

Introducción

A la Programación Lógica y Diseño

JOYCE FARRELL



7a. Ed.

Introducción a la Programación Lógica y Diseño

7a. Ed.

Introducción a la Programación Lógica y Diseño

JOYCE FARRELL

Traductor:

Jorge Alberto Velázquez Arellano

Revisión técnica:

Fabiola Ocampo Botello

José Sánchez Juárez

Roberto De Luna Caballero

Profesores de la Escuela Superior de Cómputo

Instituto Politécnico Nacional



**Introducción a la Programación Lógica
y Diseño**

7a. Ed.

Joyce Farrell

**Presidente de Cengage Learning
Latinoamérica:**

Fernando Valenzuela Migoya

**Director editorial, de producción
y de plataformas digitales para
Latinoamérica:**

Ricardo H. Rodríguez

Gerente de procesos para Latinoamérica:

Claudia Islas Licona

**Gerente de manufactura para
Latinoamérica:**

Raúl D. Zendejas Espejel

**Gerente editorial de contenidos
en español:**

Pilar Hernández Santamarina

Gerente de proyectos especiales:

Luciana Rabuffetti

Coordinador de manufactura:

Rafael Pérez González

Editores:

Ivonne Arciniega Torres

Timoteo Eliosa García

Diseño de portada:

Lisa Kuhn/Curio Press, LLC,

www.curиopress.com**Imagen de portada:**

© Leigh Prather/Veer

Composición tipográfica:

Rogelio Raymundo Reyna Reynoso

© D.R. 2013 por Cengage Learning Editores, S.A. de C.V., una Compañía de Cengage Learning, Inc. Corporativo Santa Fe Av. Santa Fe núm. 505, piso 12 Col. Cruz Manca, Santa Fe C.P. 05349, México, D.F. Cengage Learning® es una marca registrada usada bajo permiso.

DERECHOS RESERVADOS. Ninguna parte de este trabajo amparado por la Ley Federal del Derecho de Autor, podrá ser reproducida, transmitida, almacenada o utilizada en cualquier forma o por cualquier medio, ya sea gráfico, electrónico o mecánico, incluyendo, pero sin limitarse a lo siguiente: fotocopiado, reproducción, escaneo, digitalización, grabación en audio, distribución en Internet, distribución en redes de información o almacenamiento y recopilación en sistemas de información a excepción de lo permitido en el Capítulo III, Artículo 27 de la Ley Federal del Derecho de Autor, sin el consentimiento por escrito de la Editorial.

Traducido del libro *Programming Logic and Design, Introductory version*
Seventh edition
Joyce Farrell
Publicado en inglés por Course Technology, una
compañía de Cengage Learning © 2013
ISBN: 978-1-133-52651-3

Datos para catalogación bibliográfica:
Farrell, Joyce
Introducción a la Programación Lógica y Diseño,
7a. Ed.
ISBN: 978-607-481-905-2

Visite nuestro sitio en:
<http://latinoamerica.cengage.com>

Contenido

v

Prefacio	xii
CAPÍTULO 1	
Una revisión de las computadoras y la programación.	1
Comprensión de los sistemas de cómputo	2
Comprensión de la lógica de programa simple	5
Comprensión del ciclo de desarrollo del programa	7
Entender el problema	8
Planear la lógica	9
Codificación del programa	10
Uso de software para traducir el programa al lenguaje de máquina	10
Prueba del programa	12
Poner el programa en producción	13
Mantenimiento del programa	13
Uso de declaraciones en seudocódigo y símbolos de diagrama de flujo	14
Escritura en seudocódigo	15
Trazo de diagramas de flujo	16
Repetición de las instrucciones	17
Uso de un valor centinela para terminar un programa	19
Comprensión de la programación y los ambientes del usuario	22
Comprensión de los ambientes de programación	22
Comprensión de los ambientes de usuario	24
Comprensión de la evolución de los modelos de programación	25
Resumen del capítulo	27
Términos clave	28
Preguntas de repaso	31
Ejercicios	33
Encuentre los errores	35
Zona de juegos	35
Para discusión	36

CAPÍTULO 2	Elementos de los programas de alta calidad	37
	La declaración y el uso de variables y constantes	38
	Comprensión de las constantes literales y sus tipos de datos	38
	Trabajo con variables	39
	Nombramiento de variables	41
	Asignación de valores a las variables	42
	Comprensión de los tipos de datos de las variables	43
	Declaración de constantes nombradas	44
	Realización de operaciones aritméticas	45
	Comprensión de las ventajas de la modularización	48
	La modularización proporciona abstracción	49
	La modularización permite que varios programadores trabajen en un problema	50
	La modularización permite que se reutilice el trabajo	50
	Modularización de un programa	51
	Declaración de variables y constantes dentro de los módulos	55
	Comprensión de la configuración más común para la lógica de línea principal	57
	Creación de gráficas de jerarquía	61
	Características de un buen diseño de programa	63
	Uso de comentarios del programa	64
	Elección de identificadores	66
	Diseño de declaraciones precisas	68
	Evite cortes de línea confusos	68
	Escritura de indicadores claros y entradas con eco	69
	Mantener buenos hábitos de programación	71
	Resumen del capítulo	72
	Términos clave	73
	Preguntas de repaso	76
	Ejercicios	79
	Encuentre los errores	81
	Zona de juegos	82
	Para discusión	82
CAPÍTULO 3	Comprender la estructura	83
	Las desventajas del código espagueti no estructurado	84
	Comprensión de las tres estructuras básicas	86
	Uso de una entrada anticipada para estructurar un programa	95
	Comprensión de las razones para la estructura	101

Reconocimiento de la estructura	102
Estructuración y modularización de la lógica	
no estructurada	105
Resumen del capítulo	110
Términos clave	111
Preguntas de repaso.	112
Ejercicios	114
Encuentre los errores	118
Zona de juegos	118
Para discusión	119
CAPÍTULO 4 Toma de decisiones	121
Expresiones booleanas y la estructura de selección.	122
Uso de operadores de comparación relacionales	126
Evitar un error común con los operadores relacionales .	129
Comprensión de la lógica AND	129
Anidar decisiones AND para la eficiencia	132
Uso del operador AND	134
Evitar errores comunes en una selección AND	136
Comprensión de la lógica OR	138
Escritura de decisiones OR para eficiencia	140
Uso del operador OR	141
Evitar errores comunes en una selección OR	143
Hacer selecciones dentro de rangos.	148
Evitar errores comunes cuando se usan comprobaciones de rango	150
Comprensión de la precedencia cuando se combinan operadores AND y OR	154
Resumen del capítulo	157
Términos clave	158
Preguntas de repaso.	159
Ejercicios	162
Encuentre los errores	167
Zona de juegos.	167
Para discusión	168
CAPÍTULO 5 Creación de ciclos	169
Comprensión de las ventajas de crear ciclos	170
Uso de una variable de control de ciclo	171
Uso de un ciclo definido con un contador	172
Uso de un ciclo indefinido con un valor centinela	173
Comprensión del ciclo en la lógica de línea principal de un programa	175

Ciclos anidados	177
Evitar errores comunes en los ciclos	183
Error: descuidar la inicialización de la variable de control de ciclo	183
Error: descuidar la alteración de la variable de control de ciclo	185
Error: usar la comparación errónea con la variable de control de ciclo	186
Error: incluir dentro del ciclo declaraciones que pertenecen al exterior del mismo	187
Uso de un ciclo for	192
Aplicaciones comunes de los ciclos	194
Uso de un ciclo para acumular totales	194
Uso de un ciclo para validar datos	198
Limitación de un ciclo que pide entradas de nuevo	200
Validación de un tipo de datos	202
Validación de la sensatez y consistencia de los datos	203
Resumen del capítulo	205
Términos clave	205
Preguntas de repaso	206
Ejercicios	209
Encuentre los errores	211
Zona de juegos	211
Para discusión	212
CAPÍTULO 6 Arreglos	213
Almacenamiento de datos en arreglos	214
De qué modo los arreglos ocupan la memoria de la computadora	214
Cómo un arreglo puede reemplazar decisiones anidadas	216
Uso de constantes con arreglos	224
Uso de una constante como el tamaño de un arreglo	224
Uso de constantes como valores de elemento del arreglo	225
Uso de una constante como subíndice de un arreglo	225
Búsqueda de un arreglo para una correspondencia exacta	226
Uso de arreglos paralelos	230
Mejora de la eficiencia de la búsqueda	234
Búsqueda en un arreglo para una correspondencia de rango	237
Permanencia dentro de los límites del arreglo	241
Uso de un ciclo for para procesar arreglos	244
Resumen del capítulo	245

Términos clave	246
Preguntas de repaso	246
Ejercicios	249
Encuentre los errores	253
Zona de juegos	253
Para discusión	255
CAPÍTULO 7	
Manejo de archivos y aplicaciones	257
Comprensión de los archivos de computadora	258
Organización de los archivos	259
Comprensión de la jerarquía de datos	260
Ejecución de operaciones con archivos	261
Declarar un archivo	261
Abrir un archivo	262
Leer datos de un archivo	262
Escribir datos en un archivo	264
Cerrar un archivo	264
Un programa que ejecuta operaciones de archivo	264
Comprensión de los archivos secuenciales y la lógica de control de interrupciones	267
Comprensión de la lógica de control de interrupciones	268
Unión de archivos secuenciales	273
Procesamiento de archivos maestros y de transacción	281
Archivos de acceso aleatorio	290
Resumen del capítulo	292
Términos clave	293
Preguntas de repaso	295
Ejercicios	299
Encuentre los errores	302
Zona de juegos	302
Para discusión	303
APÉNDICE A	
Comprensión de los sistemas de numeración y los códigos de computadora	305
El sistema hexadecimal	311
Medición del almacenamiento	312
Términos clave	314
APÉNDICE B	
Símbolos de diagrama de flujo	315
APÉNDICE C	
Estructuras	316

CONTENIDO

APÉNDICE D	Resolución de problemas de estructuración difíciles	318
APÉNDICE E	Creación de gráficas impresas	328
x		
APÉNDICE F	Dos variaciones de las estructuras básicas: <code>case</code> y <code>do-while</code>	330
	La estructura <code>case</code>	330
	El ciclo <code>do-while</code>	332
	Reconocimiento de las características compartidas por todos los ciclos estructurados.	334
	Reconocimiento de ciclos no estructurados	335
	Términos clave	336
	Glosario	337
	Índice	349

Prefacio

Introducción a la Programación Lógica y Diseño, 7a. Ed., brinda al programador principiante una guía para desarrollar una lógica de programa estructurada. En este libro de texto se supone que el lector no tiene experiencia con los lenguajes de programación; su redacción no es técnica y enfatiza las buenas prácticas de programación. Los ejemplos se relacionan con los negocios; no requieren una formación matemática adicional a las matemáticas de negocios de bachillerato. Además, los ejemplos ilustran uno o dos puntos importantes; no contienen un número excesivo de características que hagan que los estudiantes se pierdan al seguir los detalles irrelevantes y extraños.

Los ejemplos se han creado para proporcionar a los estudiantes una formación sólida en el área de lógica, sin importar cuáles lenguajes de programación usen al final para escribir los programas. Este libro puede usarse en un curso de lógica independiente que los estudiantes tomen como prerequisito para otro de programación, o como un libro complementario para algún texto de introducción a la programación que use cualquier lenguaje de programación.

Organización y cobertura

Introducción a la Programación Lógica y Diseño, 7a. Ed., presenta a los estudiantes los conceptos de programación e impone un buen estilo y pensamiento lógico. Los conceptos generales de programación se presentan en el capítulo 1; en el capítulo 2 se expone el uso de los datos y muestra dos conceptos importantes: la modularización y la creación de programas de alta calidad. Es importante enfatizar estos temas desde el principio de modo que los estudiantes comiencen a pensar en una forma modular y se concentren en hacer sus programas más eficientes, robustos, fáciles de leer y de mantener.

El capítulo 3 cubre los conceptos clave de estructura, entre los que se encuentran qué es, cómo reconocerla y, de mayor importancia, las ventajas de escribir programas estructurados; el contenido de este capítulo es único entre los textos de programación. El esbozo inicial de la estructura que se presenta aquí da a los estudiantes un fundamento sólido para pensar en forma estructurada.

Los capítulos 4, 5 y 6 exploran las complejidades de la toma de decisiones, la elaboración de ciclos y la manipulación de arreglos. El capítulo 7 proporciona detalles del manejo de archivos de modo que los estudiantes puedan crear programas que procesen una cantidad considerable de datos.

Los primeros tres apéndices proporcionan a los estudiantes resúmenes de los sistemas de numeración, símbolos de diagramas de flujo y estructuras; los apéndices adicionales les

permiten obtener experiencia extra con la estructuración de programas no estructurados grandes, crear gráficas impresas y entender los ciclos posprueba y las estructuras case.

Introducción a la Programación Lógica y Diseño combina la explicación en el texto con ejemplos de diagramas de flujo y seudocódigo con el objetivo de proporcionar a los estudiantes medios alternativos para expresar la lógica estructurada.

Numerosos ejercicios detallados de programas completos al final de cada capítulo ilustran los conceptos que se explican en el capítulo y refuerzan la comprensión y la retención del material presentado.

Introducción a la Programación Lógica y Diseño se distingue de otros libros de lógica de programación en lo siguiente:

- Está escrito y diseñado de manera que no sea específico para ningún lenguaje. La lógica que se usa en este libro puede aplicarse a cualquier lenguaje de programación.
- Los ejemplos se relacionan con los negocios cotidianos; no se espera ningún conocimiento especial de matemáticas, contabilidad u otras disciplinas.
- El concepto de estructura se cubre antes que en muchos otros textos. Los estudiantes se exponen a la estructura en forma natural, de modo que creen en forma automática programas diseñados de manera apropiada.
- La explicación del texto se intercala con diagramas de flujo y seudocódigo de modo que los estudiantes se sientan cómodos con estas herramientas de desarrollo lógico y comprendan su interrelación. También se incluyen capturas de pantalla de programas en ejecución, lo que proporciona a los estudiantes una imagen clara y concreta de la ejecución de los programas.
- Se elaboran programas complejos por medio del uso de ejemplos de negocios completos. Los estudiantes ven cómo se construye una aplicación de principio a fin en lugar de estudiar sólo segmentos de programas.

Características

Este texto se enfoca en ayudar a los estudiantes a convertirse en mejores programadores y comprender el panorama completo en el desarrollo de programas por medio de una variedad de características clave. Además de los Objetivos, Resúmenes y Términos clave del capítulo, estas características serán útiles para los alumnos sin importar su estilo de aprendizaje.

DIAGRAMAS DE FLUJO,
figuras e ilustraciones pro-
porcionan al lector una expe-
riencia de aprendizaje visual.

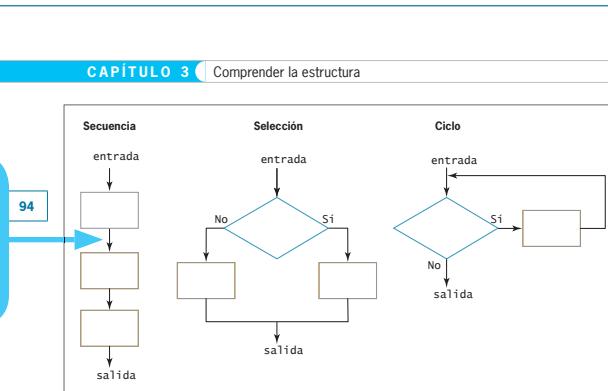


Figura 3-11 Las tres estructuras



Trate de imaginar que levanta físicamente cualquiera de las tres estructuras usando las "manijas" de entrada y salida. Éstos son los puntos en los que podría conectar una estructura con otra. Del mismo modo, cualquier estructura completa, desde su punto de entrada hasta el de salida, puede insertarse dentro del símbolo de proceso de cualquier otra estructura.

En resumen, un programa estructurado tiene las siguientes características:

- Un programa estructurado sólo incluye combinaciones de las tres estructuras básicas: secuencia, selección y ciclo. Cualquier programa estructurado podría contener uno, dos o los tres tipos de estructuras.
- Cada estructura tiene solo un punto de entrada y uno de salida.
- Las estructuras pueden apilarse o conectarse entre sí sólo en sus puntos de entrada o salida.
- Cualquier estructura puede anidarse dentro de otra.



Nunca se requiere que un programa estructurado contenga ejemplos de las tres estructuras. Por ejemplo, muchos programas sencillos sólo contienen una secuencia de varias tareas que se ejecutan de principio a fin sin que se necesite alguna selección o ciclo. Como otro ejemplo, un programa podría desplegar una serie de números usando ciclos, pero nunca tomando alguna decisión sobre los números.

NOTAS proporcionan información adicional; por ejemplo, otra ubicación en el libro que amplía un tema, o un error común con el que se debe tener cuidado.

El uso de diagramas de flujo es excelente. Éste es un libro imprescindible para aprender lógica de programación antes de abordar los diversos lenguajes.

—Lori Selby, University of Arkansas at Monticello

DOS VERDADES Y UNA MENTIRA son cuestionarios breves que aparecen después de cada sección del capítulo, en los que se proporcionan las respuestas. El cuestionario contiene tres afirmaciones basadas en la sección precedente del texto; dos afirmaciones son verdaderas y una es falsa. Las respuestas dan retroalimentación inmediata sin “revelar” las respuestas a las preguntas de opción múltiple y a los problemas de programación que aparecen al final del capítulo.

Uso de una entrada anticipada para estructurar un programa

DOS VERDADES Y UNA MENTIRA

Comprensión de las tres estructuras básicas

95

La afirmación falsa es la número 3. Las tres estructuras pueden aplicarse o aplicarse en un número infinito de formas.

1. Cada estructura en la programación estructurada es una secuencia, una selección o un ciclo.
2. Todos los problemas de lógica pueden resolverse usando sólo tres estructuras: secuencia, selección y ciclo.
3. Las tres estructuras no pueden combinarse en un solo programa.

Uso de una entrada anticipada para estructurar un programa

Recuerde el programa para duplicar números que se mencionó en el capítulo 2; la figura 3-12 muestra un programa similar. El programa da entrada a un número y comprueba la condición de fin de archivo. Si la condición no se cumple, entonces el número se duplica, la respuesta se despliega y se introduce el siguiente número.

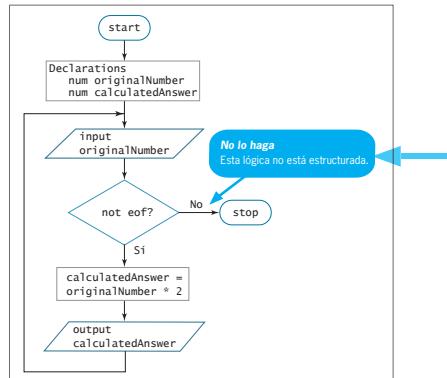


Figura 3-12 Diagrama de flujo no estructurado de un programa para duplicar números

EL ICONO NO LO HAGA ilustra cómo NO hacer algo; por ejemplo, tener una ruta de código muerta en un programa. Este ícono proporciona una sacudida visual al estudiante, enfatizando cuáles figuras particulares NO deben emularse y haciendo que sean más cuidadosos para reconocer los problemas en un código existente.

Evaluación

El material está muy bien escrito, presentado con claridad y actualizado. Todas las explicaciones son muy sólidas, y el lenguaje de Farrell es limpio, contundente y fácil de seguir.

—Judy Woodruff, Indiana University-Purdue University Indianapolis

Los **EJERCICIOS** proporcionan oportunidades para practicar los conceptos; son de dificultad creciente y permiten a los estudiantes explorar los conceptos de programación lógica. Cada ejercicio puede completarse usando diagramas de flujo, seudocódigo o ambos. Además, los profesores pueden asignarlos como problemas de programación que serán codificados y ejecutados en un lenguaje de programación particular.

CAPÍTULO 3 | Comprender la estructura

Preguntas de repaso

1. La lógica de programa enredada se llama código _____.
 - vibora
 - cadena
 - espagueti
 - retorcido
2. Las tres estructuras de la programación estructurada son _____.
 - secuencia, orden y proceso
 - secuencia, selección y ciclo
 - selección, ciclo e iteración
 - if, else y then
3. Una estructura de secuencia puede contener _____.
 - cualquier cantidad de tareas
 - no más de tres tareas
 - exactamente tres tareas
 - solo una tarea
4. ¿Cuál de los siguientes *no* es otro término para una estructura de selección?
 - de decisión
 - if de alternativa dual
 - if-then-else
 - de ciclo
5. La estructura en la que usted hace una pregunta y, dependiendo de la respuesta, efectúa alguna acción y luego hace la pregunta de nuevo, puede llamarse de todas las maneras siguientes excepto _____.
 - iteración
 - repetición
 - ciclo
 - if-then-else
6. La colocación de una estructura dentro de otra se llama _____.
 - apilarlas
 - construirlas
 - desenredarlas
 - anidarlas
7. Unir estructuras por sus extremos se llama _____.
 - apilarlas
 - construirlas
 - desenredarlas
 - anidarlas
8. La declaración `if age >= 65 then seniorDiscount = "s1"` es un ejemplo de _____.
 - secuencia
 - selección de alternativa dual
 - ciclo
 - selección de alternativa única
9. La declaración `while temperature se mantiene menor a 60, dejar el h encendido` es un ejemplo de _____.
 - secuencia
 - selección de alternativa dual
 - ciclo
 - selección de alternativa única

20. Una variable
a) es
b) es
c) es
d) to

Ejercicio

1. ¿Cuál

a.

```

a = 5
b = 2
c = 5
while
  a
  b
endwh
output

```

d.

```

j = 1
k = 9
n = 9
while
  n
  wh

```

Las PREGUNTAS DE REPASO prueban la comprensión del estudiante de las ideas y técnicas principales presentadas. Se incluyen 20 preguntas al final de cada capítulo.

Ejercicios		
20. Una variable podría contener un valor incorrecto aun cuando _____.		
a) es el tipo de datos correcto		
b) está dentro de un rango requerido		
c) es una constante codificada por el programador		
d) todos los anteriores		
Ejercicios		
1. ¿Cuál es la salida para cada uno de los segmentos de seudocódigo de la figura 5-22?		
a.	b.	c.
<pre> a = 1 b = 2 c = 5 while a < c a = a + 1 b = b + c endwhile output a, b, c </pre>	<pre> d = 4 e = 6 f = 7 while d > f d = d + 1 e = e + 1 endwhile output d, e, f </pre>	<pre> g = 4 h = 6 while g < h g = g + 1 endwhile output g, h </pre>
d.	e.	f.
<pre> j = 2 k = 5 n = 9 while j < k m = 6 while m < n output "Adiós" m = m + 1 endwhile j = j + 1 endwhile </pre>	<pre> j = 2 k = 5 m = 6 n = 9 while j < k while m < n output "Hola" m = m + 1 endwhile j = j + 1 endwhile </pre>	<pre> p = 2 q = 4 while r < q output "Adiós" r = r + 1 while s < q output "Hola" s = s + 1 endwhile p = p + 1 endwhile </pre>
Figura 5-22 Segundos de seudocódigo para el ejercicio 1		
2. Diseñe la lógica para un programa que dé como salida todos los números del 1 al 20.		
3. Diseñe la lógica para un programa que dé como salida todos los números del 1 al 20 junto con su valor al doble y al triple.		
4. Diseñe la lógica para un programa que dé como salida todos los números pares del 2 al 100.		
5. Diseñe la lógica para un programa que dé como salida números en orden invertido del 25 al 0.		
6. Diseñe la lógica para un programa que permita a un usuario introducir un número. Despliegue la suma de todos los números desde 1 hasta el número introducido.		

