenta y cuatros*. Note que cada una de estas posiciones es una potencia de la base (base 8), y que estas potencias empiezan en 0 y se incrementan en 1 conforme vamos hacia la izquierda en el número (figura E. 5).

Para números octales más grandes, las siguientes posiciones a la izquierda serían la *posición de los quinientos doces* (8 a la tercera potencia), la *posición de los cuatro mil noventa y seis* (8 a la cuarta potencia), la *posición de los treinta y dos mil setecientos y sesenta y ochos* (8 a la quinta potencia), y así en lo sucesivo.

En el número hexadecimal 3DA, decimos que la A está escrita en la *posición de las unidades*, la D está escrita en la *posición de los diez y seis*, y el 3 está escrito en la *posición de los doscientos cincuenta y seis*. Note que cada una de estas posiciones es una potencia de la base (base 16) y que estas potencias empiezan en 0 y se incrementan en 1 conforme vamos hacia la izquierda en el número (figura E.6).

Valores posicionales en el sistema numérico binario

Dígito binario	1	0	1
Nombre de la posición	Cuatros	Dos	Unidades
Valor posicional	4	2	1
Valor posicional como una potencia de la base (2)	2^2	2^1	2^0

Fig. E.4 Valores posicionales en el sistema numérico binario.

Valores posicionales en el sistema numérico octal

Dígito octal	4	2	5
Nombre de la posición	Sesenta y cuatros	Ochos	Unidades
Valor posicional	64	8	1
Valor posicional como una potencia de la base (8)	8^2	8^1	8^0

Fig. E.5 Valores posicionales en el sistema numérico octal.

Valores posicionales en el sistema numérico hexadecimal

Dígito hexadecimal	3	D	A
Nombre de la posición	Doscientos cincuenta y seis	Diez y seis	Unidades
Valor posicional	256	16	1
Valor posicional como una potencia de la base (16)	16^2	16^1	16^0

Fig. E.6 Valores posicionales en el sistema numérico hexadecimal.

Para números hexadecimales mayores, la siguientes posiciones a la izquierda serían la *posición cuatro mil noventa y seis* (16 a la tercera potencia), la *posición treinta y dos mil setecientos y sesenta y ochos* (16 a la cuarta potencia), y así sucesivamente.

E.2 Cómo abreviar números binarios como octales y hexadecimales

El uso principal para los números octales y hexadecimales en la computación es para abreviar representaciones binarias largas. En la figura E.7 se destaca el hecho que números binarios largos pueden ser expresados en forma concisa en sistemas numéricos con bases más altas que el sistema numérico binario.

Número decimal	Representación binaria	Representación octal	Representación hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Fig. E.7 Equivalentes decimal, binario, octal y hexadecimal.

Una relación particular importante que tienen tanto el sistema numérico octal como el hexadecimal con el sistema binario, es que las bases de los sistemas octal y hexadecimal (8 y 16 respectivamente), son potencias de la base del sistema numérico binario (base 2). Considere el siguiente número binario de 12 dígitos y sus equivalentes octal y hexadecimal. Vea si puede determinar cómo resulta conveniente esta relación para abreviar números binarios a octal y hexadecimal. La respuesta sigue a los números.

Número binario	Equivalente octal	Equivalente hexadecimal
100011010001	4321	8D1

Para ver como los números binarios se convierten con fácilidad a octal, sólo divida el número binario de 12 dígitos en grupos cada uno de ellos de tres bits consecutivos, y escriba dichos grupos sobre los dígitos correspondientes del número octal como sigue

```
100     011     010     001
4       3       2       1
```

Note que el dígito octal que usted ha escrito bajo cada número de estos tres bits corresponde precisamente al equivalente octal del número binario de tres dígitos, tal y como se muestra en la figura E.7.

El mismo tipo de relación se puede observar en la conversión de números de binario a hexadecimal. En particular divida el número binario de 12 dígitos en grupos de cuatro bits consecutivos cada uno y escriba estos grupos sobre los dígitos correspondientes al número hexadecimal como sigue

```
1000    1101    0001
8       D       1
```

Note que el dígito hexadecimal que ha escrito bajo cada grupo de cuatro bits corresponde precisamente al equivalente hexadecimal de dicho número binario de cuatro dígitos, tal y como se muestra en la figura E.7.

E.3 Cómo convertir números octales y hexadecimales a binarios

En la sección anterior, vimos cómo convertir números binarios a sus equivalentes octal y hexadecimal formando grupos de dígitos binarios y sólo reescribiendo estos grupos en sus valores digitales octales o hexadecimales. Este proceso pude ser utilizado en reversa para producir el equivalente binario de un número octal o hexadecimal dado.