Se incluyen **EJERCICIOS DE DEPURACIÓN** en cada capítulo debido a que examinar los programas en forma crítica y minuciosa es una habilidad crucial de la programación. Los estudiantes pueden descargar estos ejercicios en www.cengagebrain.com (disponibles sólo para la versión original en inglés). Estos archivos también están disponibles para los profesores por medio del CD de Recursos para el profesor y login.cengage.com. (**Nota importante:** Estos ejercicios se encuentran disponibles **sólo** para la versión original en inglés.)

10. Trace un diagrama de flujo estructurado o escriba un pseudocódigo estructurado que expliquen cómo enviar un regalo. Incluya al menos dos decisiones y dos ciclos.
11. Trace un diagrama de flujo estructurado o escriba un pseudocódigo estructurado que expliquen los pasos que debe seguir el dependiente de una tienda de abarrotes para cobrarle a un cliente. Incluya al menos dos decisiones y dos ciclos.

Encuentre los errores

12. Sus archivos descargables para el capítulo 3 incluyen DEBUG03-01.txt, DEBUG03-02.txt y DEBUG03-03.txt. Cada archivo empieza con algunos comentarios que describen el problema. Los comentarios son líneas que comienzan con // o diagonales(/). Después de los comentarios, cada archivo tiene uno o más errores que debe encontrar y corregir. (Los errores se encuentran disponibles sólo en inglés).

Zona de juegos

13. Elija un juego simple para niños y describa su lógica, usando un diagrama de flujo estructurado o un pseudocódigo. Por ejemplo, podría explicar el "juego de las sillas", el juego de naipes "Guerritas", o el juego "el marimón".
14. Elija un programa de concurso de televisión como Jeopardy! y describa sus reglas usando un diagrama de flujo estructurado.
15. Elija un deporte como béisbol o fútbol americano y describa las reglas y el límite de juego (como un turno al bate en el béisbol o el tiempo de juego en el fútbol americano) usando un diagrama de flujo estructurado o un pseudocódigo.

Se incluyen **EJERCICIOS DE ZONA DE JUEGOS** al final de cada capítulo. Los estudiantes pueden crear juegos como una forma adicional entretenida de comprender los conceptos clave de programación.

número no sale en los tres lanzamientos, la computadora gana.

18. Cree la lógica para el juego de dados Pig, en el que un jugador puede competir con la computadora. El objetivo del juego es ser el primero en obtener 100 puntos. El usuario y la computadora toman turnos para "lanzar" un par de dados siguiendo estas reglas:

- En un turno, cada jugador lanza dos dados. Si no aparece 1, los valores de los dados se suman a un total actualizado. Para el turno y el jugador puede elegir si lanza de nuevo o pasa el turno al otro. Cuando un jugador pasa el total acumulado en el turno se suma a su total del juego.
- Si aparece 1 en uno de los dados, el total del turno del jugador se convierte en 0; en otras palabras, no se suma nada más al total del juego del jugador para ese turno, y le toca el turno al otro.
- Si aparece 1 en ambos dados, no sólo se acaba el turno del jugador, sino que el total acumulado entero del jugador se reinicia a 0.
- Cuando la computadora no lanza un 1 y puede elegir si lanza de nuevo, genera un número aleatorio de 1 a 2. Entonces la computadora decidirá continuar cuando el valor es 1 y decidirá salir y pasar el turno al jugador cuando el valor no es 1.

Para discusión

19. Suponga que escribe un programa que usted sospecha que está en un ciclo infinito debido a que se mantiene corriendo por varios minutos sin salida y sin terminar. ¿Qué le agregaría a su programa para que le ayude a descubrir el origen del problema?
20. Suponga que sabe que todos los empleados en su organización tienen un número de identificación de siete dígitos que se usa para entrar al sistema de cómputo. Un ciclo sería útil para adivinar todas las combinaciones de siete dígitos en una identificación. ¿Hay alguna circunstancia en la que usted debería intentar adivinar el número de identificación de otro empleado?
21. Si todos los empleados en una organización tuvieran un número de identificación de siete dígitos, adivinar todas las combinaciones posibles sería una tarea de programación relativamente fácil. ¿Cómo podría alterar el formato de las identificaciones de empleados para hacerlas más difíciles de adivinar?

Las **PREGUNTAS DE ENSAYO** presentan cuestiones personales y éticas que los programadores deben considerar. Pueden usarse para asignaciones por escrito o como un punto de partida para la discusión en el salón de clases.

118

212

Otras características del texto

Esta edición incluye muchas características para ayudar a los estudiantes a convertirse en mejores programadores y comprender el panorama completo en el desarrollo de programas. Todas las explicaciones se han revisado con cuidado para proporcionar la instrucción más clara posible. El material que antes se incluía en notas marginales se ha incorporado al texto principal, dando a las páginas una apariencia más eficiente. Todos los capítulos en esta edición contienen ejercicios de programación nuevos. Todos los ejercicios de la sexta edición que se han reemplazado están disponibles en el CD de Recursos para el profesor y por medio de *login.cengage.com* de modo que los instructores puedan usarlos como ejercicios asignados adicionales o como temas para las discusiones en clase (disponibles sólo para la versión original en inglés).

- **Explicaciones claras.** El lenguaje y las explicaciones en este libro se han refinado a lo largo de siete ediciones, proporcionando las explicaciones más claras posibles de los conceptos difíciles.
- **Énfasis en la estructura.** Más que sus competidores, este libro enfatiza la estructura. El capítulo 3 proporciona una imagen inicial de los conceptos importantes de la programación estructurada, dando a los estudiantes una visión general de los fundamentos antes de que se requieran para considerar los detalles de los programas.
- **Énfasis en la modularidad.** A partir del segundo capítulo se alienta a los estudiantes a escribir código en módulos concisos, que es fácil manejar y reutilizar. Los profesores han encontrado que la modularización debe fomentarse pronto para instilar hábitos adecuados y una comprensión más clara de la estructura. Esta edición usa la modularización al principio, usando variables globales en lugar de valores locales pasados y devueltos, y guarda el paso de los parámetros para después cuando el estudiante haya adquirido más habilidad.
- **Métodos como cajas negras.** El uso de métodos es consistente con los lenguajes con los que quizás el estudiante tenga sus primeras experiencias de programación. En particular, este libro enfatiza el uso de los métodos como cajas negras, declarando todas las variables y constantes como locales para los métodos, pasando argumentos y recibiendo valores devueltos de los métodos según sea necesario.
- **Objetivos.** Cada capítulo comienza con una lista de objetivos de modo que los estudiantes sepan cuáles temas se presentarán. Además de proporcionar una referencia rápida de los temas, esta característica proporciona un auxiliar de estudio útil.
- **Seudocódigo.** Este libro incluye numerosos ejemplos de seudocódigo, que ilustran el uso correcto de los conceptos de lógica y diseño de programación que se explican.
- **Resúmenes de capítulo.** Después de cada capítulo hay un resumen que recapitula los conceptos y técnicas de programación que se estudian. Esta característica proporciona un medio conciso para que los estudiantes repasen y verifiquen su comprensión de los puntos principales en cada capítulo.
- **Términos clave.** Cada capítulo lista los términos clave y sus definiciones; la lista aparece en el orden en que se encuentran los términos en el capítulo. Junto con el resumen del capítulo, la lista de términos clave proporciona una visión general instantánea de las ideas principales. Un glosario al final del libro lista todos los términos clave en orden alfabético, junto con sus definiciones básicas.

Recursos para el instructor

Las siguientes herramientas de enseñanza están disponibles para el instructor en un CD-ROM (sólo para la versión original en inglés). Muchas también están disponibles para descargarlas a través de nuestro sitio acompañante del instructor en login.cengage.com.

- **Manual electrónico del instructor.** El Manual del instructor sigue el texto capítulo por capítulo para asistir en la planeación y organización de un curso efectivo y atractivo. Incluye objetivos de aprendizaje, esbozos del capítulo, apuntes para clase, ideas para actividades en el salón de clases y abundantes recursos adicionales. También se encuentra disponible un plan de estudios del curso de muestra.
- **Guías PAL.** Junto con *Introducción a la Programación Lógica y Diseño*, estos libros breves, o Guías PAL, proporcionan una oportunidad excelente para aprender los fundamentos de la programación mientras se logra la exposición a un lenguaje de programación. Los lectores descubrirán cómo se comporta el código real dentro del contexto del curso de lógica y diseño tradicionalmente independiente del lenguaje. Las guías PAL están disponibles para C++, Java y Visual Basic; por favor póngase en contacto con su representante de ventas para más información sobre cómo agregar las guías PAL a su curso.
- **Presentaciones PowerPoint (disponibles sólo para la versión original en inglés).** Este texto proporciona diapositivas de PowerPoint para acompañar a cada capítulo. Las diapositivas se incluyen para guiar la presentación en el salón de clases, para ponerlas a disposición de los estudiantes para un repaso del capítulo o imprimirlas como folletos para el salón de clases. Los profesores pueden personalizar las diapositivas, que incluyen los archivos de figura completos del texto, para que se adecuen mejor a sus cursos.
- **Soluciones.** Se proporcionan las soluciones a las preguntas de repaso y los ejercicios para asistirle en la calificación.
- **ExamView®** (disponible sólo para la versión original en inglés). Este libro de texto es acompañado por ExamView, un potente paquete de software de exámenes que permite a los profesores crear y administrar exámenes impresos, basados en LAN y por internet. ExamView incluye cientos de preguntas que corresponden al texto, permitiendo a los estudiantes generar guías de estudio detalladas que incluyan referencias de página para un repaso más a fondo. Los componentes de prueba basados en computadora y en internet permiten a los estudiantes presentar exámenes en sus computadoras, y los componentes le ahorran tiempo al profesor al calificar cada examen en forma automática. Estos bancos de pruebas también están disponibles en formatos compatibles para Blackboard, WebCt y Angel.

Opciones de software

Tiene la opción de vincular el software con su texto. Por favor póngase en contacto con su representante de ventas de Cengage Learning para más información.

- **Microsoft® Office Visio® Professional 2010**, versión por 60 días. Visio 2010 es un programa de diagramación que permite a los usuarios crear diagramas de flujo y diagramas con facilidad mientras trabaja a lo largo del texto, permitiéndoles visualizar conceptos y aprender de manera más efectiva.

- **Software Visual Logic™.** Visual Logic es una herramienta simple pero potente para enseñar lógica y diseño de programación sin la sintaxis tradicional del lenguaje de programación de alto nivel. Usa diagramas de flujo para explicar los conceptos de programación esenciales expuestos en este libro, incluyendo variables, entrada, asignación, salida, condiciones, ciclos, procedimientos, gráficas, arreglos y archivos. Visual Logic también interpreta y ejecuta diagramas de flujo, proporcionando a los estudiantes una retroalimentación inmediata y precisa. Visual Logic combina el poder de un lenguaje de alto nivel con la facilidad y simplicidad de los diagramas de flujo.

Reconocimientos

Me gustaría agradecer a todas las personas que ayudaron a hacer de este libro una realidad, en especial Dan Seiter. Después de siete ediciones, Dan todavía encuentra formas de mejorar mis explicaciones de modo que podamos crear un libro de la mayor calidad posible. Gracias también a Alyssa Pratt; Brandi Shailer; Catherine DiMassa y Green Pen QA. Estoy agradecida de poder trabajar con tantas personas excelentes que están dedicadas a producir materiales didácticos de calidad.

Estoy en deuda con los muchos revisores que proporcionaron comentarios útiles e intuitivos durante el desarrollo de este libro, incluyendo Linda Cohen, Forsyth Tech; Andrew Hurd, Hudson Valley Community College; George Reynolds, Strayer University; Lori Selby, University of Arkansas at Monticello; y Judy Woodruff, Indiana University—Purdue University Indianapolis.

Gracias, también, a mi esposo, Geoff, y a nuestras hijas, Andrea y Audrey, por su apoyo. Este libro, como lo fueron todas sus ediciones previas, está dedicado a ellos.

Joyce Farrell

1

CAPÍTULO

Una revisión de las computadoras y la programación

En este capítulo usted aprenderá sobre:

- ◎ Sistemas de cómputo
- ◎ Lógica de un programa simple
- ◎ Los pasos que se siguen en el ciclo de desarrollo del programa
- ◎ Declaraciones de pseudocódigo y símbolos de diagramas de flujo
- ◎ Usar un valor centinela para terminar un programa
- ◎ Programación y ambientes de usuario
- ◎ La evolución de los modelos de programación

Comprensión de los sistemas de cómputo

Un **sistema de cómputo** es una combinación de todos los componentes que se requieren para procesar y almacenar datos usando una computadora. Todos los sistemas de cómputo están compuestos por múltiples piezas de hardware y software.

2

- **Hardware** es el equipo o los dispositivos físicos asociados con una computadora. Por ejemplo, todos los teclados, ratones, altavoces e impresoras son hardware. Los dispositivos se manufanan en forma diferente para las computadoras mainframe grandes, las laptops e incluso para las computadoras más pequeñas que están incorporadas en los productos como automóviles y termostatos, pero los tipos de operaciones que efectúan las computadoras de distintos tamaños son muy parecidos. Cuando se piensa en una computadora con frecuencia son sus componentes físicos los que llegan a la mente, pero para que sea útil necesita más que dispositivos; requiere que se le den instrucciones. Del mismo modo en que un equipo de sonido no hace mucho hasta que se le incorpora la música, el hardware de computadora necesita instrucciones que controlen cómo y cuándo se introducen los datos, cómo se procesan y la forma en que se les da salida o se almacenan.
- **Software** son las instrucciones de la computadora que dicen al hardware qué hacer. El software son **programas** o conjuntos de instrucciones escritos por programadores. Usted puede comprar programas previamente escritos que se han almacenado en un disco o descargarlos de la web. Por ejemplo, en los negocios se utilizan programas de procesamiento de palabras y de contabilidad y los usuarios ocasionales de computadoras disfrutan los que reproducen música y juegos. De manera alternativa, usted puede escribir sus propios programas. Cuando escribe las instrucciones de un software se dice que está **programando**. Este libro se enfoca en el proceso de programación.

El software puede clasificarse en dos extensas categorías:

- **Software de aplicación**, que abarca todos los programas que se aplican para una tarea, como los procesadores de palabras, las hojas de cálculo, los programas de nómina e inventarios, e incluso los juegos.
- **Software de sistema**, que incluye los programas que se usan para manejar una computadora, entre los que se encuentran los sistemas operativos como Windows, Linux o UNIX.

Este libro se enfoca en la lógica que se usa para escribir programas de software de aplicación, aunque muchos de los conceptos se aplican a ambos tipos de software.

Juntos, el hardware y el software llevan a cabo tres operaciones importantes en la mayoría de los programas:

- **Entrada**: los elementos de datos entran en el sistema de cómputo y se colocan en la memoria, donde pueden ser procesados. Los dispositivos de hardware que efectúan operaciones de entrada incluyen teclados y ratones. Los **elementos de datos** constan de todo el texto, los números y otras materias primas que se introducen en una computadora y son procesadas por ella. En los negocios, muchos elementos de datos que se usan son los hechos y las cifras sobre entidades como productos, clientes y personal. Sin embargo, los datos también pueden incluir imágenes, sonidos y movimientos del ratón que el usuario efectúa.
- **Procesamiento**: procesar los elementos de datos implica organizarlos o clasificarlos, comprobar su precisión o realizar cálculos con ellos. El componente de hardware que realiza estas tareas es la **unidad central de procesamiento** o **CPU** (siglas del inglés *central processing unit*).

- **Salida:** después de que los elementos de datos se han procesado, la información resultante por lo general se envía a una impresora, un monitor o algún otro dispositivo de salida de modo que las personas vean, interpreten y usen los resultados. Los profesionales de la programación con frecuencia emplean el término *datos* para los elementos de entrada y el de **información** para los datos que se han procesado y que han salido. En ocasiones usted coloca estos datos de salida en **dispositivos de almacenamiento**, como discos o medios flash (*flash media*). Las personas no pueden leer los datos en forma directa desde tales dispositivos, pero éstos contienen información que puede recuperarse posteriormente. Cuando se envía una salida a un dispositivo de almacenamiento en ocasiones se usa después como entrada para otro programa.

Las instrucciones para el ordenador se escriben en un **lenguaje de programación** como Visual Basic, C#, C++ o Java. Del mismo modo en que algunas personas hablan inglés y otras japonés, los programadores escriben en diferentes lenguajes. Algunos de ellos trabajan de manera exclusiva en uno de ellos mientras que otros conocen varios y usan aquel que sea más adecuado para la tarea que se les presenta.

Las instrucciones que usted escribe usando un lenguaje de programación constituyen un **código de programa**; cuando las escribe está **codificando el programa**.

Cada lenguaje de programación tiene reglas que rigen el uso de las palabras y la puntuación. Estas reglas se llaman **sintaxis** del lenguaje. Los errores en el uso de un lenguaje son **errores de sintaxis**. Si usted pregunta: “¿Cómo otengo forma la guardarlo de?” en español, casi todas las personas pueden imaginar lo que probablemente usted quiso decir, aun cuando no haya usado una sintaxis apropiada en español: ha mezclado el orden de las palabras y ha escrito mal una de ellas. Sin embargo, las computadoras no son tan inteligentes como la mayoría de las personas; en este caso, bien podría haber preguntado a la computadora: “¿Xpu mxv ort dod nmcad bf B?”. A menos que la sintaxis sea perfecta, la computadora no puede interpretar en absoluto la instrucción del lenguaje de programación.

Cuando usted escribe un programa por lo general transmite sus instrucciones usando un teclado. Cuando lo hace, las instrucciones se almacenan en la **memoria de la computadora**, que es un almacenamiento interno temporal. La **memoria de acceso aleatorio** o **RAM** es una forma de memoria interna volátil. Los programas que “corren” (es decir, se ejecutan) en la actualidad y los elementos de datos que se usan se almacenan en la RAM para que sea posible tener un acceso rápido a ellos. El almacenamiento interno es **volátil**, es decir, su contenido se pierde cuando la computadora se apaga o se interrumpe la energía. Por lo general, usted desea recuperar y quizás modificar más tarde las instrucciones almacenadas, de modo que también tiene que guardarlas en un dispositivo de almacenamiento permanente, como un disco. Estos dispositivos se consideran **no volátiles**, esto es, su contenido es permanente y se conserva aun cuando la energía se interrumpa. Si usted ha experimentado una interrupción de la energía mientras trabajaba en una computadora pero pudo recuperar su información cuando aquélla se restableció, no se debió a que su trabajo todavía se encontrara en la RAM. Su sistema se ha configurado para guardar automáticamente su trabajo a intervalos regulares en un dispositivo de almacenamiento no volátil.

Después de que se ha escrito un programa usando declaraciones de un lenguaje de programación y se ha almacenado en la memoria, debe traducirse a un **lenguaje de máquina** que representa los millones de circuitos encendidos/apagados dentro de la computadora. Sus declaraciones en lenguaje de programación se llaman **código fuente** y las traducidas al lenguaje de máquina se denominan **código objeto**.

Cada lenguaje de programación usa una pieza de software llamada **compilador** o **intérprete** para traducir su código fuente al lenguaje de máquina. Este último también se llama **lenguaje binario** y se representa como una serie de 0 (ceros) y 1 (unos). El compilador o intérprete que traduce el código indica si cualquier componente del lenguaje de programación se ha usado de manera incorrecta. Los errores de sintaxis son relativamente fáciles de localizar y corregir debido a que el compilador o intérprete los resalta. Si usted escribe un programa de computadora usando un lenguaje como C++ pero deletrea en forma incorrecta una de sus palabras o invierte el orden apropiado de dos de ellas, el software le hace saber que encontró un error desplegando un mensaje tan pronto como usted intenta traducir el programa.



Aunque hay diferencias en la forma en que trabajan los compiladores y los intérpretes, su función básica es la misma: traducir sus declaraciones de programación en un código que la computadora pueda usar. Cuando usted usa un compilador, un programa entero se traduce antes de que pueda ejecutarse; cuando utiliza un intérprete, cada instrucción es traducida justo antes de la ejecución. Por lo general usted no elige cuál tipo de traducción usar, esto depende del lenguaje de programación. Sin embargo, hay algunos lenguajes que disponen tanto de compiladores como de intérpretes.

Después de que el código fuente es traducido con éxito al lenguaje de máquina, la computadora puede llevar a cabo las instrucciones del programa. Un programa **corre** o se **ejecuta** cuando se realizan las instrucciones. En un programa típico se aceptará alguna entrada, ocurrirá algún procesamiento y se producirá alguna salida.



Además de los lenguajes de programación exhaustivos populares como Java y C++, muchos programadores usan **lenguajes de programación interpretados** (que también se llaman **lenguajes de programación de scripting** o **lenguajes de script**) como Python, Lua, Perl y PHP. Los scripts escritos en estos lenguajes por lo general pueden mecanografiarse en forma directa desde un teclado y almacenarse como texto en lugar de como archivos ejecutables binarios. Los lenguajes de script son interpretados línea por línea cada vez que se ejecuta el programa, en lugar de ser almacenados en forma compilada (binaria). Aun

DOS VERDADES Y UNA MENTIRA

Comprensión de los sistemas de cómputo

En cada sección “Dos verdades y una mentira”, dos de las afirmaciones numeradas son verdaderas y una es falsa. Identifique la que es falsa y explique por qué lo es.

1. El hardware es el equipo o los dispositivos asociados con una computadora. El software son las instrucciones.
2. Las reglas gramaticales de un lenguaje de programación de computadoras constituyen su sintaxis.
3. Usted escribe programas usando el lenguaje de máquina y el software de traducción convierte las declaraciones a un lenguaje de programación.

La afirmación falsa es la número 3. Se escriben programas usando un lenguaje de programación como Visual Basic o Java, y un programa de traducción (llamado compilador o intérprete) convierte las declaraciones en lenguaje de máquina, el cual se compone de 0 (ceros) y 1 (unos).

así, con todos los lenguajes de programación cada instrucción debe traducirse al lenguaje de máquina antes de que pueda ejecutarse.

Comprendión de la lógica de programa simple

Un programa con errores de sintaxis no puede traducirse por completo ni ejecutarse. Uno que no los tenga es traducible y puede ejecutarse, pero todavía podría contener **errores lógicos** y dar como resultado una salida incorrecta. Para que un programa funcione en forma apropiada usted debe desarrollar una **lógica** correcta, es decir, escribir las instrucciones en una secuencia específica, no dejar fuera ninguna instrucción y no agregar instrucciones ajenas.

Suponga que indica a alguien que haga un pastel de la siguiente manera:

Consiga un tazón
Revuelva
Agregue dos huevos
Agregue un litro de gasolina
Hornee a 350° por 45 minutos
Agregue tres tazas de harina

No lo haga
¡No hornee un pastel
como éste!



Las instrucciones peligrosas para hornear un pastel se muestran con un icono “No lo haga”. Verá este icono cuando se presente en el libro una práctica de programación no recomendada que se usa como ejemplo de lo que *no* debe hacerse.

Aun cuando la sintaxis de las instrucciones para hornear un pastel es correcta en español, están fuera de secuencia; faltan algunas y otras pertenecen a procedimientos distintos que no tienen nada que ver con hornear un pastel. Si las sigue no hará un pastel que sea posible ingenier y quizá termine por ser un desastre. Muchos errores lógicos son más difíciles de localizar que los de sintaxis; es más fácil determinar si la palabra “huevos” en una receta está escrita en forma incorrecta que decir si hay demasiados o si se agregaron demasiado pronto.

Del mismo modo en que las instrucciones para hornear pueden proporcionarse en chino mandarín, urdu o inglés, la lógica de programa puede expresarse en forma correcta en cualquier cantidad de lenguajes de programación. Debido a que este libro no se enfoca en algún lenguaje específico, los ejemplos de programación pudieron escribirse en Visual Basic, C++ o Java. Por conveniencia, ¡en este libro las instrucciones se han escrito en español!



Después de aprender francés, usted automáticamente conoce, o puede imaginar con facilidad, muchas palabras en español. Del mismo modo, después de aprender un lenguaje de programación, es mucho más fácil entender otros lenguajes.

Los programas de computadora más sencillos incluyen pasos que ejecutan la entrada, el procesamiento y la salida. Suponga que desea escribir un programa para duplicar cualquier número que proporcione. Puede escribirlo en un lenguaje como Visual Basic o Java, pero si lo escribiera con declaraciones en inglés, se verían así:

```
input myNumber
set myAnswer = myNumber * 2
output myAnswer
```

El proceso de duplicar el número incluye tres instrucciones:

- La instrucción `input myNumber` es un ejemplo de una operación de entrada. Cuando la computadora interpreta esta instrucción sabe que debe buscar un dispositivo de entrada para obtener un número. Cuando usted trabaja en un lenguaje de programación específico, escribe instrucciones que indican a la computadora a cuál dispositivo se tiene acceso para obtener la entrada. Por ejemplo, cuando un usuario introduce un número como los datos para un programa podría hacer clic en el número con un ratón, mecanografiarlo en un teclado o hablar en un micrófono. Sin embargo, es lógico que no importa cuál dispositivo de hardware se use siempre que la computadora sepa que debe aceptar un número. Cuando el número se recupera desde un dispositivo de entrada, se coloca en la memoria de la computadora en una variable llamada `myNumber`. Una **variable** es una ubicación de memoria nombrada cuyo valor puede variar; por ejemplo, el valor de `myNumber` podría ser 3 cuando el programa se usa por primera vez y 45 cuando se usa la siguiente vez. En este libro, los nombres de las variables no llevarán espacios; por ejemplo, se usará `myNumber` en lugar de `my Number`.



Desde una perspectiva lógica, cuando usted introduce, procesa o da salida a un valor, el dispositivo de hardware es irrelevante. Lo mismo sucede en su vida diaria. Si sigue la instrucción “Obtener huevos para el pastel”, en realidad no importa si los compra en una tienda o los recolecta de sus propias gallinas; usted los consigue de cualquier forma. Podría haber diferentes consideraciones prácticas para obtenerlos, del mismo modo que las hay para obtener los datos de una base de datos grande en contraposición a obtenerlos de un usuario inexperto que trabaja en casa en una laptop. Por ahora, este libro sólo se interesa en la lógica de las operaciones, no en detalles menores.

- La instrucción `set myAnswer = myNumber * 2` es un ejemplo de una operación de procesamiento. En la mayoría de los lenguajes de programación se usa un asterisco para indicar una multiplicación, de modo que esta instrucción significa “Cambiar el valor de la ubicación de memoria `myAnswer` para igualar el valor en la ubicación de memoria `myNumber` por dos”. Las operaciones matemáticas no son el único tipo de operaciones de procesamiento, pero son típicas. Como sucede con las operaciones de entrada, el tipo de hardware que se usa para el procesamiento es irrelevante; después de que usted escribe un programa, éste puede usarse en computadoras de diferentes marcas, tamaños y velocidades.
- En el programa para duplicar un número, la instrucción `output myAnswer` es un ejemplo de una operación de salida. Dentro de un programa particular, esta declaración podría causar que la salida aparezca en el monitor (digamos, una pantalla plana de plasma o una de tubo de rayos catódicos), que vaya a una impresora (láser o de inyección de tinta) o que se escriba en un disco o un DVD. La lógica del proceso de salida es la misma sin importar qué dispositivo de hardware se use. Cuando se ejecuta esta instrucción, el valor almacenado en la memoria en la ubicación llamada `myAnswer` se envía a un dispositivo de salida. (El valor de salida también permanece en la memoria hasta que se almacena algo más en la misma ubicación o se interrumpe la energía eléctrica.)



La memoria de la computadora consiste en millones de ubicaciones numeradas donde es posible almacenar datos. La ubicación de memoria de `myNumber` tiene una ubicación numérica específica, pero cuando usted escribe programas rara vez necesita preocuparse por el valor de la dirección de memoria; en cambio, usa el nombre fácil de recordar que creó. Los programadores de computadoras con frecuencia se refieren a las direcciones de memoria usando la notación hexadecimal, o en base 16. Con este sistema podrían utilizar un valor como `42FF01A` para referirse a una dirección de memoria. A pesar del uso de letras, dicha dirección todavía es un número hexadecimal. El apéndice A contiene información sobre este sistema de numeración.

DOS VERDADES Y UNA MENTIRA

Comprensión de la lógica de programa simple

1. Un programa con errores de sintaxis puede ejecutarse pero podría generar resultados incorrectos.
2. Aunque la sintaxis de los lenguajes de programación difiere, la misma lógica de programa puede expresarse en diferentes lenguajes.
3. Los programas de computadora más sencillos incluyen pasos que efectúan entrada, procesamiento y salida.

La afirmación falsa es la número 1. Un programa con errores de sintaxis no puede ejecutarse; uno que no tenga este tipo de errores puede ejecutarse, pero producirá resultados incorrectos.

Comprensión del ciclo de desarrollo del programa

El trabajo de un programador implica escribir instrucciones (como las del programa para duplicar números en la sección anterior), pero por lo general un profesional no sólo se sienta ante un teclado de computadora y comienza a mecanografiar. La figura 1-1 ilustra el **ciclo de desarrollo del programa**, que se divide al menos en siete pasos:

1. Entender el problema.
2. Planear la lógica.
3. Codificar el programa.
4. Usar software (un compilador o intérprete) para traducir el programa a lenguaje de máquina.
5. Probar el programa.
6. Poner el programa en producción.
7. Mantener el programa.

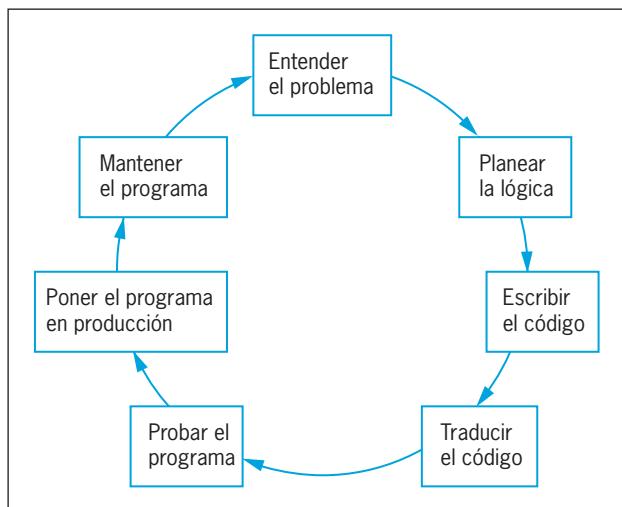


Figura 1-1 El ciclo de desarrollo del programa

Entender el problema

Los programadores profesionales escriben programas para satisfacer las necesidades de otras personas, llamadas **usuarios** o **usuarios finales**. Entre los ejemplos de usuarios finales estaría un departamento de recursos humanos que necesita una lista impresa de todos los empleados, un área de facturación que desea un listado de los clientes que se han retrasado en sus pagos 30 días o más, o un departamento de pedidos que requiere un sitio web para proporcionar a los compradores un carrito de compras en línea. Debido a que los programadores brindan un servicio a estos usuarios deben comprender primero lo que éstos desean. Cuando usted corre un programa con frecuencia piensa en la lógica como un ciclo de operaciones de entrada-procesamiento-salida; pero cuando planea un programa piensa primero en la salida. Después de entender cuál es el resultado deseado puede planear los pasos de entrada y procesamiento para lograrlo.

Suponga que el director de recursos humanos dice a un programador: "Nuestro departamento necesita una lista de todos los empleados que han estado aquí por más de cinco años, porque queremos invitarlos a una cena especial de agradecimiento". En apariencia esta es una solicitud sencilla. Sin embargo, un programador experimentado sabrá que la solicitud está incompleta. Por ejemplo, quizás no sepa las respuestas a las siguientes preguntas sobre cuáles empleados incluir:

- ¿El director desea una lista sólo de empleados de tiempo completo o de tiempo completo y de medio tiempo juntos?
- ¿Desea incluir personas que han trabajado para la compañía con una base contractual mensual durante los pasados cinco años o sólo los empleados permanentes regulares?
- ¿Los empleados necesitan haber trabajado para la organización por cinco años hasta el día de hoy, hasta la fecha de la cena o en alguna otra fecha límite?
- ¿Qué pasa con un empleado que trabajó tres años, tomó una licencia de dos años y volvió a trabajar por tres años?

El programador no puede tomar ninguna de estas decisiones; el usuario (en este caso, el director de recursos humanos) debe abordar estas preguntas.

Tal vez aún se requiera tomar otras decisiones, por ejemplo:

- ¿Qué datos deben incluirse para cada empleado en la lista? ¿Es preciso anotar el nombre y los apellidos? ¿Los números de seguro social? ¿Números telefónicos? ¿Direcciones?
- ¿La lista debe estar en orden alfabético? ¿Por número de identificación del empleado? ¿En orden de años de servicio? ¿Algún otro orden?
- ¿Los empleados deberían agruparse con algún criterio, como número de departamento o años de servicio?

A menudo se proporcionan algunas piezas de documentación para ayudar al programador a entender el problema. La **documentación** consiste en todo el papeleo de soporte para un programa; podría incluir elementos como las solicitudes originales para el programa de los usuarios, muestras de salida y descripciones de los elementos de datos disponibles para la entrada.

Entender por completo el problema es uno de los aspectos más difíciles de la programación. En cualquier trabajo, la descripción de lo que el usuario necesita puede ser imprecisa; peor aún, los usuarios quizás no sepan qué desean en realidad, y los que piensan que saben a menudo cambian de opinión después de ver una muestra de salida. ¡Un buen programador es en parte consejero y en parte detective!

Planear la lógica

El corazón del proceso de programación se encuentra en la planeación de la lógica del programa. Durante esta fase, el programador planifica los pasos del mismo, decidiendo cuáles incluir y cómo ordenarlos. Usted puede visualizar la solución de un problema de muchas maneras. Las dos herramientas de programación más comunes son los diagramas de flujo y el seudocódigo; ambas implican escribir los pasos del programa en inglés, del mismo modo en que planearía un viaje en papel antes de subirse al automóvil o el tema de una fiesta antes de comprar alimentos y recuerdos.

Quizás usted haya escuchado a los programadores referirse a la planeación de un programa como “desarrollar un algoritmo”. Un **algoritmo** es la secuencia de pasos necesarios para resolver cualquier problema.



Además de los diagramas de flujo y el seudocódigo, los programadores usan una variedad de herramientas distintas para desarrollar el programa. Una de ellas es la **gráfica IPO**, que define las tareas de entrada, procesamiento y salida. Algunos programadores orientados hacia los objetos también usan **gráficas TOE**, que listan tareas, objetos y eventos.

El programador no debe preocuparse por la sintaxis de algún lenguaje en particular durante la etapa de planeación, sino enfocarse en averiguar qué secuencia de eventos llevará desde la entrada disponible hasta la salida deseada. La planeación de la lógica incluye pensar con

cuidado en todos los valores de datos posibles que un programa podría encontrar y cómo desea que éste maneje cada escenario. El proceso de recorrer en papel la lógica de un programa antes de escribirlo en realidad se llama **prueba de escritorio (desk-checking)**. Aprenderá más sobre la planeación de la lógica a lo largo de este libro; de hecho, éste se enfoca casi de manera exclusiva en este paso crucial.

10

Codificación del programa

Sólo después de que se ha desarrollado la lógica el programador puede escribir el código fuente. Hay cientos de lenguajes de programación disponibles. Los programadores eligen lenguajes particulares debido a que algunos incorporan capacidades que los hacen más eficientes que otros para manejar ciertos tipos de operaciones. A pesar de sus diferencias, los lenguajes de programación son bastante parecidos en sus capacidades básicas; cada uno puede manejar operaciones de entrada, procesamiento aritmético, operaciones de salida y otras funciones estándares. La lógica que se desarrolla para resolver un problema de programación puede ejecutarse usando cualquier cantidad de lenguajes. Sólo después de elegir alguno el programador debe preocuparse por la puntuación y la ortografía correctas de los comandos; en otras palabras, por usar la *sintaxis* correcta.

Algunos programadores experimentados combinan con éxito en un paso la planeación de la lógica y la codificación del programa. Esto funciona para planear y escribir un programa muy sencillo, del mismo modo en que usted puede planear y escribir una postal para un amigo en un solo paso. Sin embargo, la redacción de un buen ensayo semestral o un guión cinematográfico requiere planeación y lo mismo sucede con la mayor parte de los programas.

¿Cuál paso es más difícil: planear la lógica o codificar el programa? Ahora mismo quizás le parezca que escribir en un lenguaje de programación es una tarea muy difícil, considerando todas las reglas de ortografía y sintaxis que debe aprender. Sin embargo, en realidad el paso de planeación es más difícil. ¿Qué es más complicado: pensar en los giros de la trama de una novela de misterio que es un éxito de ventas o escribir la traducción del inglés al español de una novela que ya se ha escrito? ¿Y quién cree que recibe más paga, el escritor que crea la trama o el traductor? (¡Haga la prueba pidiendo a algunos amigos que nombren a algún traductor famoso!)

Uso de software para traducir el programa al lenguaje de máquina

Aun cuando hay muchos lenguajes de programación, cada computadora conoce sólo uno: su lenguaje de máquina, que consiste en 1 (unos) y 0 (ceros). Las computadoras entienden el lenguaje de máquina porque están formadas por miles de diminutos interruptores eléctricos, cada uno de los cuales presenta un estado de encendido o apagado, que se representa con 1 o 0, respectivamente.

Lenguajes como Java o Visual Basic están disponibles para los programadores debido a que alguien ha escrito un programa traductor (un compilador o intérprete) que cambia el **lenguaje de programación de alto nivel** en inglés del programador en un **lenguaje de máquina de bajo nivel** que la computadora entiende. Cuando usted aprende la sintaxis de un lenguaje de programación, los comandos funcionan en cualquier máquina en la que el software del lenguaje se haya instalado. Sin embargo, sus comandos son traducidos entonces al lenguaje de máquina, que es distinto en las distintas marcas y modelos de computadoras.

Si usted escribe en forma incorrecta una declaración de programación (por ejemplo, escribe mal una palabra, usa alguna que no existe o utiliza gramática “ilegal”), el programa traductor no sabe cómo proceder y emite un mensaje al detectar un error de sintaxis. Aunque nunca es deseable cometerlos, los errores de sintaxis no son una preocupación importante para los programadores porque el compilador o intérprete los detecta y muestra un mensaje que les notifica el problema. La computadora no ejecutará un programa que contenga aunque sea sólo un error de sintaxis.

Por lo común, un programador desarrolla la lógica, escribe el código y compila el programa, recibiendo una lista de errores de sintaxis. Entonces los corrige y compila el programa de nuevo. La corrección del primer conjunto de errores con frecuencia revela otros nuevos que al principio no eran evidentes para el compilador. Por ejemplo, si usted usa un compilador en español y envía la declaración *El prro persiguen al gato*, el compilador al principio señalaría sólo un error de sintaxis. La segunda palabra, *prro*, es ilegal porque no forma parte del español. Sólo después de corregirla a *perro* el compilador hallaría otro error de sintaxis en la tercera palabra, *persiguen*, porque es una forma verbal incorrecta para el sujeto *perro*. Esto no significa que *persiguen* necesariamente sea la palabra equivocada. Quizá *perro* es incorrecto; tal vez el sujeto debería ser *perros*, en cuyo caso *persiguen* sería correcto. Los compiladores no siempre saben con exactitud qué quiere usted ni cuál debería ser la corrección apropiada, pero sí saben cuando algo anda mal con su sintaxis.

Cuando un programador escribe un programa tal vez necesite recompilar el código varias veces. Un programa ejecutable sólo se crea cuando el código no tiene errores de sintaxis. Después de que un programa se ha traducido al lenguaje de máquina, se guarda y puede ser ejecutado cualquier número de veces sin repetir el paso de traducción. Usted sólo necesita retraducir su código si hace cambios en las declaraciones de su código fuente. La figura 1-2 muestra un diagrama de este proceso en su totalidad.

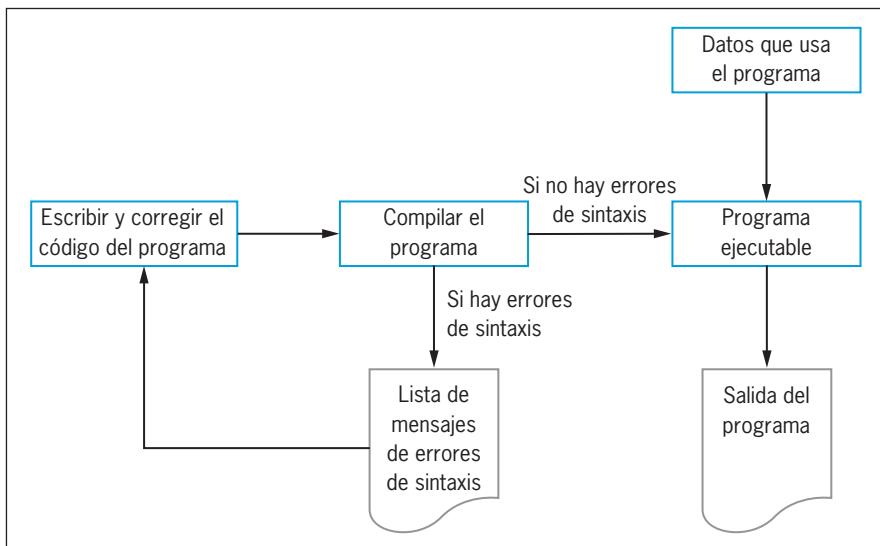


Figura 1-2 Creación de un programa ejecutable

Prueba del programa

Un programa que no tiene errores de sintaxis no necesariamente está libre de errores lógicos. Un error lógico resulta cuando se utiliza una declaración correcta desde el punto de vista sintáctico pero equivocada para el contexto actual. Por ejemplo, la declaración en español *El perro persigue al gato*, aunque sintácticamente es perfecta, no es correcta desde una perspectiva lógica si el perro persigue una pelota o si el gato es el agresor.

Una vez que un programa queda limpio de errores de sintaxis el programador puede probarlo, es decir, ejecutarlo con algunos datos de muestra para ver si los resultados son lógicamente correctos. Recuerde el programa para duplicar un número:

```
input myNumber
set myAnswer = myNumber * 2
output myAnswer
```

Si usted ejecuta el programa, proporciona el valor 2 como entrada para el mismo y se despliega la respuesta 4, ha ejecutado una corrida de prueba exitosa del programa.

Sin embargo, si se despliega la respuesta 40, quizás el programa contenga un error lógico. Tal vez usted tecleó mal la segunda línea del código con un cero extra, de modo que el programa se lee:

```
input myNumber
set myAnswer = myNumber * 20
output myAnswer
```

No lo haga
 El programador tecleó
 20 en lugar de 2.

Escribir 20 en lugar de 2 en la declaración de multiplicación causó un error lógico. Observe que desde el punto de vista sintáctico no hay nada incorrecto en este segundo programa (es

igual de razonable multiplicar un número por 20 que por 2) pero si el programador sólo pretende duplicar `myNumber`, entonces ha ocurrido un error lógico.

El proceso de hallar y corregir los errores del programa se llama **depuración**. Usted depura un programa al probarlo usando muchos conjuntos de datos. Por ejemplo, si escribe el programa para duplicar un número, luego introduce 2 y obtiene un valor de salida de 4, esto no necesariamente significa que el programa es correcto. Quizá tecleó por error este programa:

```
input myNumber
set myAnswer = myNumber + 2
output myAnswer
```

No lo haga
El programador tecleó
“+” en lugar de “*”.

Una entrada de 2 da como resultado una respuesta de 4, pero esto no significa que su programa duplique los números; en realidad sólo les suma 2. Si prueba su programa con datos adicionales y obtiene la respuesta errónea; por ejemplo, si introduce 7 y obtiene una respuesta de 9, sabe que hay un problema con su código.

La selección de los datos de prueba es casi un arte en sí misma y debe hacerse con cuidado. Si el departamento de recursos humanos desea una lista de los nombres de los empleados con antigüedad de cinco años, sería un error probar el programa con un pequeño archivo de muestra que sólo contiene empleados de tiempo indeterminado. Si los empleados más recientes no son parte de los datos que se usan para probar, en realidad no sabe si el programa los habría eliminado de la lista de cinco años. Muchas compañías no saben que su software tiene un problema hasta que ocurre una circunstancia extraña; por ejemplo, la primera vez que un empleado registra más de nueve dependientes, la primera vez que un cliente ordena más de 999 artículos al mismo tiempo o cuando a la internet se le agotan las direcciones IP asignadas, un problema que se conoce como *agotamiento IPV4*.

Poner el programa en producción

Una vez que se ha probado y depurado el programa en forma minuciosa, está listo para que la organización lo use. “Ponerlo en producción” significaría simplemente ejecutarlo una vez, si fue escrito para satisfacer una solicitud del usuario para una lista especial. Sin embargo, el proceso podría llevar meses si el programa se ejecutará en forma regular o si es uno de un gran sistema de programas que se están desarrollando. Quizá las personas que introducirán los datos deben recibir capacitación con el fin de preparar las entradas para el nuevo programa, los usuarios deben recibir instrucción para entender la salida o sea preciso cambiar los datos existentes en la compañía a un formato por completo nuevo para que tengan cabida en dicho programa. Completar la **conversión**, el conjunto entero de acciones que debe efectuar una organización para cambiar al uso de un programa o un conjunto de programas nuevos, en ocasiones puede llevar meses o años.