Por ejemplo, el número octal 653 se convierte a binario simplemente escribiendo el 6 como su equivalente binario de 3 dígitos 110, el 5 como su equivalente binario de 3 dígitos 101, y el 3 como su equivalente binario de 3 dígitos 011 para formar el número binario de 9 dígitos 110101011.

El número hexadecimal FAD5 se convierte a binario simplemente escribiendo la F como su equivalente binario de 4 dígitos 1111, la A como su equivalente binario de 4 dígitos 1010, la D como su equivalente binario de 4 dígitos 1101, y el 5 como su equivalente binario de 4 dígitos 0101 para formar el número de 16 dígitos 1111101011010101.

E.4 Cómo convertir de binario, octal y hexadecimal a decimal

Dado que estamos acostumbrados a escribir en decimal, a menudo resulta conveniente convertir un número binario, octal o hexadecimal a decimal para darnos cuenta de lo que "realmente" vale ese número. Nuestros diagramas en la sección E.1 expresan los valores posicionales en decimal. Para convertir un número de decimal a otra base, multiplique el equivalente decimal de cada dígito por su valor posicional, y sume dichos productos. Por ejemplo, el número binario 110101 se convierte al decimal 53 tal y como se muestra en la figura E.8.

Cómo convertir un número binario a decimal

Valores posicionales:	32	16	8	4	2	1
Valores simbólicos:	1	1	0	1	0	1
Productos:	1*32=32	1*16=16	0*8=0	1*4=4	0*2=0	1*1=1
Suma:	= 32 + 16 + 0 + 4 + 0 + 1 = 53					

Fig. E.8 Cómo convertir un número binario a decimal.

Para convertir el octal 7614 al decimal 3980, utilizamos la misma técnica, esta vez utilizando los valores posicionales octales apropiados, tal y como se muestra en la figura E.9.

Para convertir el valor hexadecimal AD3B al decimal 44347, utilizamos la misma técnica, esta vez utilizando los valores posicionales hexadecimales apropiados, como se muestra en la figura E.10.

E.5 Cómo convertir de decimal a binario, octal o hexadecimal

Las conversiones de la sección anterior son una consecuencia natural de las reglas convencionales de la notación posicional. La conversión de decimal a binario, octal o hexadecimal también sigue estas reglas convencionales.

Suponga que deseamos convertir el decimal 57 a binario. Empezamos escribiendo los valores posicionales de las columnas derecha a la izquierda hasta que alcanzamos una columna cuyo valor posicional es mayor que el número decimal. No necesitamos esta columna, por lo que la descartamos. Por lo tanto primero escribimos:

Cómo convertir un número octal a decimal

Valores posicionales:	512	64	8	1
Valores simbólicos:	7	6	1	4
Productos:	7*512=3584	6*64=384	1*8=8	4*1=4
Suma:	= 3584 + 384 + 8 + 4 = 3980			

Fig. E.9 Cómo convertir un número octal a decimal.

Cómo convertir un número hexadecimal a decimal

Valores posicionales:	4096	256	16	1
Valores simbólicos:	A	D	3	B
Productos:	A*4096=40960	D*256=3328	3*16=48	B*1=11
Suma:	= 40960 + 3328 + 48 + 11 = 44347			

Fig. E.10 Cómo convertir un número hexadecimal a decimal.

```
Valores posicionales:   64   32   16   8   4   2   1
```

A continuación descartamos la columna con el valor posicional 64 dejando:

```
Valores posicionales:       32   16   8   4   2   1
```

A continuación trabajamos partiendo de la columna más a la izquierda hacia la derecha. Dividimos 57 entre 32 y observamos que el resultado es 1 con un residuo de 25, por lo que en la columna de los 32 escribimos 1. Ahora dividimos 25 entre 16 y observamos que 25 entre 16 da 1 con un residuo de 9 y escribimos 1 en la columna de los 16. Ahora dividimos 9 entre 8 y observamos que 9 entre 8 da 1 con un residuo de 1. Las siguientes dos columnas cada una de ellas produce resultados de cero cuando dividimos 1 entre sus valores posicionales, por lo que escribimos ceros en las columnas 4 y 2. Finalmente 1 dividido entre 1 es 1, por lo que escribimos 1 en la columna de los unos. Esto da como resultado:

```
Valores posicionales:       32   16   8   4   2   1
Valores simbólicos          1    1    1   0   0   1
```

y, por lo tanto, el equivalente del decimal 57 en binario es 111001

Para convertir el decimal 103 a octal, empezamos escribiendo los valores posicionales de las columnas hasta que alcanzamos una columna cuyo valor posicional es mayor que el número decimal. No necesitamos de esa columna, por lo que la descartamos. Por lo tanto primero escribimos:

```
Valores posicionales:   512   64   8   1
```