Mantenimiento del programa

Después de que los programas se colocan en producción, la realización de los cambios necesarios se denomina **mantenimiento**. Puede requerirse por diversas razones: por ejemplo, debido a que se han legislado nuevas tasas de impuestos, se alteró el formato de un archivo de entrada

o el usuario final requiere información adicional no incluida en las especificaciones de salida originales. Con frecuencia, la primera labor de programación que usted lleve a cabo requerirá dar mantenimiento a los programas escritos de manera previa. Cuando dé mantenimiento a los programas que otras personas han escrito apreciará el esfuerzo que hicieron para obtener un código claro, usar nombres de variables razonables y documentar su trabajo. Cuando hace cambios a los programas existentes repite el ciclo de desarrollo. Es decir, debe entender los cambios, luego planearlos, codificarlos, traducirlos y probarlos antes de ponerlos en producción. Si el programa original requiere una cantidad considerable de modificaciones podría ser retirado y empezaría el ciclo de desarrollo del programa para uno nuevo.

DOS VERDADES Y UNA MENTIRA

Comprendiendo el ciclo de desarrollo del programa

1. Entender el problema que debe resolverse puede ser uno de los aspectos más difíciles de la programación.
 2. Las dos herramientas más comunes que se usan en la planeación de la lógica son los diagramas de flujo y el seudocódigo.
 3. La elaboración del diagrama de flujo de un programa es un proceso muy diferente si se usa un lenguaje de programación antiguo en lugar de uno más reciente.

La farmacia es la numero 3. A pesar de sus diferencias, los lenguajes de programación son bastante parecidos en sus capacidades básicas; cada uno puede manejar operaciones de entrada, procesamiento aritmético, operaciones de salida y otras funciones estándar. La logica que se ha desarrollado para resolver un problema de programación puede ejecutarse usando cualquier cantidad de lenguajes.

Uso de declaraciones en seudocódigo y símbolos de diagrama de flujo

Cuando los programadores planean la lógica para dar solución a un problema de programación con frecuencia usan dos herramientas: pseudocódigo o diagramas de flujo.

- El **seudocódigo** es una representación parecida al inglés de los pasos lógicos que se requieren para resolver un problema. *Seudo* es un prefijo que significa *falso*, y *codificar* un programa significa ponerlo en un lenguaje de programación; por consiguiente, *seudocódigo* simplemente significa *código falso*, o declaraciones que en apariencia se han escrito en un lenguaje de programación pero no necesariamente siguen todas las reglas de sintaxis de alguno en específico.
 - Un **diagrama de flujo** es una representación gráfica de lo mismo.

Escritura en seudocódigo

Usted ha visto antes en este capítulo ejemplos de declaraciones que representan un seudocódigo y no hay nada misterioso en ellas. Las siguientes cinco declaraciones constituyen una representación en seudocódigo de un problema para duplicar un número:

```
start
    input myNumber
    set myAnswer = myNumber * 2
    output myAnswer
stop
```

Usar un seudocódigo implica escribir todos los pasos que se usarán en un programa. Por lo general, los programadores introducen su seudocódigo con una declaración inicial como `start` y lo terminan con uno de terminación como `stop`. Las declaraciones entre `start` y `stop` están en inglés y tienen una ligera sangría de modo que destaque `start` y `stop`. La mayoría de los programadores no se preocupan por la puntuación como los puntos al final de las declaraciones del seudocódigo, aunque si usted prefiere ese estilo no sería un error usarlos. Del mismo modo, no hay necesidad de escribir con mayúscula la primera palabra en una declaración, aunque podría elegir hacerlo. Este libro sigue las convenciones de usar letras minúsculas para los verbos que comienzan las declaraciones en el seudocódigo y omitir los puntos al final de las mismas.

El seudocódigo es bastante flexible porque es una herramienta de planeación y no el producto final. Por consiguiente, por ejemplo, quizás prefiera cualquiera de los siguientes:

- En lugar de `start` y `stop`, algunos desarrolladores de seudocódigo usan otros términos como `begin` y `end`.
- En lugar de `input myNumber` algunos escriben `get myNumber` o `read myNumber`.
- En vez de `set myAnswer = myNumber * 2`, algunos escribirán `calculate myAnswer = myNumber times 2` o `compute myAnswer as myNumber doubled`.
- En lugar de `output myAnswer`, muchos desarrolladores escribirán `display myAnswer`, `print myAnswer` o `write myAnswer`.

El punto es que las declaraciones en seudocódigo son instrucciones para recuperar un número original de un dispositivo de entrada y almacenarlo en la memoria, donde puede usarse en un cálculo, para después obtener la respuesta calculada de la memoria y enviarla a un dispositivo de salida de modo que una persona pueda verlo. Cuando al final usted convierte su seudocódigo en un lenguaje de programación específico, no tiene esta flexibilidad porque se requerirá una sintaxis determinada. Por ejemplo, si usa el lenguaje de programación C# y escribe la declaración para dar salida a la respuesta en un monitor, codificará lo siguiente:

```
Console.WriteLine(myAnswer);
```

El uso exacto de las palabras, las mayúsculas y la puntuación es importante en la declaración en C#, pero no en el de seudocódigo.

Trazo de diagramas de flujo

Algunos programadores profesionales prefieren escribir el seudocódigo para trazar los diagramas de flujo debido a que este procedimiento es más parecido a escribir las declaraciones finales en el lenguaje de programación. Otros prefieren trazar diagramas de flujo para representar el flujo lógico debido a que éstos les permiten visualizar con más facilidad cómo se conectarán las declaraciones del programa. Los diagramas de flujo son una herramienta excelente, en especial para los programadores principiantes, pues son útiles a visualizar cómo se interrelacionan las declaraciones en un programa.

Usted puede trazar un diagrama de flujo a mano o usar software como Microsoft Word y Microsoft PowerPoint, que cuentan con herramientas para elaborarlos. Hay otros programas, como Visio y Visual Logic, específicamente para crear diagramas de flujo.

Cuando usted crea uno dibuja formas geométricas que contienen las declaraciones individuales y que se conectan por medio de flechas. (En el apéndice B hay un resumen de todos los símbolos de los diagramas de flujo que verá en este libro.) Para representar un **símbolo de entrada** se usa un paralelogramo que indica una operación de entrada. Se escribe una declaración de entrada en inglés dentro del paralelogramo, como se muestra en la figura 1-3.

Las declaraciones de operaciones aritméticas son ejemplos de procesamiento. En un diagrama de flujo, se usa un rectángulo como el **símbolo de procesamiento** que contiene una declaración de procesamiento, como se muestra en la figura 1-4.

Para representar una declaración de salida se usa el mismo símbolo que para las de entrada: el **símbolo de salida** es un paralelogramo, como se muestra en la figura 1-5. Debido a que el paralelogramo se usa tanto para la entrada como para la salida, con frecuencia se llama **símbolo de entrada/salida** o **símbolo I/O**.



Algunos programas que usan diagramas de flujo (como Visual Logic) utilizan un paralelogramo inclinado hacia la izquierda para representar la salida. Siempre que el creador y el lector del diagrama de flujo estén en comunicación, la forma que se use es irrelevante. En este libro se seguirá la convención estándar de usar el paralelogramo inclinado hacia la derecha tanto para la entrada como para la salida.

A fin de mostrar la secuencia correcta de estas declaraciones se usan flechas o **líneas de flujo** para conectar los pasos. Siempre que sea posible, la mayor parte de un diagrama de flujo debe leerse de arriba hacia abajo o de izquierda a derecha en una página. Esta es la forma en que se lee el inglés, así que cuando los diagramas de flujo siguen esta convención son más fáciles de entender.

Para que un diagrama de flujo esté completo debe incluir dos elementos más: **símbolos terminales** o de inicio/fin en cada extremo. Con frecuencia usted coloca una palabra como **start** o **begin** en el primer símbolo terminal y una palabra como **end** o **stop** en el otro. Los símbolos terminales estándar tienen forma de pista de carreras; muchos programadores la llaman “pastilla” porque se parece a la forma del medicamento que se usa para aliviar una garganta irritada. La figura 1-6 muestra un diagrama de flujo completo para el programa que duplica un

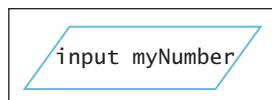


Figura 1-3 Símbolo de entrada

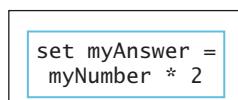


Figura 1-4 Símbolo de procesamiento

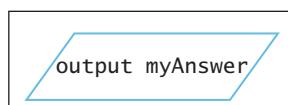


Figura 1-5 Símbolo de salida

número y el seudocódigo para el mismo problema. Es posible observar que las declaraciones en el diagrama de flujo y en el seudocódigo son las mismas, sólo el formato de presentación es diferente.

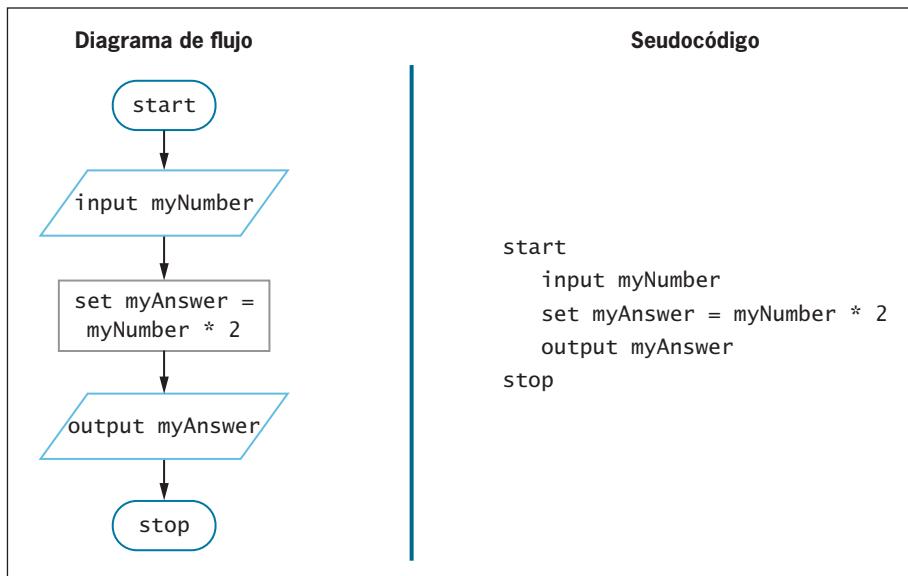


Figura 1-6 Diagrama de flujo y seudocódigo del programa que duplica un número

Los programadores rara vez crean un seudocódigo y un diagrama de flujo para el mismo problema. Por lo general se usa uno u otro. En un programa grande quizás usted prefiera escribir un seudocódigo para algunas partes y trazar un diagrama de flujo para otras.

Cuando usted indica a un amigo cómo llegar a su casa, podría escribir una serie de instrucciones o hacer un mapa. El seudocódigo se parece a las instrucciones escritas paso a paso; un diagrama de flujo, como un mapa, es una representación visual de lo mismo.

Repetición de las instrucciones

Después de desarrollar el diagrama de flujo o el seudocódigo, el programador sólo necesita: 1) adquirir una computadora, 2) comprar un compilador de lenguaje, 3) aprender un lenguaje de programación, 4) codificar el programa, 5) intentar compilarlo, 6) arreglar los errores de sintaxis, 7) compilarlo de nuevo, 8) probarlo con varios conjuntos de datos y 9) ponerlo en producción.

Quizás en este momento usted piense: “¡Vaya! ¡Esto simplemente no vale la pena! ¿Todo ese trabajo para crear un diagrama de flujo o un seudocódigo y *luego* todos esos otros pasos? ¡Por 5 dólares puedo comprar una calculadora de bolsillo que duplicará cualquier número para mí en forma instantánea!”. Tiene razón. Si éste fuera un programa de computadora real y todo lo que hiciera fuera duplicar el valor de un número no valdría el esfuerzo. Escribir un programa tendría caso sólo si requiriera duplicar muchos números (digamos 10,000) en una cantidad de tiempo limitada (quizás los próximos dos minutos).

Por desgracia, el programa que se representa en la figura 1-6 no duplica 10,000 números; sólo duplica uno. Podría ejecutarlo 10,000 veces, por supuesto, pero esto requeriría que se sentara frente a la computadora y lo corriera una y otra vez. Se las arreglaría mejor con uno que pudiera procesar 10,000 números, uno después de otro.

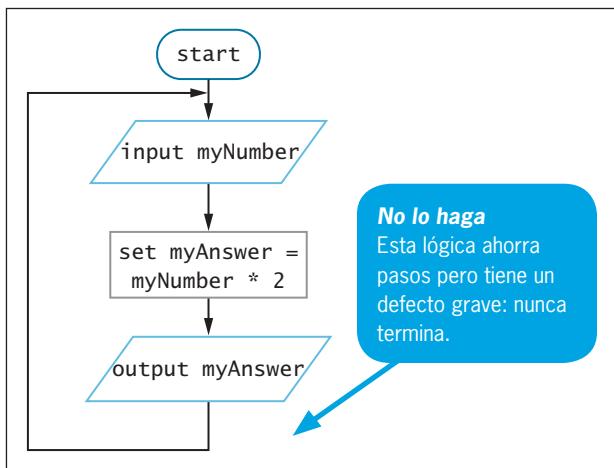
Una solución es escribir el programa que se muestra en la figura 1-7 y ejecutar los mismos pasos 10,000 veces. Por supuesto, escribirlo requeriría mucho tiempo (también podría comprar la calculadora).

```
start
    input myNumber
    set myAnswer = myNumber * 2
    output myAnswer
    input myNumber
    set myAnswer = myNumber * 2
    output myAnswer
    input myNumber
    set myAnswer = myNumber * 2
    output myAnswer
...
...y así otras 9,997 veces más
```

No lo haga
Nunca desearía escribir una lista de instrucciones tan repetitiva.

Figura 1-7 Seudocódigo ineficiente para un programa que duplique 10,000 números

Una mejor solución es hacer que la computadora ejecute el mismo conjunto de tres instrucciones una y otra vez, como se muestra en la figura 1-8. La repetición de una serie de pasos se llama **ciclo**. Con este enfoque, la computadora obtiene un número, lo duplica, despliega la respuesta y luego comienza de nuevo con la primera instrucción. El mismo punto en la memoria, llamado *myNumber*, se reutiliza para el segundo número y para cualesquier números subsiguientes. El punto en la memoria llamado *myAnswer* se reutiliza cada vez para almacenar el resultado de la operación de multiplicación. Sin embargo, la lógica que se ilustra en el diagrama de flujo de la figura 1-8 presenta un problema importante: la secuencia de instrucciones nunca termina. Esta situación se conoce en programación como **ciclo infinito**; un flujo repetitivo de lógica sin fin. Usted aprenderá cómo manejar este problema más adelante en este capítulo; estudiará un método más complejo en el capítulo 3.

**Figura 1-8** Diagrama de flujo de un programa de duplicación de números infinito

DOS VERDADES Y UNA MENTIRA

Uso de declaraciones de seudocódigo y símbolos de diagrama de flujo

1. Cuando se traza un diagrama de flujo, se usa un paralelogramo para representar una operación de entrada.
2. Cuando se traza un diagrama de flujo, se usa un paralelogramo para representar una operación de procesamiento.
3. Cuando se traza un diagrama de flujo, se usa un paralelogramo para representar una operación de salida.

un rectángulo para representar una operación de procesamiento.

La afirmación falsa es la número 2. Cuando se traza un diagrama de flujo, se usa

Uso de un valor centinela para terminar un programa

La lógica en el diagrama de flujo para duplicar números, que se muestra en la figura 1-8, tiene una falla importante: el programa contiene un ciclo infinito. Si, por ejemplo, los números de entrada se introducen por medio del teclado, el programa seguirá aceptando números y dando salida a sus valores duplicados para siempre. Por supuesto, el usuario podría negarse a teclear más números; pero el programa no puede avanzar más mientras esté esperando una entrada; mientras tanto, ocupa memoria de la computadora e inmoviliza recursos del sistema operativo. Dejar de introducir más números no es una solución práctica. Otra forma de terminar el

programa es simplemente apagar la computadora pero, una vez más, ésta no es la mejor solución ni una forma apropiada de hacerlo.

Una mejor forma es establecer un valor predeterminado para `myNumber` que signifique “¡Detén el programa!”. Por ejemplo, el programador y el usuario podrían acordar que este último nunca necesitará conocer el doble de 0 (cero), de modo que podría introducir un 0 para detenerlo. El programa entonces probaría cualquier valor contenido en `myNumber` que entre y si es 0 se detendría. Probar un valor también se llama **tomar una decisión**.

En un diagrama de flujo se representa una decisión al trazar un **símbolo de decisión**, que tiene forma de diamante. El diamante por lo general contiene una pregunta cuya respuesta es una de dos opciones mutuamente excluyentes, con frecuencia sí o no. Todas las buenas preguntas de computación tienen sólo dos respuestas mutuamente excluyentes, como sí y no o verdadero y falso. Por ejemplo, “¿Qué día es su cumpleaños?” no es una pregunta de computación adecuada porque hay 366 respuestas posibles. Sin embargo, “¿Su cumpleaños es el 24 de junio?” sí lo es porque la respuesta siempre es sí o no.

La pregunta para detener el programa de duplicación debería ser “¿El valor de `myNumber` que se acaba de introducir es igual a 0?” o “¿`myNumber = 0?`” para abreviar. El diagrama de flujo completo se verá entonces como el que se muestra en la figura 1-9.

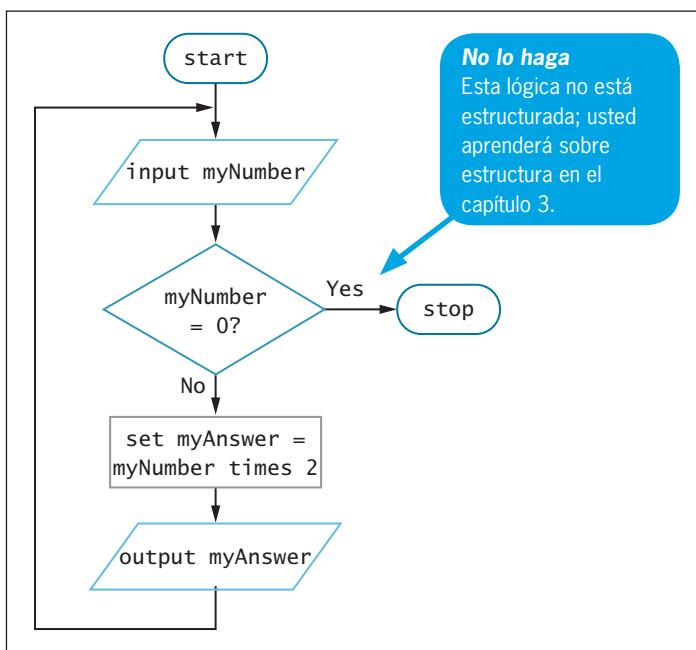


Figura 1-9 Diagrama de flujo del programa de duplicación de números con valor centinela de 0

Un inconveniente de usar 0 para detener un programa, por supuesto, es que no funcionará si el usuario necesita hallar el doble de 0. En este caso, podría seleccionar algún otro valor de

entrada de datos que nunca usará, como 999 o -1, para señalar que el programa debe terminar. Un valor predeterminado que detiene la ejecución de un programa con frecuencia se llama **valor comodín** porque no representa datos reales, sino sólo una señal para detener. En ocasiones, dicho valor se llama **valor centinela** debido a que representa un punto de entrada o salida, como un centinela que vigila una fortaleza.

No todos los programas dependen de la entrada de datos de un usuario desde un teclado; muchos leen los datos de un dispositivo de entrada, como un disco. Cuando las organizaciones guardan datos en un disco u otro dispositivo de almacenamiento en general no usan un valor comodín para señalar el final de un archivo. Por una parte, un registro de entrada podría tener cientos de campos y si almacena un registro comodín en cada archivo desperdiciará una gran cantidad de almacenamiento en “no datos”. Además, con frecuencia es difícil elegir valores centinela para los campos en los archivos de datos de una compañía. Cualquier `balanceDue`, incluso un cero o un número negativo, puede ser un valor legítimo, y cualquier `customer-Name`, incluso “ZZ”, podría ser el nombre de alguien. Por suerte, los lenguajes de programación reconocen el fin de los datos en un archivo de manera automática, por medio de un código que es almacenado en ese punto. Muchos lenguajes de programación usan el término **eof** (por el inglés *end of file* [*final del archivo*]) para referirse a este marcador que actúa en forma automática como un centinela. Este libro, por consiguiente, usa `eof` para indicar el final de los datos siempre que el uso de un valor comodín sea poco práctico o inconveniente. En el diagrama de flujo que se muestra en la figura 1-10, se sombra la pregunta `eof`.

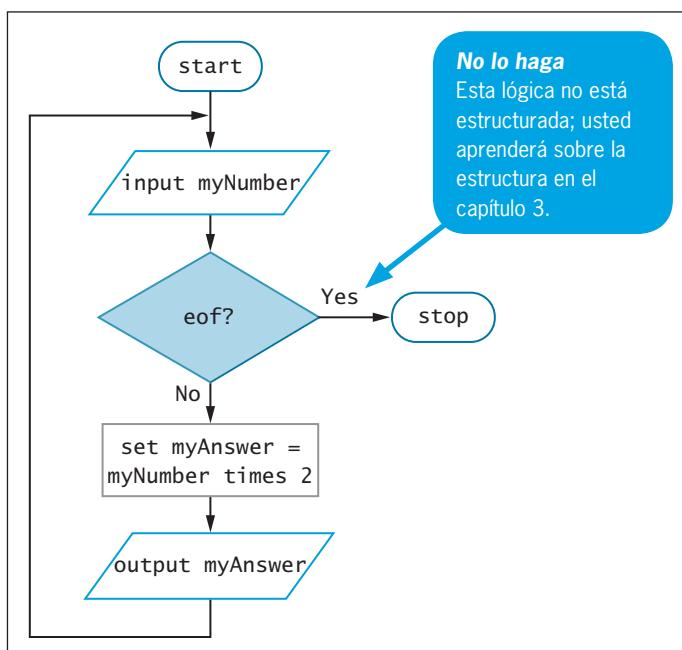


Figura 1-10 Diagrama de flujo que usa `eof`

DOS VERDADES Y UNA MENTIRA

Uso de un valor centinela para terminar un programa

1. Un programa que contiene un ciclo infinito nunca termina.
2. Un valor predeterminado que detiene la ejecución de un programa con frecuencia se llama valor comodín o valor centinela.
3. Muchos lenguajes de programación usan el término `fe` (por *file end [fin de archivo]*) para referirse a un marcador que actúa de manera automática como centinela.

La afirmación falsa es la número 3. El término `eof` (por *end of file*) es el término que más se usa para un centinela de archivo.

Comprensión de la programación y los ambientes del usuario

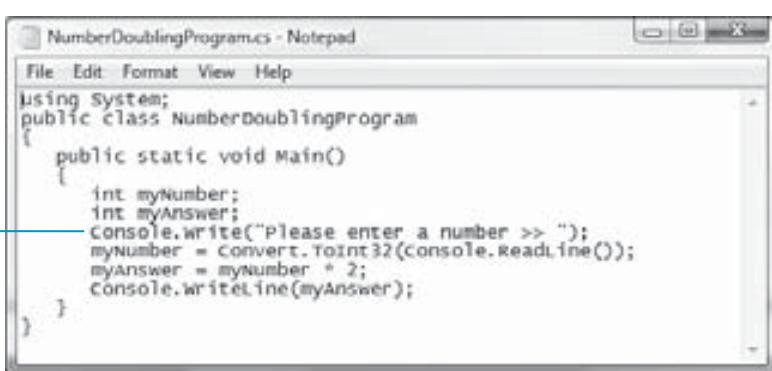
Es posible usar muchos enfoques para escribir y ejecutar un programa de computadora. Cuando usted planea la lógica de uno de ellos puede usar un diagrama de flujo, un seudocódigo o una combinación de ambos. Cuando codifica el programa, puede teclear las declaraciones en una variedad de editores de texto. Cuando su programa se ejecuta podría aceptar entradas de un teclado, ratón, micrófono o cualquier otro dispositivo de entrada, y cuando proporcione la salida podría usar texto, imágenes o sonido. Esta sección describe los ambientes más comunes que encontrará como programador que recién inicia.

Comprensión de los ambientes de programación

Cuando usted planea la lógica para un programa de computadora puede usar papel y lápiz para crear un diagrama de flujo o software que le permita manipular las modalidades de dichos diagramas. Si decide escribir un seudocódigo puede hacerlo a mano o con un programa de procesamiento de palabras. Para introducir el programa en una computadora de modo que lo traduzca y ejecute, por lo general usará un teclado para mecanografiar las declaraciones del programa en un editor. Puede hacerlo en uno de los siguientes:

- Un editor de texto sencillo
- Uno que sea parte de un ambiente de desarrollo integrado

Un **editor de texto** es un programa que se usa para crear archivos de texto sencillos. Es parecido a un procesador de palabras, pero sin tantas características. Usted puede usar alguno como Notepad, que está incluido en Microsoft Windows. La figura 1-11 muestra un programa C# en Notepad que acepta un número y lo duplica. Una ventaja de usar un editor de texto sencillo para mecanografiar y guardar un programa es que el programa completado no requiere mucho espacio del disco para almacenamiento. Por ejemplo, el archivo que se muestra en la figura 1-11 sólo ocupa 314 bytes.



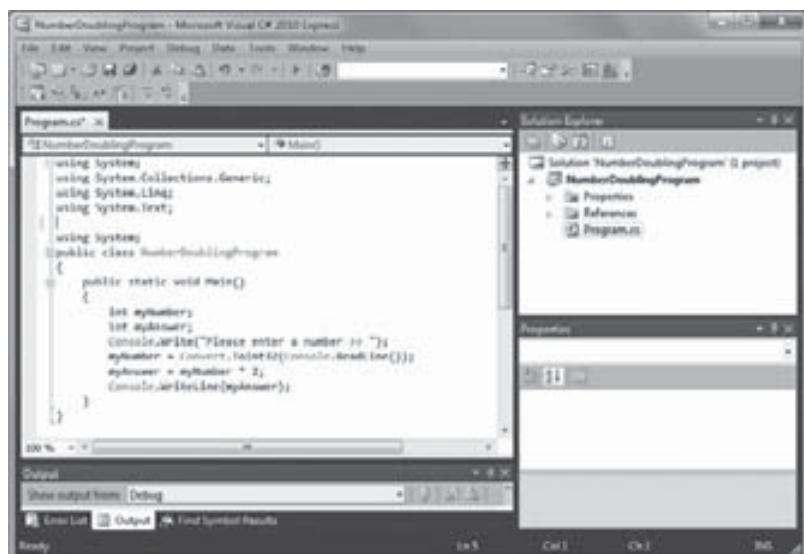
NumberDoublingProgram.cs - Notepad

```
File Edit Format View Help
using System;
public class NumberDoublingProgram
{
    public static void Main()
    {
        int myNumber;
        int myAnswer;
        Console.WriteLine("Please enter a number >> ");
        myNumber = Convert.ToInt32(Console.ReadLine());
        myAnswer = myNumber * 2;
        Console.WriteLine(myAnswer);
    }
}
```

Esta línea contiene un indicador que dice al usuario qué introducir. Usted aprenderá más sobre los indicadores en el capítulo 2.

Figura 1-11 Un programa C# para duplicar números en Notepad

Usted puede usar el editor de un **ambiente de desarrollo integrado (IDE, integrated development environment)** para introducir su programa. Un IDE es un paquete de software que proporciona un editor, compilador y otras herramientas de programación. Por ejemplo, la figura 1-12 muestra un programa C# en el **Microsoft Visual Studio IDE**, un ambiente que contiene herramientas útiles para crear programas en Visual Basic, C++ y C#.



NumberDoublingProgram - Microsoft Visual Studio 2010 Express

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace NumberDoublingProgram
{
    public static void Main()
    {
        int myNumber;
        int myAnswer;
        Console.WriteLine("Please enter a number >> ");
        myNumber = Convert.ToInt32(Console.ReadLine());
        myAnswer = myNumber * 2;
        Console.WriteLine(myAnswer);
    }
}
```

Solution Explorer

- Solution NumberDoublingProgram (1 project)
 - NumberDoublingProgram
 - Properties
 - References
 - Program.cs

Properties

Figura 1-12 Un programa C# para duplicar números en Visual Studio

Usar un IDE es útil para los programadores porque por lo general proporciona características similares a las que se encuentran en muchos procesadores de palabras. En particular, un editor de IDE por lo común tiene características como las siguientes:

- Usa diferentes colores para desplegar varios componentes del lenguaje, lo que facilita la identificación de elementos como tipos de datos.
- Resalta errores de sintaxis en forma visual para usted.
- Emplea la terminación automática de las declaraciones; cuando empieza a teclear una el IDE sugiere una culminación probable, que usted puede aceptar con sólo oprimir una tecla.
- Proporciona herramientas que le permiten seguir paso a paso la ejecución de una declaración del programa a la vez de modo que puede seguir con más facilidad la lógica del mismo y determinar la fuente de cualquier error.

Cuando usa el IDE para crear y guardar un programa ocupa mucho más espacio en disco que cuando usa un editor de texto sencillo. Por ejemplo, el programa en la figura 1-12 ocupa más de 49,000 bytes de espacio de disco.

Aunque varios ambientes de programación podrían verse diferentes y ofrecer características distintas, el proceso para usarlos es muy parecido. Cuando usted planea la lógica para un programa usando un seudocódigo o un diagrama de flujo, no importa cuál ambiente de programación utilice para escribir su código, y cuando escribe el código en un lenguaje de programación, no importa cuál ambiente use para hacerlo.

Comprensión de los ambientes de usuario

Un usuario podría ejecutar un programa que usted ha escrito en cualquier cantidad de ambientes. Por ejemplo, alguien podría hacerlo para duplicar números desde una línea de comandos como la que se muestra en la figura 1-13. Una **línea de comandos** es una ubicación en la pantalla de su computadora en la que usted teclea entradas de texto para comunicarse con el sistema operativo de la computadora. En el programa de la figura 1-13, se pide al usuario un número y se despliegan los resultados.

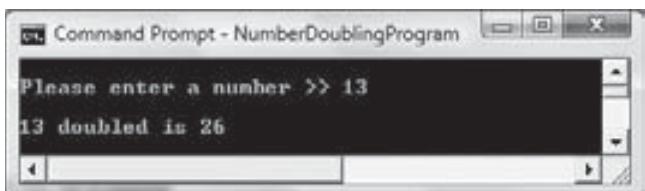


Figura 1-13 Ejecución de un programa para duplicar números en un ambiente de línea de comandos

Muchos programas no se ejecutan en la línea de comandos en un ambiente de texto, pero lo hacen usando una **interfaz gráfica del usuario**, o **GUI (graphical user interface)**, que permite a los usuarios interactuar con un programa en un ambiente gráfico. Cuando se ejecuta un programa GUI, el usuario podría teclear entradas en un cuadro de texto o usar un ratón u otro dispositivo apuntador para seleccionar opciones en la pantalla. La figura 1-14 muestra un programa para duplicar números que ejecuta exactamente la misma tarea que el de la figura 1-13, pero con una GUI.



Figura 1-14 Ejecución de un programa para duplicar números en un ambiente GUI

Un programa de línea de comandos y uno GUI podrían escribirse en el mismo lenguaje de programación. (Por ejemplo, los programas que se muestran en las figuras 1-13 y 1-14 se escribieron con C#.) No obstante, sin importar cuál ambiente se use para escribir o ejecutar un programa, el proceso lógico es el mismo. Los dos programas en las figuras 1-13 y 1-14 aceptan entradas, ejecutan la multiplicación y la salida. En este libro, usted no se concentrará en saber cuál ambiente se usa para teclear las declaraciones de un programa ni se preocupará por el tipo de ambiente que verá el usuario. En cambio, se enfocará en la lógica que se aplica a todas las situaciones de programación.

DOS VERDADES Y UNA MENTIRA

Comprensión de la programación y los ambientes de usuario

1. Puede teclear un programa en un editor que es parte de un ambiente de desarrollo integrado, pero usar un editor de texto sencillo le proporciona más ayuda en la programación.
2. Cuando un programa corre o se ejecuta desde la línea de comandos, un usuario teclea el texto para proporcionar la entrada.
3. Aunque los ambientes GUI y de línea de comandos se ven diferentes, la lógica de entrada, procesamiento y salida se aplica a ambos tipos de programa.

La afirmación falsa es la número 1. Un ambiente de desarrollo integrado proporciona más ayuda de programación que un editor de texto sencillo.

Comprensión de la evolución de los modelos de programación

Las personas han escrito programas de computadora modernos desde la década de 1940. Los lenguajes de programación más antiguos requerían que los programadores trabajaran con direcciones de memoria y que memorizaran códigos incómodos asociados con los lenguajes de máquina. Los lenguajes de programación más recientes se parecen mucho más al lenguaje natural y son más fáciles de usar, en parte debido a que permiten a los programadores nombrar variables en lugar de usar direcciones de memoria poco manejables. Además, los

lenguajes de programación más novedosos permiten crear módulos o segmentos de programa autónomos que pueden armarse en diversas formas. Los programas de computadora más antiguos se escribían en una pieza, de principio a fin, pero los modernos rara vez se escriben así; son creados por equipos de programadores, y cada equipo desarrolla procedimientos de programa reutilizables y conectables. Escribir varios módulos pequeños es más fácil que escribir un programa grande, y la mayor parte de las tareas grandes son más fáciles cuando usted las divide para trabajar en unidades y hacer que otros colegas ayuden con algunas de ellas.



Ada Byron Lovelace predijo el desarrollo del software en 1843; con frecuencia se considera como la primera programadora. Las bases para la mayor parte del software moderno fueron propuestas por Alan Turing en 1935.

En la actualidad, los programadores usan dos modelos o paradigmas principales para desarrollar programas y sus procedimientos:

- La **programación procedural** se enfoca en los procedimientos que crean los programadores. Es decir, los programadores procedimentales se centran en las acciones que se llevan a cabo; por ejemplo, obtener datos de entrada para un empleado y escribir los cálculos necesarios para generar un cheque de pago a partir de los datos. Los programadores procedimentales enfocarían la generación del cheque dividiendo el proceso en subtareas manejables.
- La **programación orientada hacia los objetos** se enfoca en los objetos o “cosas” y describe sus características (también llamadas atributos) y comportamientos. Por ejemplo, los programadores orientados hacia los objetos podrían diseñar una aplicación de nómina pensando en los empleados y cheques de pago, y describiendo sus atributos. Los empleados tienen nombres y números de seguro social y los cheques de pago contienen los nombres y las cantidades del cheque. Luego los programadores pensarán en los comportamientos de los empleados y los cheques de pago, como que los empleados obtienen aumentos y agregan dependientes y los cheques de pago son calculados y producidos. Los programadores orientados hacia los objetos construirán entonces aplicaciones a partir de estas entidades.

Con cualquier enfoque, procedural u orientado hacia los objetos, usted puede generar un cheque de pago correcto y ambos modelos emplean módulos de programa reutilizables. La diferencia principal está en el enfoque que adopta el programador durante las primeras etapas de la planeación de un proyecto. Por ahora, este libro se enfoca en las técnicas de programación procedural. Las habilidades que obtenga al aplicar este tipo de programación (declarar variables, aceptar entradas, tomar decisiones, producir salidas, etc.) le servirán en gran medida ya sea que al final escriba los programas con un enfoque procedural, uno orientado hacia los objetos, o ambos. El lenguaje de programación en el que escriba su código fuente podría determinar su enfoque. Puede escribir un programa procedural en cualquier lenguaje que soporte orientación a objetos, pero lo opuesto no siempre es cierto.

DOS VERDADES Y UNA MENTIRA

Comprensión de la evolución de los modelos de programación

1. Los programas de computadora más antiguos se escribían en muchos módulos separados.
2. Los programadores procedimentales se enfocan en las acciones que un programa lleva a cabo.
3. Los programadores orientados hacia los objetos se centran en los objetos de un programa y sus atributos y comportamientos.

La afirmación falsa es la número 1. Los programas más antiguos se escribían en una sola pieza; los más recientes se dividen en módulos.

Resumen del capítulo

- Juntos, el hardware (dispositivos físicos) y el software (instrucciones) realizan tres operaciones importantes: entrada, procesamiento y salida. Las instrucciones para la computadora se escriben en un lenguaje de programación que requiere una sintaxis específica; un compilador o intérprete traduce las instrucciones al lenguaje de máquina. Cuando la sintaxis y la lógica de un programa son correctas, usted puede correr o ejecutar el programa para obtener los resultados deseados.
- Para que un programa funcione en forma apropiada, usted debe desarrollar una lógica correcta. Es mucho más difícil localizar los errores lógicos que los de sintaxis.
- La labor de un programador implica entender el problema, planear la lógica, codificar el programa, traducirlo a lenguaje de máquina, probarlo, ponerlo en producción y mantenerlo.
- Cuando los programadores planean la lógica de una solución para un problema de programación con frecuencia usan diagramas de flujo o seudocódigo. Cuando usted traza un diagrama de flujo usa paralelogramos para representar las operaciones de entrada y salida, y rectángulos para representar el procesamiento. Los programadores también toman decisiones para controlar la repetición de los conjuntos de instrucciones.
- Para evitar la creación de un ciclo infinito cuando usted repite las instrucciones puede probar un valor centinela. Se representa una decisión en un diagrama de flujo al dibujar un símbolo en forma de diamante que contiene una pregunta cuya respuesta es sí o no.
- Usted puede teclear un programa en un editor de texto sencillo o uno que sea parte de un ambiente de desarrollo integrado. Cuando los valores de datos de un programa se introducen desde un teclado, pueden ingresarse en la línea de comandos en un ambiente de texto o en una GUI. De cualquier forma, la lógica es similar.

- Los programadores procedimentales y orientados hacia los objetos enfocan los problemas de manera diferente. Los procedimentales se concentran en las acciones que se ejecutan con los datos. Los orientados hacia los objetos se enfocan en los objetos, sus comportamientos y atributos.

28

Términos clave

Un **sistema de cómputo** es una combinación de todos los componentes que se requieren para procesar y almacenar datos usando una computadora.

El **hardware** es el conjunto de dispositivos físicos que componen un sistema de cómputo.

El **software** consiste en los programas que indican a la computadora qué debe hacer.

Los **programas** son los conjuntos de instrucciones para una computadora.

La **programación** es el acto de desarrollar y escribir programas.

El **software de aplicación** comprende todos los programas que usted aplica a una tarea.

El **software de sistema** comprende los programas que usted usa para manejar su computadora.

La **entrada** describe la introducción de elementos de datos en la memoria de la computadora por medio de los dispositivos de hardware como teclados y ratones.

Los **elementos de datos** incluyen todo el texto, los números y otra información procesada por una computadora.

El **procesamiento** de elementos de datos implica organizarlos, comprobar su precisión o realizar operaciones matemáticas en ellos.

La **unidad central de procesamiento**, o **CPU**, es el componente de hardware que procesa los datos.

La **salida** describe la acción de recuperar información de la memoria y enviarla a un dispositivo, como un monitor o impresora, de modo que las personas puedan ver, interpretar y trabajar con los resultados.

Información son los datos procesados.

Los **dispositivos de almacenamiento** son los tipos de equipo de hardware, como discos, que contienen información para su recuperación posterior.

Los **lenguajes de programación**, como Visual Basic, C#, C++, Java o COBOL, se usan para escribir los programas.

El **código de programa** es el conjunto de instrucciones que un programador escribe en un lenguaje de programación.

Codificar el programa es la acción de escribir instrucciones en lenguaje de programación.

La **sintaxis** de un lenguaje son sus reglas gramaticales.

Un **error de sintaxis** es un error en el lenguaje o la gramática.

La **memoria de la computadora** es el almacenamiento interno temporal dentro de una computadora.

La **memoria de acceso aleatorio (RAM)** es el almacenamiento interno temporal de la computadora.

El término **volátil** describe el almacenamiento cuyo contenido se pierde cuando la energía eléctrica se interrumpe.

El término **no volátil** describe el almacenamiento cuyo contenido se conserva cuando la energía eléctrica se interrumpe.

El **lenguaje de máquina** es un lenguaje de circuitería encendido/apagado de una computadora.

El **código fuente** son las declaraciones que un programador escribe en un lenguaje de programación.

Código objeto es lenguaje de máquina traducido.

Un **compilador o intérprete** traduce un lenguaje de alto nivel a uno de máquina e indica si ha usado en forma incorrecta un lenguaje de programación.

El **lenguaje binario** se representa usando una serie de 0 (ceros) y 1 (unos).

Correr o ejecutar un programa significa que se llevan a cabo sus instrucciones.

Los **lenguajes de programación interpretados** (también llamados **lenguajes de programación de scripting o lenguajes de script**) como Python, Lua, Perl y PHP se usan para escribir programas que se introducen en forma directa desde un teclado. Los lenguajes de programación interpretados se almacenan como texto y no como archivos ejecutables binarios.

Un **error lógico** ocurre cuando se ejecutan instrucciones incorrectas, o cuando éstas se efectúan en el orden incorrecto.

Usted desarrolla la **lógica** del programa de computadora cuando da instrucciones a la computadora en una secuencia específica, sin omitir alguna instrucción ni agregar instrucciones superfluas.

Una **variable** es una ubicación de memoria nombrada cuyo valor puede variar.

El **ciclo de desarrollo del programa** consiste en los pasos que se siguen durante la vida de un programa.

Los **usuarios** (o **usuarios finales**) son personas que emplean los programas de computadora y obtienen beneficios de ellos.

La **documentación** consiste en todo el papeleo de soporte para un programa.

Un **algoritmo** es la secuencia de pasos necesarios para resolver cualquier problema.

Una **gráfica IPO** es una herramienta de desarrollo de programas que define las tareas de entrada, procesamiento y salida.

Una **gráfica TOE** es una herramienta de desarrollo de programas que lista tareas, objetos y eventos.

Prueba de escritorio es el proceso de recorrer la solución de un programa en papel.

Un **lenguaje de programación de alto nivel** soporta sintaxis en inglés.

Un **lenguaje de máquina de bajo nivel** está formado por 1 (unos) y 0 (ceros) y no usa nombres de variables que se interpretan con facilidad.

Depuración es el proceso de hallar y corregir errores del programa.

Conversión es el conjunto entero de acciones que debe emprender una organización para cambiar al uso de un programa o conjunto de programas nuevos.

El **mantenimiento** consiste en todas las mejoras y correcciones hechas a un programa después de que está en producción.

El **seudocódigo** es una representación en inglés de los pasos lógicos que se requieren para resolver un problema.

Un **símbolo de entrada** indica una operación de entrada y en los diagramas de flujo se representa con un paralelogramo.

Un **símbolo de procesamiento** indica una operación de procesamiento y en los diagramas de flujo se representa con un rectángulo.

Un **símbolo de salida** indica una operación de salida y en los diagramas de flujo se representa con un paralelogramo.

Un **símbolo de entrada/salida** o **símbolo I/O** en los diagramas de flujo se representa con un paralelogramo.

Las **líneas de flujo**, o flechas, conectan los pasos en un diagrama de flujo.

Un **símbolo terminal** indica el inicio o el fin de un segmento de diagrama de flujo y se representa con una pastilla.

Un **ciclo** es una repetición de una serie de pasos.

Un **ciclo infinito** ocurre cuando la lógica que se repite no puede terminar.

Tomar una decisión es el acto de probar un valor.

Un **símbolo de decisión** tiene forma de diamante y en los diagramas de flujo se usa para representar una decisión.

Un **valor comodín** es un valor predeterminado que detiene la ejecución de un programa.

Un **valor centinela** es un valor predeterminado que detiene la ejecución de un programa.

El término **eof** significa *fin del archivo*.

Un **editor de texto** es un programa que se usa para crear archivos de texto sencillos; es parecido a un procesador de palabras, pero sin tantas características.

Un **ambiente de desarrollo integrado (IDE)** es un paquete de software que proporciona un editor, un compilador y otras herramientas de programación.

Microsoft Visual Studio IDE es un paquete de software que contiene herramientas útiles para crear programas en Visual Basic, C++ y C#.

Una **línea de comandos** es una ubicación en la pantalla de su computadora en la que teclea entradas de texto para comunicarse con el sistema operativo de la computadora.

Una **interfaz gráfica del usuario**, o **GUI**, permite a los usuarios interactuar con un programa en un ambiente gráfico.

La **programación procedimental** es un modelo de programación que se enfoca en los procedimientos que crean los programadores.

La **programación orientada hacia los objetos** es un modelo de programación que se enfoca en objetos, o “cosas”, y describe sus características (también llamadas atributos) y comportamientos.

Preguntas de repaso

8. La tarea más importante de un programador antes de planear la lógica de un programa es _____.
 - a) decidir cuál es el lenguaje de programación que va a usar
 - b) codificar el problema
 - c) capacitar a los usuarios del programa
 - d) entender el problema
9. Las dos herramientas que más se utilizan para planear la lógica de un programa son _____.
 - a) diagramas de flujo y pseudocódigo
 - b) ASCII y EBCDIC
 - c) Java y Visual Basic
 - d) procesadores de palabras y hojas de cálculo
10. Escribir un programa en un lenguaje como C++ o Java se conoce como _____ el programa.

a) traducir	c) interpretar
b) codificar	d) compilar
11. Un lenguaje de programación en inglés como Java o Visual Basic es un lenguaje de programación _____.

a) de nivel de máquina	c) de alto nivel
b) de bajo nivel	d) de nivel binario
12. ¿Cuál de los siguientes es un ejemplo de un error de sintaxis?
 - a) producir la salida antes de aceptar la entrada
 - b) restar cuando quiere sumar
 - c) escribir mal una palabra del lenguaje de programación
 - d) todos los anteriores
13. ¿Cuál de los siguientes es un ejemplo de un error lógico?
 - a) ejecutar la aritmética con un valor antes de introducirlo
 - b) aceptar dos valores de entrada cuando un programa sólo requiere uno
 - c) dividir entre 3 cuando se quiere dividir entre 30
 - d) todos los anteriores
14. El paralelogramo es el símbolo de diagrama de flujo que representa _____.

a) la entrada	c) tanto a como b
b) la salida	d) ninguno de los anteriores

15. En un diagrama de flujo, un rectángulo representa _____.
a) la entrada c) una pregunta
b) un centinela d) el procesamiento

16. En los diagramas de flujo, el símbolo de decisión es _____.
a) un paralelogramo c) una pastilla
b) un rectángulo d) un diamante

17. El término **eof** representa _____.
a) un dispositivo de entrada estándar
b) un valor centinela genérico
c) una condición en la cual no hay más memoria disponible para el almacenamiento
d) el flujo lógico en un programa

18. Cuando se usa un IDE en lugar de un editor de texto simple para desarrollar un programa, _____.
a) la lógica es más complicada c) la sintaxis es diferente
b) la lógica es más sencilla d) se proporciona alguna ayuda

19. Cuando se escribe un programa que correrá en un ambiente GUI en contraposición a un ambiente de línea de comandos, _____.
a) la lógica es muy diferente c) no necesita planear la lógica
b) alguna sintaxis es diferente d) los usuarios se confunden más

20. En comparación con la programación procedimental, con la programación orientada a objetos, _____.
a) el enfoque del programador difiere
b) no pueden usarse algunos lenguajes, como Java
c) no se aceptan entradas
d) no se codifican cálculos; ellos se crean en forma automática

Ejercicios

1. Relacione la definición con el término apropiado.
 1. Dispositivos del sistema de cómputo a) compilador
 2. Otra palabra para *programas* b) sintaxis
 3. Reglas del lenguaje c) lógica
 4. Orden de las instrucciones d) hardware
 5. Traductor de lenguaje e) software
 2. En sus propias palabras, describa los pasos para escribir un programa de computadora.

3. Relacione el término con la forma apropiada (véase la figura 1-15).

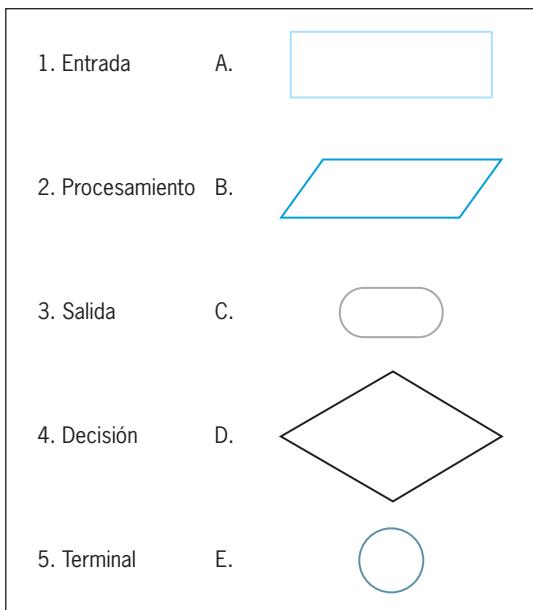


Figura 1-15 Identificación de formas

4. Trace un diagrama de flujo o escriba un seudocódigo para representar la lógica de un programa que permita al usuario introducir un valor. El programa divide el valor entre 2 y da salida al resultado.
5. Trace un diagrama de flujo o escriba un seudocódigo para representar la lógica de un programa que permita al usuario introducir un valor para una arista de un cubo. El programa calcula el área de la superficie de un lado del cubo, el área de la superficie del cubo y su volumen. El programa da salida a todos los resultados.
6. Trace un diagrama de flujo o escriba un seudocódigo para representar la lógica de un programa que permite al usuario introducir dos valores. El programa da salida al producto de los dos valores.
7. a) Trace un diagrama de flujo o escriba un seudocódigo para representar la lógica de un programa que permita al usuario introducir valores para el ancho y el largo del piso de un salón en metros. El programa da salida al área del piso en metros cuadrados.
b) Modifique el programa que calcula el área del piso para que calcule y dé salida al número de mosaicos de 30 centímetros cuadrados que se necesitan para cubrir el piso.

8. a) Trace un diagrama de flujo o escriba un seudocódigo para representar la lógica de un programa que permita al usuario introducir valores para el ancho y el largo de una pared en metros. El programa da salida al área de la pared en metros cuadrados.
- b) Modifique el programa que calcula el área de la pared para permitir al usuario introducir el precio de 1 galón de pintura. Suponga que 1 galón de pintura cubre 35 metros cuadrados de una pared. El programa da salida al número de galones necesarios y al costo del trabajo. (Para este ejercicio suponga que no necesita tomar en cuenta las ventanas o puertas y que puede comprar galones de pintura en partes.)
9. Investigue las tasas actuales de cambio monetario. Trace un diagrama de flujo o escriba un seudocódigo para representar la lógica de un programa que permita al usuario introducir un número de dólares y convertirlos en euros y en yenes japoneses.
10. Trace un diagrama de flujo o escriba un seudocódigo para representar la lógica de un programa que permita al usuario introducir valores para el salario base, las ventas totales y la tasa de comisión de un vendedor. El programa calcula y da salida al pago del vendedor agregando el salario base al producto de las ventas totales y la tasa de comisión.



Encuentre los errores

11. Desde los primeros días de la programación de computadoras, a los errores de programa se les ha llamado *bugs* (insectos). Se dice con frecuencia que el término se originó por una polilla real que se descubrió atrapada en los circuitos de una computadora en la Universidad de Harvard en 1945. En realidad, el término *bug* se usaba antes de 1945 para referirse a los problemas con cualquier aparato eléctrico; incluso durante la vida de Thomas Edison significaba un defecto industrial. Sin embargo, el término depuración (*debugging*) se asocia más con la corrección de errores de sintaxis y lógicos de un programa que con cualquier otro tipo de problema.

Sus archivos descargables para el capítulo 1 incluyen DEBUG01-01.txt, DEBUG01-02.txt y DEBUG01-03.txt. Cada archivo comienza con algunos comentarios (líneas que comienzan con dos barras oblicuas) que describen el programa. Examine el seudocódigo que sigue a los comentarios introductorios, luego halle y corrija todos los errores. (NOTA: Estos archivos se encuentran disponibles sólo para la versión original en inglés.)



Zona de juegos

12. En 1952, A. S. Douglas escribió su tesis doctoral en la Universidad de Cambridge sobre la interacción humano-computadora y creó el primer juego gráfico de computadora, una versión de Tic-Tac-Toe. El juego fue programado en una computadora mainframe EDSAC de tubo de vacío. Por lo general se supone que el

primer juego de computadora fue *Spacewar!*, que se creó en 1962 en el MIT; el primer videojuego disponible en forma comercial fue *Pong*, introducido por Atari en 1972. En 1980, *Asteroids* y *Lunar Lander*, de Atari, se convirtieron en los primeros videojuegos que se registraron en la U. S. Copyright Office. A lo largo de la década de 1980, las personas pasaron horas con juegos que ahora parecen muy simples y sin glamour; ¿recuerda haber jugado *Adventure*, *Oregon Trail*, *Where in the World Is Carmen San-diego?* o *Myst*?

En la actualidad, los juegos de computadora comerciales son mucho más complejos; su desarrollo requiere muchos programadores, artistas gráficos y probadores, y se necesita mucho personal administrativo y de mercadotecnia para promoverlos. Desarrollar y comercializar un juego podría costar muchos millones de dólares, pero uno que sea exitoso podría ganar cientos de millones de dólares. Obviamente, con la breve introducción a la programación que tuvo en este capítulo no espere crear un juego muy complejo. Sin embargo, puede empezar.

Mad Libs[®] es un juego para niños en el que los participantes proporcionan algunas palabras que luego son incorporadas en una historia ridícula. El juego ayuda a los niños a entender diferentes partes del habla porque se les pide que proporcionen tipos específicos de palabras. Por ejemplo, podría pedir a un niño un sustantivo, otro sustantivo, un adjetivo y un verbo en tiempo pasado. El niño respondería con palabras como *mesa*, *libro*, *ridículo* y *estudió*. Un Mad Lib recién creado podría ser:

María tenía una pequeña *mesa*

Su *libro* era *ridículo* como la nieve

Y en todas partes que María *estudió*

La *mesa* de seguro la acompañó.

Cree la lógica para un programa Mad Lib que acepte cinco palabras de entrada, luego cree y despliegue una historia breve o un verso infantil que las use.



Para discusión

13. ¿Cuál es la mejor herramienta para aprender programación: los diagramas de flujo o un seudocódigo? Cite cualquier investigación educativa que pueda encontrar.
14. ¿Cuál es la imagen del programador de computadoras en la cultura popular? ¿Es diferente en los libros que en los programas de televisión y las películas? ¿Le gustaría tener esa imagen?

CAPÍTULO

2

Elementos de los programas de alta calidad

En este capítulo usted aprenderá sobre:

- ◎ La declaración y el uso de variables y constantes
- ◎ La realización de operaciones aritméticas
- ◎ Las ventajas de la modularización
- ◎ Modularizar un programa
- ◎ Las gráficas de jerarquía
- ◎ Las características de un buen diseño de programa

La declaración y el uso de variables y constantes

Como aprendió en el capítulo 1, los elementos de datos incluyen todo el texto, los números y otra información que son procesados por una computadora. Cuando usted introduce los elementos de datos en un ordenador, éstos se almacenan en variables en la memoria, donde se procesan y se convierten en información de salida.

Cuando usted escribe programas trabaja con los datos en tres formas diferentes: literales (o constantes no nombradas), variables y constantes nombradas.

Comprensión de las constantes literales y sus tipos de datos

Todos los lenguajes de programación soportan dos amplios tipos de datos: el **numérico** describe los datos que consisten en números y la **cadena** los que no son numéricos. La mayoría de los lenguajes de programación soportan varios tipos de datos adicionales, incluidos los tipos múltiples para valores numéricos de diferentes tamaños y con y sin lugares decimales. Los lenguajes como C++, C#, Visual Basic y Java distinguen entre las variables numéricas **enteras** (número entero) y las variables numéricas de **punto flotante** (fraccionarias) que tienen un punto decimal. (Los números de punto flotante también se conocen como **números reales**.) Por tanto, en algunos lenguajes los valores 4 y 4.3 se almacenarían en tipos diferentes de variables numéricas. Además, muchos lenguajes permiten la distinción entre los valores más pequeños y más grandes que ocupan diferentes cantidades de bytes en la memoria. Usted aprenderá más sobre estos tipos de datos especializados cuando estudie un lenguaje de programación, pero este libro usa los dos más amplios: numérico y cadena.

Cuando usted usa un valor numérico específico, como 43, en un programa, lo escribe usando los dígitos y sin comillas. Un valor numérico específico con frecuencia se llama **constante numérica** (o **constante numérica literal**) debido a que no cambia; un 43 siempre tiene el valor 43. Cuando se almacena un valor numérico en la memoria de la computadora no se introducen o almacenan caracteres adicionales como signos de pesos y comas. Esos caracteres pueden agregarse a la salida con fines de legibilidad, pero no son parte del número.

Un valor de texto específico, o cadena de caracteres, como “Amanda”, es una **constante de cadena** (o **constante de cadena literal**). Las constantes de cadena, a diferencia de las constantes numéricas, aparecen entre comillas en los programas. Los valores de cadena también se llaman **valores alfanuméricos** porque pueden contener caracteres alfabéticos al igual que números y otros caracteres. Por ejemplo, “\$3,215.99 U.S.”, incluido el signo de dólares, la coma, el punto, las letras y los números, es una cadena. Aunque las cadenas pueden contener números, los valores numéricos no contienen caracteres alfabéticos. La constante numérica 43 y la constante de cadena “Amanda” son ejemplos de **constants literales**, no tienen identificadores como las variables.

Trabajo con variables

Las variables son ubicaciones de memoria nombradas cuyo contenido varía o difiere con el tiempo. Por ejemplo, en el programa para duplicar números en la figura 2-1, `myNumber` y `myAnswer` son variables. En cualquier momento en el tiempo, una variable sólo contiene un valor. En ocasiones, `myNumber` contiene 2 y `myAnswer` contiene 4; otras veces, `myNumber` contiene 6 y `myAnswer` 12. La capacidad de las variables de memoria para cambiar de valor es lo que hace que las computadoras y la programación sean valiosas. Debido a que una ubicación de memoria puede usarse en repetidas ocasiones con diferentes valores, es posible escribir las instrucciones del programa una vez y luego usarlas para miles de cálculos separados. *Un* conjunto de instrucciones de nómina en su empresa genera el cheque de pago de cada empleado y *un* conjunto de instrucciones en su compañía eléctrica resulta en la factura de cada hogar.

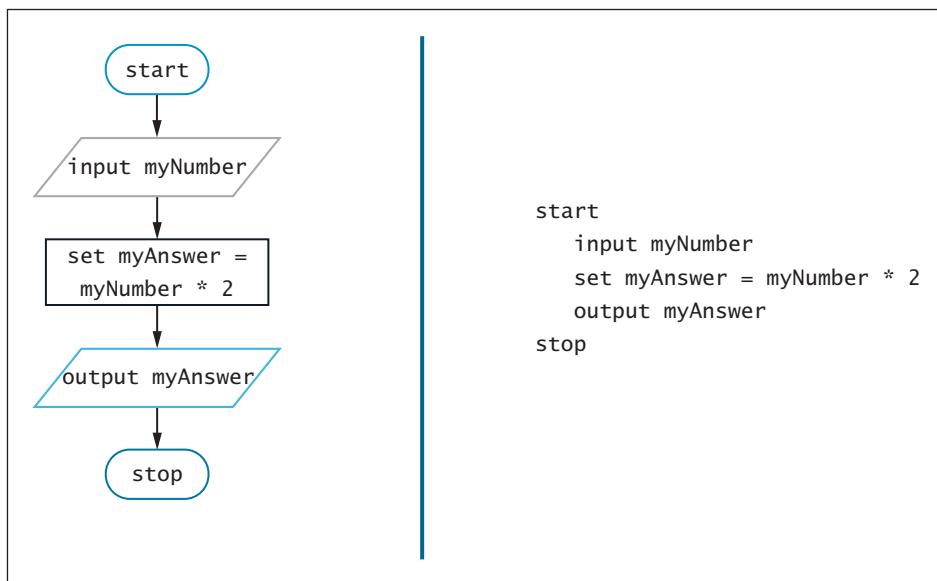


Figura 2-1 Diagrama de flujo y seudocódigo para el programa para duplicar números

En la mayoría de los lenguajes de programación, antes de que usted pueda usar alguna variable debe incluir una declaración para ello. Una **declaración** es un enunciado que proporciona el tipo de datos y un identificador para una variable. Un **identificador** es el nombre de un componente del programa. El **tipo de datos** de un elemento de datos es una clasificación que describe lo siguiente:

- Qué valores puede contener el elemento
- Cómo se almacena el elemento en la memoria de la computadora
- Qué operaciones pueden ejecutarse en el elemento de datos

Como se mencionó antes, casi todos los lenguajes de programación soportan varios tipos de datos, pero en este libro sólo se usarán dos tipos: `num` y `string`.

Cuando usted declara una variable proporciona tanto un tipo de datos como un identificador. De manera opcional, puede declarar un valor inicial para cualquier variable y hacer esto es **inicializar la variable**. Por ejemplo, cada una de las siguientes declaraciones es válida. Dos de ellas incluyen inicializaciones y dos no:

40