A continuación descartamos la columna con el valor posicional 512, dando por resultado:

```
Valores posicionales:       64   8   1
```

A continuación trabajamos a partir de la columna más a la izquierda y hacia la derecha. Dividimos 103 entre 64 y observamos que 103 entre 64 da a uno con un residuo de 39, por lo que escribimos 1 en la columna de los 64. A continuación dividimos 39 entre 8 y observamos que el resultado es 4 con un residuo de 7, y escribimos 4 en la columna de los 8. Por último dividimos 7 entre 1 y observamos que el resultado es 7 sin residuo, por lo que escribimos 7 en la columna de los unos. Esto da como resultado:

```
Valores posicionales:       64   8   1
Valores simbólicos          1    4   7
```

y, por lo tanto, el decimal 103 es equivalente al octal 147.

Para convertir el decimal 375 a hexadecimal, empezamos escribiendo los valores posicionales de las columnas hasta que llegamos a una columna cuyo valor posicional es mayor que el número decimal. No necesitamos dicha columna, por lo que la descartamos. Entonces, primero escribimos

```
Valores posicionales:   4096 256   16   1
```

A continuación descartamos la columna con el valor posicional 4096, dando por resultado:

```
Valores posicionales:       256   16   1
```

A continuación trabajamos a partir de la columna más a la izquierda hacia la derecha. Dividimos 375 entre 256 y observamos que el resultado es uno con un residuo de 119, por lo que escribimos 1 en la columna de los 256. Ahora dividimos 119 entre 16 y observamos que el resultado es 7 con un residuo de 7 y escribimos 7 en la columna de los 16. Por último dividimos 7 entre 1 y observamos que existe un resultado de 7 sin residuo, por lo que escribimos 7 en la columna de los unos. Esto da como resultado:


```
Valores posicionales:       256   16   1
Valores simbólicos          1     7    7
```

por lo tanto, el equivalente del decimal 375 en hexadecimal es 177.

E.6 Números binarios negativos: notación de complemento a dos

El análisis en este apéndice ha sido enfocado a números positivos. En esta sección, explicarémos cómo representan las computadoras números negativos mediante la *notación de complemento a dos*. Primero explicaremos cómo se forma el complemento a dos de un número binario, y a continuación mostraremos porqué representa el valor negativo de un número binario dado.

Considere una máquina con enteros de 32 bits. Suponga

```
int value = 13;
```

La representación en 32 bits de `value` es

```
00000000 00000000 00000000 00001101
```

Para formar el negativo de `value`, primero formamos su complemento a uno aplicando el operador de complemento a nivel de bits (~)de C.

```
ones_complement_of_value = ~value;
```

Internamente, `~value` es ahora `value` con cada uno de sus bits invertidos —los unos se han convertido en ceros y los ceros se han convertido en unos, como sigue:

```
value:
00000000 00000000 00000000 00001101

~value (es decir, el complemento a uno de value):
11111111 11111111 11111111 11110010
```

Para formar el complemento a dos de `value`, simplemente añadimos uno al complemento a uno de `value`. Por lo tanto

```
El complemento a dos de value:
11111111 11111111 11111111 11110011
```

Ahora si esto de hecho es igual a -13, deberíamos estar en condiciones de añadirlo al número binario 13 y obtener como resultado 0. Probémoslo:

```
 00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
-----------------------------------
 00000000 00000000 00000000 00000000
```

Se descarta el bit de acarreo que resulta en la columna más a la izquierda y como resultado obtenemos cero. Si a un número le añadimos el complemento a uno del número, el resultado serían todos unos. La clave para obtener un resultado de todos ceros es que el complemento a dos es uno más que el complemento a uno. La adición de uno hace que cada columna añada a cero, con un uno que se acarrea a la izquierda. El acarreo se va trasladando hacia la izquierda hasta que se descarta del bit más a la izquierda, y de ahí el número resultante de todos ceros.

En realidad las computadoras llevan a cabo una substracción como

```
x = a - value;
```

sumando el complemento a dos de **value** a **a** como sigue:

```
x = a + (~value + 1);
```

Suponga que **a** es 27 y **value** es 13, como antes. Si el complemento a dos de **value** es de hecho el valor negativo de **value**, entonces la suma del complemento a dos de **value** a **a** debería producir el resultado 14. Probémoslo:

```
a (es decir, 27)     00000000 00000000 00000000 00011011
+(~value + 1)       +11111111 11111111 11111111 11110011
                    -----------------------------------
                     00000000 00000000 00000000 00001110
```

lo que realmente da igual a 14.

Resumen

- Cuando escribimos un entero como 19 y 227 ó -63 en un programa C, automáticamente se supone que el número está en sistema numérico decimal (en base 10). Los dígitos en sistema numérico decimal son 0, 1, 2, 3, 4, 5, 6, 7, 8, y 9. El dígito menor es cero y el dígito mayor es 9 —uno menos que la base de 10.
- En forma interna, las computadoras utilizan el sistema numérico binario (de base 2). El sistema numérico binario tiene sólo dos dígitos, es decir 0 y 1. Su número menor es 0 y su número mayor es 1 —uno menos que el de la base de 2.
- El sistema numérico octal (de base 8) y el sistema numérico hexadecimal (de base 16) se han hecho populares principalmente debido a que resultan convenientes para abreviar números binarios.
- Los dígitos del sistema numérico octal van desde el 0 al 7.
- El sistema numérico hexadecimal presenta un problema, porque requiere de 16 dígitos —un dígito menor 0 y un dígito mayor con un valor equivalente al 15 decimal (uno menos que la base de 16). Por regla convencional, utilizamos las letras A hasta F para representar los dígitos hexadecimales que corresponden a los valores decimales 10 hasta el 15.
- Cada sistema numérico utiliza notación posicional —cada posición en la cual se escribe un dígito, tiene un valor posicional diferente.
- Una relación particularmente importante que tienen tanto el sistema numérico octal como el hexadecimal con el sistema binario, es que las bases de los sistemas octal y hexadecimal (8 y 16 respectivamente) son potencias de la base del sistema numérico binario (base 2).
- Para convertir un número octal a un número binario, simplemente remplace cada dígito octal por su equivalente binario de tres dígitos.
- Para convertir un número hexadecimal a un número binario, simplemente remplace cada dígito hexadecimal por su equivalente binario de cuatro dígitos.
- Dado que estamos acostumbrados a trabajar en decimal, a menudo resulta conveniente convertir un número binario, octal o hexadecimal a decimal para tener una mejor idea de lo que en verdad vale un número.
- Para convertir un número de decimal desde otra base, multiplique el equivalente decimal de cada dígito por su valor posicional y sume dichos productos.
- Las computadoras representan los números negativos utilizando notación de complemento a dos.

- Para formar el negativo de un valor en binario, primero forme el complemento a uno aplicando el operador de complemento a nivel de bits (~) de C. Esto invierte los bits del valor. Para formar el complemento a dos de dicho valor, sólo añada uno al valor del complemento a uno.

Terminología

base
sistema numérico de base 2
sistema numérico de base 8
sistema numérico de base 10
sistema numérico de base 16
sistema numérico binario
operador de complemento a nivel de bits (~)
conversiones
sistema numérico decimal
dígito
sistema numérico hexadecimal
valor negativo
sistema numérico octal
notación de complemento a uno
notación posicional
valor posicional
valor simbólico
notación de complemento a dos

Ejercicios de autoevaluación

E.1 Las bases de los sistemas numéricos decimal, binario, oct y hexadecimal son _____, _____, _____, y _____ respectivamente.

E.2 En general, las representaciones decimal, octal y hexadecimal de un número binario dado contienen (más/menos) dígitos que los que contiene el número binario.

E.3 (Verdadero/falso). Una razón popular para utilizar el sistema numérico decimal es que forma una notación conveniente para abreviar números binarios simplemente sustituyendo un dígito decimal por cada grupo de cuatro dígitos binarios.

E.4 La representación (octal/hexadecimal/decimal) de un valor binario muy grande es la más concisa (de las alternativas dadas).

E.5 (Verdadero/falso). El dígito mayor en cualquier base es uno más que la base.

E.6 (Verdadero/falso). El dígito menor de cualquier base es uno menos que la base.

E.7 El valor posicional del dígito más a la derecha de cualquier número en binario, octal, decimal o hexadecimal es siempre _____.

E.8 El valor posicional del dígito a la izquierda del dígito más a la derecha de cualquier número en binario, octal, decimal o hexadecimal es siempre igual a _____.

E.9 Complete los valores faltantes en esta tabla de valores posicionales para las cuatro posiciones más a la derecha en cada uno de los sistema numéricos indicados.

decimal	1000	100	10	1
hexadecimal	...	256	...	...
binario	...	...	...	...
octal	512	...	8	...

E.10 Convertir el número binario `110101011000` a octal y a hexadecimal.

E.11 Convertir el número hexadecimal **FACE** a binario.

E.12 Convertir el número octal `7316` a binario.

E.13 Convertir el número hexadecimal `4FEC` a octal. (*Sugerencia:* primero convierta `4FEC` a binario y a continuación convierta el número binario a octal).)

E.14 Convierta el número binario `1101110` a decimal.

E.15 Convierta el número octal `317` a decimal.

E.16 Convierta el número hexadecimal `EFD4` a decimal.

E.17 Convierta el número decimal `177` a binario, a octal, y a hexadecimal.

E.18 Muestre la representación binaria del decimal `417`. Después muestre el complemento a uno de `417` y el complemento a dos.

E.19 ¿Cuál es el resultado cuando el complemento a uno de un número se añade a sí mismo?

Respuestas a los ejercicios de autoevaluación

E.1 `10, 2, 8, 16.`

E.2 Menos.

E.3 Falso.

E.4 Hexadecimal.

E.5 Falso —el dígito mayor en cualquier base es uno menos que la base.

E.6 Falso —el dígito menor de cualquier base es cero.

E.7 1 (la base elevada a la potencia cero).

E.8 La base del sistema numérico.

E.9 Complete los valores faltantes en esta tabla de valores posicionales para las cuatro posiciones más a la derecha en cada uno de los sistemas numéricos indicados.

decimal	1000	100	10	1
hexadecimal	4096	256	16	1
binario	8	4	2	1
octal	512	64	8	1

E.10 Octal `6530`; hexadecimal `D58`.

E.11 Binario `1111 1010 1100 1110`.

E.12 Binario `111 011 001 110`

E.13 Binario `0 100 111 111 101 100`; Octal `47754`

E.14 Decimal `2+4+8+32+64=110`.

E.15 Decimal `7+1*8+3*64=7+8+192=207`

E.16 Decimal `4+13*16+15*256+14*4096=61396`.

E.17 Decimal `177`

a binario:

```
256 128 64 32 16 8 4 2 1
128 64 32 16 8 4 2 1
(1*128)+(0*64)+(1*32)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1)
10110001
```


a octal:

```
512 64 8 1
64 8 1
(2*64)+(6*8)+(1*1)
261
```

a hexadecimal:

```
256 16 1
16 1
(11*16)+(1*1)
(B*16)+(1*1)
B1
```

E.18 Binario:

```
512 256 128 64 32 16 8 4 2 1
256 128 64 32 16 8 4 2 1
(1*256)+(1*128)+(0*64)+(1*32)+(0*16)+(0*8)+(0*4)+(0*2)+
(1*1)
110100001
```

Complemento a uno:`001011110`
Complemento a dos:`001011111`
Verificación: número binario original + su complemento a dos

```
110100001
001011111
---------
000000000
```

E.19 Cero

Ejercicios

E.20 Algunas personas alegan que muchos de nuestros cálculos serían más fáciles en un sistema numérico de base `12` porque `12` es divisible por muchísimos más números que `10` (debido a la base `10`). ¿Cuál es el dígito menor en la base `12`? ¿Cuál debería ser el símbolo más alto para el dígito en la base `12`? ¿Cuáles son los valores posicionales para las cuatro posiciones más a la derecha de cualquier número en un sistema numérico de base `12`?

E.21 ¿Cómo se relaciona el valor simbólico más alto en los sistemas numéricos anteriormente analizados con el valor posicional del primer dígito a la izquierda del dígito más a la derecha de cualquier número en estos sistemas numéricos?

E.22 Complete la tabla siguiente de valores posicionales para las cuatro posiciones más a la derecha en cada uno de los sistemas numéricos indicados:

decimal	1000	100	10	1
base 6	...	...	6	...
base 13	...	169	...	...
base 3	27	...	...	...

E.23 Convierta el binario `100101111010` a octal y a hexadecimal.

E.24 Convierta el hexadecimal a `3A7D` a binario.

E.25 Convierta el hexadecimal a `765F` a octal. (*Sugerencia*: primero convierta `765F` a binario y a continuación convierta dicho número binario a octal).

E.26 Convierta el número binario `1011110` a decimal.

E.27 Convierta el número octal `426` a decimal.

E.28 Convierta el número hexadecimal `FFFF` a decimal.

E.29 Convierta el número decimal `299` a binario, a octal y a hexadecimal.

E.30 Muestre la representación binario del decimal `779`. A continuación muestre el complemento a uno de `779`, así como el complemento a dos.

E.31 ¿Cuál es el resultado cuando el complemento a dos de un número se añade a sí mismo?.

E.32 Muestre el complemento a dos del valor entero `-1` en una máquina con enteros de 32 bits.

INDICE

Indice

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W