```
num mySalary  
num yourSalary = 14.55  
string myName  
string yourName = "Juanita"
```

La figura 2-2 muestra el programa para duplicar números de la figura 2-1 con las declaraciones agregadas sombreadas. Las variables deben declararse antes de que se usen por primera vez en un programa. Algunos lenguajes requieren que todas se declaren al principio del programa; otros permiten que esto se haga en cualquier parte con tal de que sea antes de su primer uso. Este libro seguirá la convención de declarar todas las variables juntas.

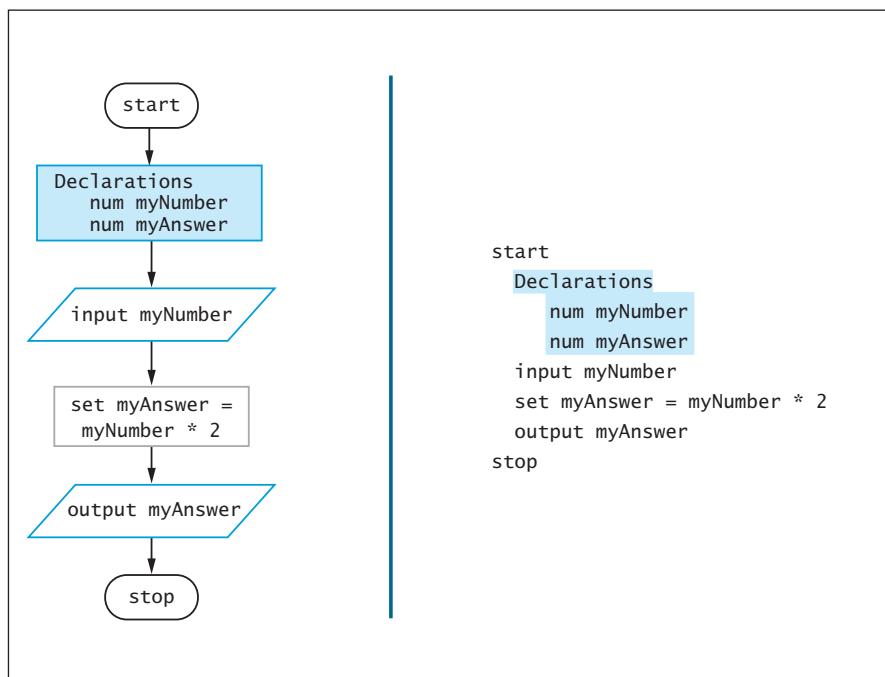


Figura 2-2 Diagrama de flujo y seudocódigo del programa para duplicar números con declaraciones de variables

En muchos lenguajes de programación, si se declara una variable y no se inicializa, ésta contendrá un valor desconocido hasta que se le asigne uno. El valor desconocido de una variable por lo común se llama **basura**. Aunque algunos lenguajes usan un valor por defecto para algunas variables (como asignar 0 a cualquier variable numérica no asignada), en este libro se supondrá que una no asignada contiene basura. En muchos lenguajes es ilegal usar una variable que contenga basura en una declaración aritmética o desplegarla como salida. Aun si usted trabaja con un lenguaje que le permite desplegar basura, esto no sirve para ningún propósito y constituye un error lógico.

Cuando usted crea una variable sin asignarle un valor inicial (como con `myNumber` y `myAnswer` en la figura 2-2), su intención es asignarlo después; por ejemplo, al recibir uno como entrada o colocar ahí el resultado de un cálculo.

Nombramiento de variables

El ejemplo de la duplicación de números en la figura 2-2 requiere dos variables: `myNumber` y `myAnswer`. De manera alternativa, éstas podrían nombrarse `userEntry` y `programSolution`, o `inputValue` y `twiceTheValue`. Como programador usted elige nombres razonables y descriptivos para sus variables. El intérprete del lenguaje asocia luego los nombres que usted elija con direcciones de memoria específicas.

Cada lenguaje de programación tiene su propio conjunto de reglas para crear identificadores. Casi todos los lenguajes permiten letras y dígitos en los identificadores; algunos aceptan guiones en los nombres de las variables, como `hourly-wage`, y otros permiten subrayados, como en `hourly_wage`. Unos aceptan signos de dólar u otros caracteres especiales (por ejemplo, `hourly$`); otros permiten caracteres de alfabetos extranjeros, como π u Ω . Cada lenguaje tiene algunas (quizá de 100 a 200) **palabras clave** reservadas que no se permite usar como nombres de variables porque son parte de la sintaxis del lenguaje. Cuando usted aprenda un lenguaje de programación conocerá su lista de palabras clave.

Los lenguajes diferentes ponen límites distintos sobre la extensión de los nombres de las variables aunque, en general, la de los identificadores en lenguajes más recientes es casi ilimitada. En muchos lenguajes los identificadores son sensibles al uso de mayúsculas y minúsculas, así `HoUrLyWaGe`, `hourlywage` y `hourlyWage` son tres nombres de variables separados. Los programadores usan numerosas convenciones para nombrar variables, con frecuencia dependiendo del lenguaje de programación o de los estándares implementados por sus patrones. Las convenciones comunes incluyen las siguientes:

- La **notación de camello**: la variable empieza con una letra minúscula y cualquier palabra subsiguiente comienza con una mayúscula, como `hourlyWage`. Para los nombres de las variables en este libro se usa la notación de camello.
- La **caja de Pascal**: la primera letra del nombre de una variable es mayúscula, como en `HourlyWage`.
- La **notación húngara**: el tipo de datos de una variable es parte del identificador; por ejemplo, `numHourlyWage` o `stringLastName`.

Adoptar una convención para nombrar variables y usarla en forma consistente le ayudará para que su programa sea más fácil de leer y entender.

Aun cuando cada lenguaje tiene sus propias reglas para nombrar variables usted no debe preocuparse por la sintaxis específica de alguno en particular cuando diseñe la lógica de un programa. La lógica, después de todo, funciona con cualquier lenguaje. Los nombres de las variables que se usan a lo largo de este libro sólo siguen tres reglas:

1. *Los nombres de las variables deben constar de una palabra.* Pueden contener letras, dígitos, guiones, subrayados o cualquier otro carácter que elija, con excepción de espacios. Por consiguiente, `r` es un nombre de variable legal, lo mismo que `rate` e `interestRate`. El nombre `interest rate` no se permite debido al espacio.



2. *Los nombres de las variables deben comenzar con una letra.* Algunos lenguajes de programación permiten que los nombres comiencen con un carácter no alfabético, como un subrayado. Casi todos los lenguajes no aceptan que los nombres de las variables inician con un dígito. Este libro sigue la convención más común de empezar los nombres de las variables con una letra.

Cuando usted escribe un programa usando un editor que viene en paquete con un compilador en un IDE, el compilador puede desplegar los nombres de las variables en un color diferente del resto del programa. Esta ayuda visual distingue los nombres de las variables de las palabras que son parte del lenguaje de programación.

3. *Los nombres de las variables deben tener algún significado apropiado.* Ésta no es una regla formal de los lenguajes de programación. Cuando se calcula una tasa de interés en un programa, a la computadora no le interesa si usted llama a la variable `g`, `u84` o `fred`. En tanto que el resultado numérico se coloque en la variable, su nombre real no importa. Sin embargo, es mucho más fácil seguir la lógica de una declaración como `set interestEarned = initialInvestment * interestRate` que de una como `set f = i * r` o `set someBanana = j89 * myFriendLinda`. Cuando un programa requiere cambios, que podrían hacerse meses o años después de que se escribió la versión original, usted y sus colegas programadores apreciarán los nombres descriptivos claros en lugar de los identificadores confusos. En este capítulo usted aprenderá más sobre la selección de identificadores adecuados.

Note que el diagrama de flujo en la figura 2-2 sigue las reglas anteriores: ambos nombres, `myNumber` y `myAnswer`, son palabras únicas sin espacios y tienen significados apropiados. Algunos programadores nombran a las variables en honor a sus amigos o crean juegos de palabras con ellos, pero los profesionales de la computación consideran dicho comportamiento poco profesional y serio.

Asignación de valores a las variables

Cuando deba crear un diagrama de flujo o un seudocódigo para un programa que duplique números puede incluir una declaración como la siguiente:

```
set myAnswer = myNumber * 2
```

Ésta es una **declaración de asignación** e incorpora dos acciones. Primera, la computadora calcula el valor aritmético de `myNumber * 2`. Segunda, el valor calculado se almacena en la ubicación de memoria `myAnswer`.

El signo de igual es el **operador de asignación**. Éste es un ejemplo de **operador binario**, lo que significa que requiere dos operandos, uno en cada lado. El operador de asignación siempre opera de derecha a izquierda, es decir, tiene **asociatividad derecha** o **asociatividad de derecha a izquierda**. Esto significa que se evalúa primero el valor de una expresión a la derecha del operador de asignación, y luego el resultado se asigna al operando de la izquierda. El operando a la derecha de un operador de asignación puede ser un valor, una fórmula, una constante nombrada o una variable. El operando a la izquierda debe ser un nombre que represente una dirección de memoria, el de la ubicación donde se almacenará el resultado.

Por ejemplo, si usted ha declarado dos variables numéricas llamadas `someNumber` y `someOtherNumber`, entonces cada una de las siguientes es una declaración de asignación válida:

```
set someNumber = 2
set someNumber = 3 + 7
set someOtherNumber = someNumber
set someOtherNumber = someNumber * 5
```

En cada caso, la expresión a la derecha del operador de asignación se evalúa y almacena en la ubicación referenciada en el lado izquierdo. El resultado a la izquierda de un operador de asignación se llama un **lvalue**. La *l* es por izquierda en inglés. Los lvalues siempre son identificadores de una dirección de memoria.

Sin embargo, las siguientes declaraciones *no* son válidas:

```
set 2 + 4 = someNumber
set someOtherNumber * 10 = someNumber
```

No lo haga

El operando a la izquierda de un operador de asignación debe representar una dirección de memoria.

En cada uno de estos casos, el valor a la izquierda de un operador de asignación no es una dirección de memoria, de modo que las declaraciones no son válidas.

Cuando usted escriba un seudocódigo o trace un diagrama de flujo sería útil que en las declaraciones de asignación usara la palabra `set`, como se muestra en estos ejemplos, para enfatizar que se establece el valor del lado izquierdo. Sin embargo, en la mayoría de los lenguajes de programación no se usa la palabra `set` y las declaraciones de asignación adoptan la siguiente forma más sencilla:

```
someNumber = 2
someOtherNumber = someNumber
```

Debido a que las asignaciones en la mayoría de los lenguajes aparecen en la forma abreviada, así se usa en el resto de este libro.

Comprensión de los tipos de datos de las variables

Las computadoras manejan los datos de cadena de manera diferente de lo que lo hacen con los datos numéricos. Usted tal vez se ha percatado de estas diferencias si ha usado software de aplicación, como hojas de cálculo o bases de datos. Por ejemplo, en una hoja de cálculo no puede sumar una columna de palabras. Del mismo modo, cada lenguaje de programación requiere que especifique el tipo correcto para cada variable y que use cada tipo de manera apropiada.

- Una **variable numérica** es aquella que puede contener dígitos y operaciones matemáticas que se efectúan en ellos. En este libro todas las variables numéricas pueden contener un punto decimal y un signo que indica positivo o negativo; algunos lenguajes de programación proporcionan tipos numéricos especializados para estas opciones. En la declaración `myAnswer = myNumber * 2`, tanto `myAnswer` como `myNumber` son variables numéricas; es decir, su contenido previsto son valores numéricos, como 6 y 3, 14.8 y 7.4 o -18 y -9.
- Una **variable de cadena** puede contener texto, como las letras del alfabeto y otros caracteres especiales, como los signos de puntuación. Si un programa en funcionamiento contiene la declaración `lastName = "Lincoln"`, entonces `lastName` es una variable de cadena. Una

variable de cadena también puede contener dígitos ya sea con o sin otros caracteres. Por ejemplo, tanto “235 Main Street” como “86” son cadenas. Una cadena como “86” se almacena de manera diferente que el valor numérico 86 y usted no puede realizar aritmética con la cadena.

La **seguridad del tipo** es la característica de los lenguajes de programación que impide asignar valores de un tipo de datos incorrecto. Usted puede asignar datos a una variable sólo si son del tipo correcto. Si usted declara `taxRate` como una variable numérica e `inventoryItem` como una cadena, entonces las siguientes declaraciones son válidas:

```
taxRate = 2.5  
inventoryItem = "monitor"
```

Las siguientes no son válidas debido a que el tipo de datos que se asigna no corresponden al tipo de la variable:

```
taxRate = "2.5"  
inventoryItem = 2.5
```

No lo haga
Si `taxRate` es numérico e `inventoryItem` es una cadena, entonces estas asignaciones no son válidas.

Declaración de constantes nombradas

Además de las variables, casi todos los lenguajes de programación permiten la creación de constantes nombradas. Una **constante nombrada** es similar a una variable, excepto que se le puede asignar un valor sólo una vez. Se usa cuando se desea asignar un nombre útil para un valor que nunca cambiará durante la ejecución de un programa. El uso de constantes nombradas hace que sus programas sean más fáciles de entender al eliminar números mágicos. Un **número mágico** es una constante literal, como 0.06, cuyo propósito no es evidente de inmediato.

Por ejemplo, si un programa usa una tasa de impuesto sobre las ventas de 6%, quizás desee declarar una constante nombrada como sigue:

```
num SALES_TAX_RATE = 0.06
```

Ya que se ha declarado `SALES_TAX_RATE`, las siguientes declaraciones tienen significado idéntico:

```
taxAmount = price * 0.06  
taxAmount = price * SALES_TAX_RATE
```

La forma en que se declaran las constantes nombradas difiere entre los lenguajes de programación. Este libro sigue la convención de usar sólo letras mayúsculas en los identificadores de las constantes y subrayados para separar las palabras por legibilidad. Estas normas facilitan el reconocimiento de las constantes nombradas. En muchos lenguajes debe asignarse un valor a una constante cuando se declara, pero en otros es posible hacerlo después. Sin embargo, en ningún caso es posible cambiar el valor de una constante después de la primera asignación. Aquí se inicializan todas las constantes cuando se declaran.

Cuando usted declara una constante nombrada, el mantenimiento del programa se vuelve más fácil. Por ejemplo, si el valor del impuesto sobre las ventas cambia de 0.06 a 0.07 en el futuro, y usted ha declarado una constante llamada `SALES_TAX_RATE`, sólo necesita cambiar el valor asignado a la constante nombrada al principio del programa, luego retraducir éste

al lenguaje de máquina y todas las referencias a SALES_TAX_RATE se actualizan en forma automática. Si en su lugar usa la constante literal 0.06, tendría que buscar cada caso del valor y reemplazarlo con el nuevo. Además, si la literal 0.06 se usó en otros cálculos dentro del programa (por ejemplo, como tasa de descuento), tendría que seleccionar con cuidado cuáles casos del valor se alterarán y quizás cometió un error.



En ocasiones, el uso de constantes literales es apropiado en un programa, en especial si su significado es claro para la mayoría de los lectores. Por ejemplo, en un programa que calcula la mitad de un valor dividiéndolo entre dos, podría elegir la constante literal 2 en lugar de suceder en el tiempo y los costos de memoria adicionales de crear una constante nombrada HALF y asignarle 2. Los costos adicionales que resulten de agregar variables o instrucciones al programa se conocen como **carga adicional**.

DOS VERDADES Y UNA MENTIRA

Declaración y uso de variables y constantes

1. El tipo de datos de una variable describe la clase de valores que ésta puede contener y los tipos de operaciones que es posible efectuar con ellos.
2. Si name es una variable de cadena, entonces la declaración `set name = "Ed"` es válida.
3. El operando a la derecha de un operador de asignación debe ser un nombre que represente una dirección de memoria.

La afirmación falsa es la número 3. El operando a la izquierda de un operador de asignación debe ser un número que represente una dirección de memoria; el número de la ubicación donde se almacenará el resultado. Cuálquier operando a la derecha de un operador de asignación puede ser una dirección de memoria (una variable) o una constante.

Realización de operaciones aritméticas

La mayoría de los lenguajes de programación usan los siguientes operadores aritméticos estándar:

- + (signo de suma): adición
- (signo de resta): sustracción
- * (asterisco): multiplicación
- / (diagonal): división

Muchos lenguajes también soportan operadores adicionales que calculan el residuo después de la división, elevan un número a una potencia mayor, manipulan bits individuales almacenados dentro de un valor y efectúan otras operaciones.

Cada uno de los operadores aritméticos estándar es un operador binario; es decir, que requiere una expresión en ambos lados. Por ejemplo, la siguiente declaración suma dos puntuaciones de examen y asigna la suma a una variable llamada `totalScore`:

```
totalScore = test1 + test2
```

La siguiente suma 10 a `totalScore` y almacena el resultado en `totalScore`:

```
totalScore = totalScore + 10
```

En otras palabras, este ejemplo aumenta el valor de `totalScore`. Este último ejemplo se ve extraño en álgebra porque parecería que el valor de `totalScore` y el de `totalScore` más 10 son equivalentes. Debe recordar que el signo de igual es el operador de asignación y que la declaración en realidad toma el valor original de `totalScore`, sumándole 10 y asignando el resultado a la dirección de memoria a la izquierda del operador, que es `totalScore`.

En los lenguajes de programación usted puede combinar declaraciones aritméticas. Cuando lo hace, cada operador sigue las **reglas de precedencia** (también llamadas **orden de las operaciones**) que dictan el orden en que se realizan las operaciones en la misma declaración. Las reglas de precedencia para las declaraciones aritméticas básicas son las siguientes:

- Las expresiones entre paréntesis se evalúan primero. Si hay varios conjuntos de paréntesis, la expresión dentro del más interior se evalúa primero.
- La multiplicación y la división se evalúan a continuación, de izquierda a derecha.
- La suma y la resta se evalúan después, de izquierda a derecha.

El operador de asignación tiene poca precedencia. Por consiguiente, en una declaración como `d = e * f + g`, las operaciones a la derecha del operador de asignación siempre se efectúan antes de la asignación final a la variable en la izquierda.



Cuando usted aprenda un lenguaje de programación específico, conocerá todos los operadores que se usan en él. Muchos libros sobre el tema contienen una tabla que especifica la precedencia relativa de todos los operadores que se usan en el lenguaje.

Por ejemplo, considere las siguientes dos declaraciones aritméticas:

```
firstAnswer = 2 + 3 * 4  
secondAnswer = (2 + 3) * 4
```

Después de que se ejecutan estas declaraciones, el valor de `firstAnswer` es 14. De acuerdo con las reglas de precedencia, la multiplicación se efectúa antes que la suma, así que 3 se multiplica por 4 y resulta 12, y luego se suman 2 y 12, y se asigna 14 a `firstAnswer`. Sin embargo, el valor de `secondAnswer` es 20, porque el paréntesis obliga a que se efectúe primero la operación de suma que contiene. Se suman 2 y 3, lo que resulta 5, y luego 5 se multiplica por 4, lo que da 20.

El olvido de las reglas de precedencia aritmética o de añadir los paréntesis cuando se necesitan puede generar errores lógicos que son difíciles de encontrar en los programas. Por ejemplo, la siguiente declaración parecería promediar dos puntuaciones de examen:

```
average = score1 + score2 / 2
```

Sin embargo, no es así. Debido a que la división tiene una precedencia mayor que la suma, la declaración anterior indica que se obtiene la mitad de `score2`, se suma a `score1` y el resultado se almacena en `average`. La declaración correcta es:

```
average = (score1 + score2) / 2
```

Usted es libre de agregar paréntesis aun cuando no los necesite para forzar un orden diferente de operaciones; en ocasiones se usan sólo para hacer más claras sus intenciones. Por ejemplo, las siguientes declaraciones operan en forma idéntica:

```
totalPriceWithTax = price + price * TAX_RATE
totalPriceWithTax = price + (price * TAX_RATE)
```

En ambos casos, `price` se multiplica primero por `TAX_RATE`, luego este resultado se suma a `price` y lo que se obtiene al final se almacena en `totalPriceWithTax`. Debido a que la multiplicación ocurre antes de la suma en el lado derecho del operador de asignación, ambas declaraciones son iguales. Sin embargo, si usted piensa que la declaración con paréntesis hace más claras sus intenciones para alguien que lea su programa, entonces debe usarlos.

Todos los operadores aritméticos tienen **asociatividad de izquierda a derecha**. Esto significa que las operaciones con la misma precedencia tienen lugar de izquierda a derecha. Considere la siguiente declaración:

```
answer = a + b + c * d / e - f
```

La multiplicación y la división tienen una precedencia mayor que la suma o la resta, así que se llevan a cabo de izquierda a derecha como sigue:

`c` se multiplica por `d`, y el resultado se divide entre `e`, lo que da un resultado nuevo.

Por tanto, la declaración queda:

```
answer = a + b + (resultado temporal que ya se ha calculado) - f
```

Entonces se efectúan la suma y la resta de izquierda a derecha como sigue:

Se suman `a` y `b`, se suma el resultado temporal, y luego se resta `f`. El resultado final se asigna entonces a `answer`.

Otra forma de decirlo es que las siguientes dos declaraciones son equivalentes:

```
answer = a + b + c * d / e - f
answer = a + b + ((c * d) / e) - f
```

El cuadro 2-1 resume la precedencia y asociatividad de los cinco operadores que se usan con mayor frecuencia.

Símbolo del operador	Nombre del operador	Precedencia (comparada con otros operadores en este cuadro)	Asociatividad
=	Asignación	Más baja	Derecha a izquierda
+	Suma	Media	Izquierda a derecha
-	Resta	Media	Izquierda a derecha
*	Multiplicación	Más alta	Izquierda a derecha
/	División	Más alta	Izquierda a derecha

Cuadro 2-1 Precedencia y asociatividad de cinco operadores comunes

DOS VERDADES Y UNA MENTIRA

Realización de operaciones aritméticas

1. Los paréntesis tienen mayor precedencia que cualquiera de los operadores aritméticos comunes.
2. Las operaciones en declaraciones aritméticas ocurren de izquierda a derecha en el orden en que aparecen.
3. La siguiente suma 5 a una variable nombrada `points`:

```
points = points + 5
```

La afirmación falsa es la número 2. Las operaciones de igual precedencia se efectúan entre paréntesis se realizan primero, la multiplicación y la división se efectúan a continuación, y la suma y la resta se hacen al último. La declaración aritmética se efectúan de izquierda a derecha, pero las operaciones que se encuentran entre paréntesis se realizan primero, la multiplicación y la división se efectúan a continuación, y la suma y la resta se hacen al último.

Comprensión de las ventajas de la modularización

Los programadores rara vez escriben los programas como una larga serie de pasos. En cambio, dividen sus problemas de programación en unidades más pequeñas y abordan una tarea cohesiva a la vez. Estas unidades más pequeñas son **módulos**. Los programadores también se refieren a ellos como **subrutinas**, **procedimientos**, **funciones** o **métodos**; el nombre por lo general refleja el lenguaje de programación que se esté usando. Por ejemplo, los programadores en Visual Basic usan *procedimiento* (o *subprocedimiento*). En C y C++ los módulos se llaman *funciones*, mientras que en C#, Java y otros lenguajes orientados hacia los objetos es más probable que se conozcan como *método*. En COBOL, RPG y BASIC (lenguajes más antiguos) en general se denominan *subrutinas*.

Un programa principal ejecuta un módulo al llamarlo. **Llamar a un módulo** quiere decir que se usa su nombre para atraerlo, causando que se ejecute. Cuando las tareas del módulo están completas, el control regresa al punto desde el que se llamó en el programa principal. Cuando usted entra a un módulo, la acción es parecida a la de poner en pausa un reproductor de DVD: abandona su acción primaria (ver un video), se ocupa de alguna otra tarea (por ejemplo, hacer un emparedado) y luego regresa a la tarea principal exactamente donde la dejó.

La **modularización** es el proceso de descomponer un programa extenso en módulos; los científicos en computación también la llaman **descomposición funcional**. Nunca se requiere que modularice un programa extenso para que corra en una computadora, pero hay al menos tres razones para hacerlo:

- La modularización proporciona abstracción.
- Permite que muchos programadores trabajen en un problema.
- Hace posible reutilizar el trabajo con más facilidad.

La modularización proporciona abstracción

Una razón para que sea más fácil entender los programas modularizados es que permiten a un programador ver “todo el panorama”. La **abstracción** es el proceso de poner atención en las propiedades importantes mientras se ignoran los detalles no esenciales. La abstracción es ignorancia selectiva; la vida sería tediosa sin ella. Por ejemplo, usted puede crear una lista de cosas que hará hoy:

Lavar ropa
 Llamar a la tía Nan
 Empezar el ensayo semestral

Sin abstracción, la lista de quehaceres comenzaría así:

Levantar el cesto de ropa sucia
 Poner el cesto de ropa sucia en el automóvil
 Conducir hasta la lavandería
 Salir del automóvil con el cesto
 Entrar a la lavandería
 Bajar el cesto
 Encontrar monedas para la máquina lavadora
 ... etcétera.

Usted podría enumerar una docena de pasos más antes de terminar de lavar la ropa y pasar a la segunda labor en su lista original. Si tuviera que considerar todos los detalles pequeños de bajo nivel de cada tarea en su día es probable que nunca saliera de la cama por la mañana. Cuando usa una lista más abstracta de nivel superior hace su día más manejable. La abstracción hace que las tareas complejas se vean más sencillas.



Los artistas abstractos crean pinturas en las que sólo se ve el panorama general (color y forma) y se ignoran los detalles. La abstracción tiene un significado similar entre los programadores.

Del mismo modo, ocurre algún nivel de abstracción en cada programa de computadora. Hace 50 años un programador tenía que entender las instrucciones de circuitería de bajo nivel que la máquina usaba, pero ahora los lenguajes de programación de alto nivel más recientes permiten el uso de vocabulario en inglés en el que una declaración amplia corresponde a docenas de instrucciones. Sin importar cuál lenguaje de programación de alto nivel use, cuando usted despliega un mensaje en el monitor nunca se le pide que entienda cómo funciona éste para crear cada píxel en la pantalla. Usted escribe una instrucción como `output message` y el sistema operativo maneja los detalles de las operaciones de hardware para usted.

Los módulos brindan otra opción para lograr la abstracción. Por ejemplo, un programa de nómina puede llamar a un módulo que se ha nombrado `computeFederalWithholdingTax()`. Cuando usted lo llama desde su programa usa una declaración; el módulo en sí podría contener docenas de declaraciones. Usted puede escribir los detalles matemáticos del módulo después, alguien más puede hacerlo o puede adquirirlos de una fuente externa. Cuando usted planea su programa de nómina principal, su única preocupación es que tendrá que calcularse la retención de un impuesto federal; guarde los detalles para más tarde.

La modularización permite que varios programadores trabajen en un problema

Cuando usted disecciona cualquier tarea grande en módulos, tiene la capacidad de dividir con más facilidad la tarea entre varias personas. Rara vez un solo programador escribe un programa comercial que se pueda comprar. Considere cualquier procesador de palabras, hoja de cálculo o base de datos que haya usado. Cada programa tiene tantas opciones y responde a las selecciones del usuario en tantas formas posibles, que tomaría años que un solo programador escribiera todas las instrucciones. Los desarrolladores de software profesionales pueden escribir programas nuevos en semanas o meses, en lugar de años, si dividen los programas extensos en módulos y asignan cada uno a un programador individual o un equipo.

La modularización permite que se reutilice el trabajo

Si un módulo es útil y está bien escrito quizá usted desee usarlo más de una vez en ese programa o en otros. Por ejemplo, una rutina que verifica la validez de las fechas es útil en muchos programas escritos para un negocio. (Por ejemplo, un valor de mes es válido si no es menor que 1 o mayor que 12, un valor de día es válido si no es menor que 1 o mayor que 31 si el mes es 1, etc.). Si un archivo de personal computarizado contiene las fechas de nacimiento, de contratación, del último ascenso y de renuncia de cada empleado, el módulo de validación de fechas puede usarse cuatro veces con cada registro de empleado. Otros programas en una organización también pueden usar el módulo; podrían embarcar pedidos del cliente, planear fiestas de cumpleaños de los empleados o calcular cuándo deben hacerse los pagos de un préstamo. Si usted escribe las instrucciones de comprobación de las fechas de modo que estén entremezcladas con otras declaraciones en un programa, extraerlas y reutilizarlas será difícil. Por otra parte, si coloca las instrucciones en su propio módulo, será fácil usar y transportar la unidad a otras aplicaciones. La característica de los programas modulares que permite usar módulos individuales en una variedad de aplicaciones es la **reutilización**.

Es posible encontrar muchos ejemplos de reutilización en el mundo real. Cuando alguien construye una casa no inventa la plomería y los sistemas de calefacción; incorpora sistemas con diseños probados. Esto reduce con seguridad el tiempo y el esfuerzo que se requieren para construirla. Los sistemas de plomería y eléctricos que se elijan están en servicio en otras casas, así que han sido probados en diversas circunstancias, lo que aumenta su confiabilidad. La **confiabilidad** es la característica de los programas que asegura que un módulo funciona en forma correcta. El software confiable ahorra tiempo y dinero. Si usted crea los componentes funcionales de sus programas como módulos independientes y los prueba en sus programas actuales, gran parte del trabajo estará hecho cuando use los módulos en aplicaciones futuras.

DOS VERDADES Y UNA MENTIRA

Comprensión de las ventajas de la modularización

1. La modularización elimina la abstracción, una característica que hace que los programas sean confusos.
2. La modularización hace más fácil que muchos programadores trabajen en un problema.
3. La modularización permite reutilizar el trabajo con más facilidad.

que permite ver el panorama general.

La afirmación falsa es la número 1. La modularización permite la abstracción, lo

Modularización de un programa

La mayoría de los programas consiste en un **programa principal**, que contiene los pasos básicos, o la **lógica de línea principal** del programa. Entonces el programa principal entra en los módulos que proporcionan detalles más refinados.

Cuando se crea un módulo se incluye lo siguiente:

- Un encabezado: el **encabezado del módulo** incluye el identificador del mismo y posiblemente otra información de identificación necesaria.
- Un cuerpo: El **cuerpo del módulo** contiene todas las declaraciones en el mismo.
- Una declaración return: La **declaración return del módulo** marca el final de éste e identifica el punto en que el control regresa al programa o módulo que llamó al otro. En casi todos los lenguajes de programación, si no se incluye una declaración **return** al final de un módulo, la lógica todavía regresará. Sin embargo, este libro sigue la norma de incluir en forma explícita una declaración **return** con cada módulo.

Nombrar un módulo es algo similar a nombrar una variable. Las reglas para hacerlo son ligeramente diferentes en cada lenguaje de programación, pero en este texto los nombres de los módulos siguen las mismas reglas generales que se usan para los identificadores de variables:

- Los nombres de los módulos deben constar de una palabra y comenzar con una letra.
- Los nombres de los módulos deben tener algún significado.



Aunque no es un requisito en los lenguajes de programación, con frecuencia tiene sentido usar un verbo como nombre o parte del nombre de un módulo, debido a que los módulos realizan alguna acción. Los nombres típicos de los módulos comienzan con palabras de acción como `get`, `calculate` y `display`. Cuando usted programa en lenguajes visuales que usan componentes de pantalla como botones y cuadros de texto, los nombres de los módulos con frecuencia contienen verbos que representan las acciones del usuario, como `click` o `drag`.

Además, en este texto una serie de paréntesis sigue a los nombres de los módulos. Esto le ayudará a distinguir los nombres de módulo de los nombres de variable. Este estilo corresponde a la forma en que los módulos se nombran en muchos lenguajes de programación, como Java, C++ y C#.



Conforme aprenda más sobre módulos en lenguajes de programación específicos, encontrará que en ocasiones coloca los nombres de variable dentro de los paréntesis de los nombres de módulo. Cualquier variable que se encierre en los paréntesis contiene información que usted desea enviar al módulo. Por ahora, los paréntesis al final de los nombres de módulo estarán vacíos en este libro.

Cuando un programa principal desea usar un módulo, lo llama. Un módulo puede llamar a otro, y éste puede llamar a otro. Sólo la cantidad de memoria disponible en su computadora limita el número de llamadas encadenadas. En este libro, el símbolo de diagrama de flujo que se usa para llamar a un módulo es un rectángulo con una barra a lo largo de la parte superior. Dentro del rectángulo se coloca el nombre del módulo que se llama.



Algunos programadores usan un rectángulo con franjas a cada lado para representar un módulo en un diagrama de flujo, y en este libro se usa esta norma si un módulo es externo a un programa. Por ejemplo, los módulos preescritos incorporados que generan números aleatorios, calculan las funciones trigonométricas estándar y clasifican los valores que con frecuencia son externos a sus programas. Sin embargo, si el módulo se crea como parte del programa, en este libro se usa un rectángulo con una sola franja a lo largo de la parte superior.

En un diagrama de flujo se traza cada módulo de manera separada con sus propios símbolos centinela. El centinela del principio contiene el nombre del módulo. Este nombre debe ser idéntico al que se usa en el programa que llama. El centinela del final contiene `return`, lo cual indica que cuando el módulo termina, la progresión lógica de las declaraciones saldrá del mismo y regresará al programa que lo llamó. Del mismo modo, en un seudocódigo, empieza cada módulo con su nombre y termina con una declaración `return`; el nombre del módulo y las declaraciones `return` se alinean en forma vertical y todas las declaraciones del módulo tienen sangría entre ellas.

Por ejemplo, considere el programa de la figura 2-3, que no contiene ningún módulo. Acepta el nombre de un cliente y el saldo deudor como entradas y genera una factura. En la parte superior de la factura se despliega el nombre y la dirección de la compañía en tres líneas, seguidas por el nombre del cliente y el saldo deudor. Para desplegar el nombre y la dirección de la compañía, sólo puede incluir tres declaraciones `output` en la lógica de línea principal de un programa, como se muestra en la figura 2-3, o modularizar el programa creando tanto la lógica de línea principal y un módulo `displayAddressInfo()`, como se muestra en la figura 2-4.

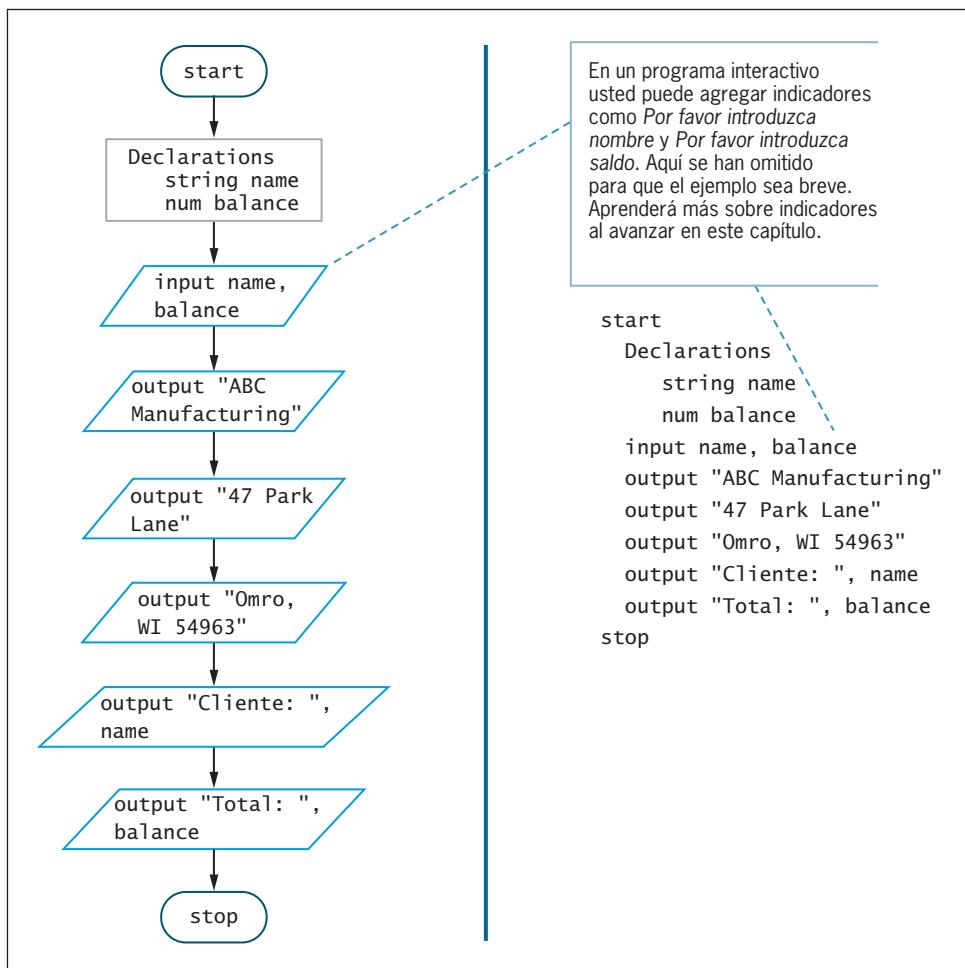


Figura 2-3 Programa que genera una factura usando sólo un programa principal

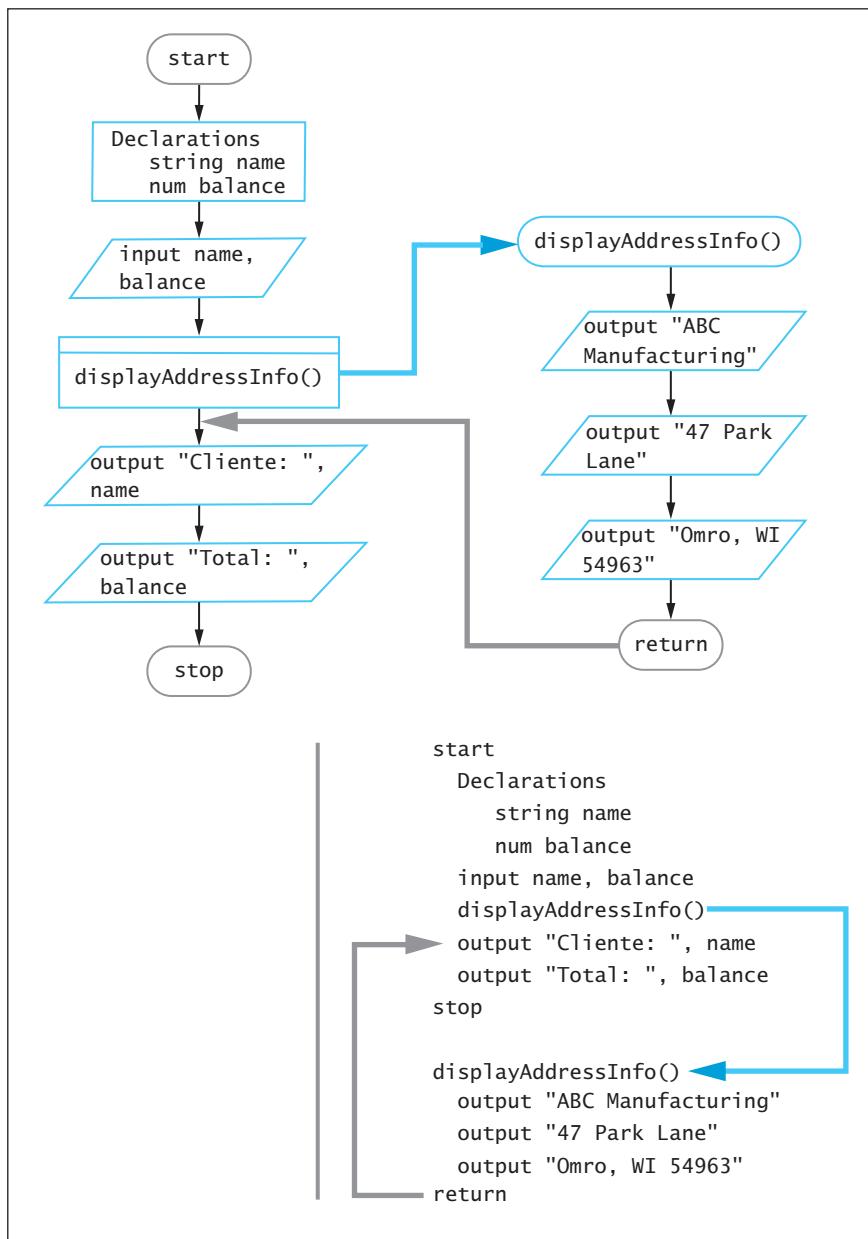


Figura 2-4 Programa que genera una factura usando un programa principal que llama al módulo `displayAddressInfo()`

Cuando se llama el módulo `displayAddressInfo()` en la figura 2-4, la lógica se transfiere del programa principal al módulo `displayAddressInfo()`, como se muestra con la flecha azul larga tanto en el diagrama de flujo como en el seudocódigo. Ahí, cada declaración del módulo se ejecuta una a la vez antes de que el control lógico sea transferido de vuelta al programa.

principal, donde continúa con la declaración que sigue a la llamada al módulo, como lo muestra la flecha gris larga. Los programadores dicen que las declaraciones que están contenidas en un módulo han sido **encapsuladas**.

Ninguno de los programas en las figuras 2-3 y 2-4 es superior al otro desde el punto de vista de la funcionalidad; ambos realizan exactamente las mismas tareas en el mismo orden. Sin embargo, usted quizás prefiera la versión modularizada del programa al menos por dos razones:

- Primera, el programa principal se mantiene breve y fácil de seguir porque contiene sólo una declaración para llamar al módulo, en lugar de tres declaraciones `output` separadas para realizar el trabajo de un módulo.
- Segunda, un módulo es fácilmente reutilizable. Después de que usted crea el módulo de información de dirección, puede usarlo en cualquier aplicación que necesite el nombre y dirección de la compañía. En otras palabras, usted hace el trabajo una vez y luego puede usar el módulo muchas veces.



Una desventaja potencial de crear módulos y moverse entre ellos es la carga adicional en que se incurre. La computadora sigue la pista a la dirección de memoria correcta a la que debe regresar después de ejecutar un módulo grabándola en una ubicación conocida como **pila**. Este proceso requiere una pequeña cantidad de tiempo y recursos de la computadora. En la mayoría de los casos, la ventaja de crear módulos supera con mucho la pequeña cantidad de carga adicional que se requiere.

La determinación de cuándo modularizar un programa no depende de una serie de reglas fijas; requiere experiencia y perspicacia. Los programadores siguen algunos lineamientos cuando deciden cuánto dividir en módulos o cuánto poner en cada uno. Algunas compañías tienen reglas arbitrarias, como “las instrucciones de un módulo nunca deberán ocupar más de una página” o “un módulo nunca debe tener más de 30 declaraciones” o “nunca tenga un módulo con una sola declaración”. En lugar de usar esas reglas arbitrarias, una mejor política consiste en colocar juntas las declaraciones que contribuyen a una tarea específica. Entre más contribuyan las declaraciones a la misma tarea, la **cohesión funcional** del módulo será mayor. Un módulo que verifique la validez del valor de una variable de fecha o uno que pida a un usuario un valor y lo acepte como entrada se consideran cohesivos. Un módulo que verifique la validez de la fecha, deduzca primas de seguro y calcule la retención de impuestos federales para un empleado sería menos cohesivo.

Declaración de variables y constantes dentro de los módulos

Usted puede colocar cualquier declaración dentro de los módulos, incluyendo declaraciones de entrada, de procesamiento y de salida; incluso declaraciones variables y constantes. Por ejemplo, quizás decida modificar el programa de facturación de la figura 2-4 de modo que se parezca al de la figura 2-5. En esta versión del programa, tres constantes nombradas que contienen las tres líneas de datos de la compañía se declaran dentro del módulo `displayAddressInfo()`. (Véase el sombreado.)

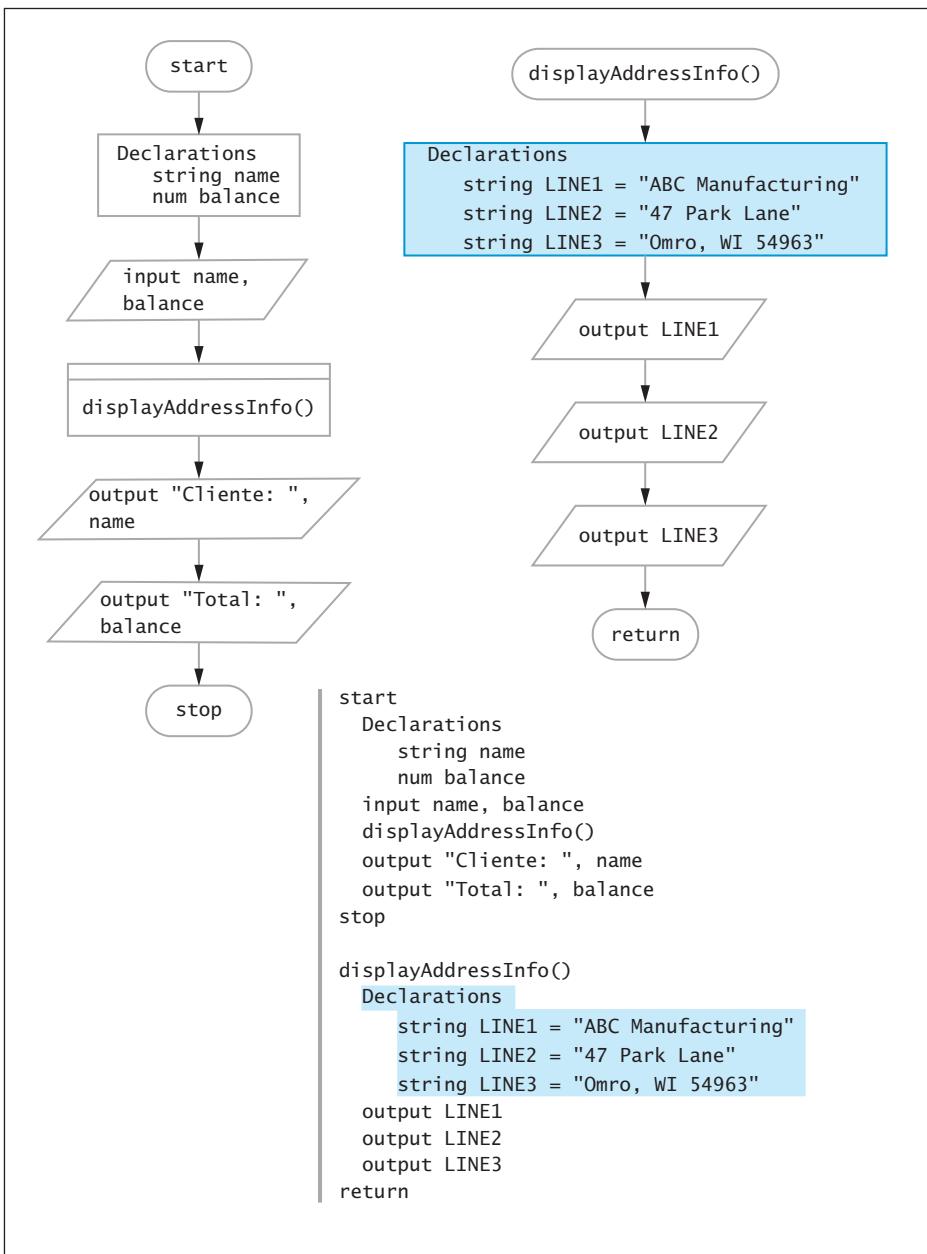


Figura 2-5 El programa de facturación con constantes declaradas dentro del módulo

Las variables y constantes que se declaran en un módulo son utilizables sólo dentro del mismo. Los programadores dicen que los elementos de datos son **visibles** sólo dentro del módulo en el que se declaran; esto significa que el programa sólo las reconoce ahí. También dicen que las variables y constantes que se declaran en un módulo están **en ámbito** sólo dentro de él. Además, mencionan que las variables y las constantes son **locales** para el módulo en el que se declaran. En otras palabras, cuando las cadenas LINE1, LINE2 y LINE3 se declaran en el módulo `displayAddressInfo()` en la figura 2-5, el programa principal no las reconoce y no puede usarlas.

Una de las motivaciones para crear módulos es que los que están separados se reutilizan con facilidad en diversos programas. Si varios programas de la organización usarán el módulo `displayAddressInfo()`, tiene sentido que las definiciones para sus variables y constantes deban venir con él. Esto hace que los módulos sean más **portátiles**; es decir, son unidades autónomas que se transportan sin problemas.

Además de las variables y constantes locales, usted puede crear variables y constantes **globales**. Éstas se conocen para el programa entero; se dice que se declaran en el **nivel de programa**. Esto significa que son visibles y utilizables en todos los módulos que llama el programa. Lo contrario es falso: las variables y constantes que se declaran dentro de un módulo no se utilizan en cualquier parte; sólo son visibles para ese módulo. (Por ejemplo, en la figura 2-5 las variables del programa principal `name` y `balance` son globales, aunque en este caso no se usan en los módulos.) En casi todo este libro se utilizarán sólo variables y constantes globales de modo que sea más fácil seguir los ejemplos y usted pueda concentrarse en la lógica principal.



Muchos programadores no aprueban el uso de variables y constantes globales. Aquí se usan de modo que usted comprenda con mayor facilidad la modularización antes de aprender las técnicas para enviar variables locales de un módulo a otro.

Comprensión de la configuración más común para la lógica de línea principal

En el capítulo 1 usted aprendió que un programa procedimental contiene pasos que se suceden en una secuencia. La lógica de línea principal de casi todos los programas procedimentales puede seguir una estructura general que consta de tres partes:

1. Las **tareas de administración** incluyen todos los pasos que usted debe efectuar al principio de un programa a fin de estar listo para el resto del mismo. Aquí se encuentran las declaraciones de variables y constantes, el despliegue de las instrucciones para los usuarios y de los encabezados de los informes, la apertura de todos los archivos que el programa requiera y la introducción de las primeras piezas de datos.



El ingreso de los primeros elementos de datos siempre es parte del módulo de administración. En el capítulo 3 usted aprenderá la teoría que hay detrás de esta práctica. En el capítulo 7 se trata el manejo de archivos, incluyendo lo que significa abrirlos y cerrarlos.

2. Las **tareas de ciclo detallado** hacen el trabajo central del programa. Cuando éste procesa muchos registros, las tareas de ciclo detallado se ejecutan de manera repetida para cada conjunto de datos de entrada hasta que ya no hay más. Por ejemplo, en un programa de nómina, el mismo conjunto de cálculos se ejecuta muchas veces hasta que se genera un cheque para cada empleado.
3. Las **tareas de fin de trabajo** son los pasos que usted sigue al final del programa para terminar la aplicación; puede llamarlas tareas de culminación o de limpieza. Incluye el despliegue de totales u otros mensajes finales y el cierre de todos los archivos abiertos.

La figura 2-6 muestra la relación de estas tres partes típicas de un programa. Note cómo las tareas `housekeeping()` y `endOfJob()` se ejecutan sólo una vez, pero las `detailLoop()` se repiten todo el tiempo mientras no se haya cumplido la condición `eof`. En el diagrama se usa una línea de flujo para mostrar cómo se repite el módulo `detailLoop()`; el seudocódigo usa las palabras `while` y `endwhile` para contener las declaraciones que se ejecutan en un ciclo. Aprenderá más sobre los términos `while` y `endwhile` en los capítulos siguientes; por ahora comprenda que son una forma de expresar acciones repetidas.

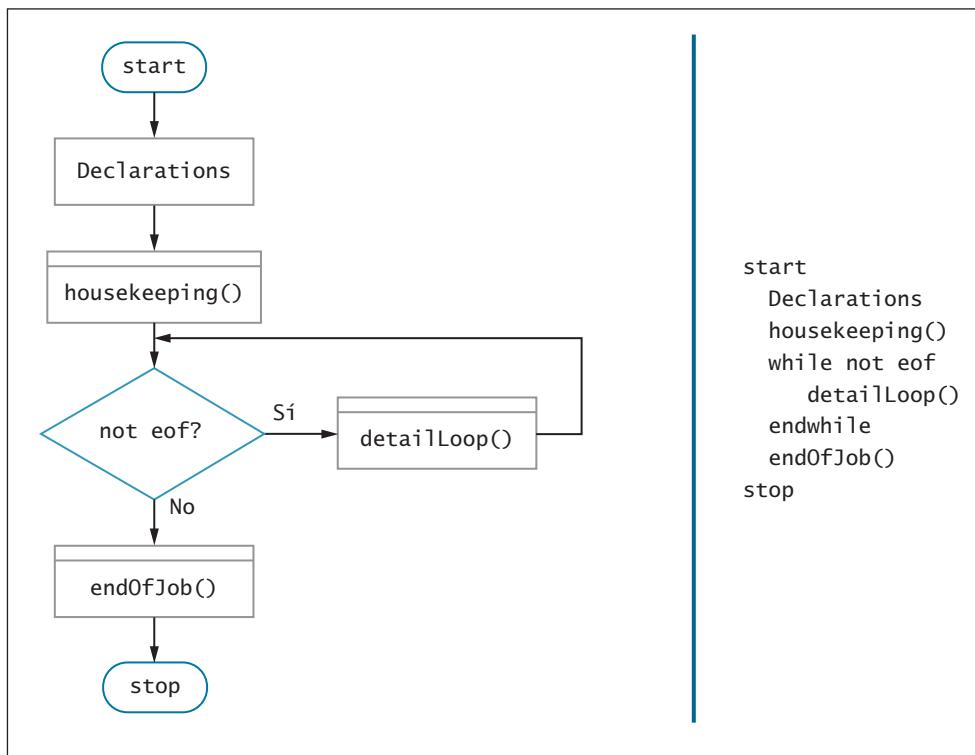


Figura 2-6 Diagrama de flujo y seudocódigo de lógica de línea principal para un programa procedural típico

Muchas tareas cotidianas siguen el formato de tres módulos que se acaba de describir. Por ejemplo, una fábrica de dulces abre por la mañana y las máquinas se encienden y se llenan con ingredientes. Estas tareas de administración ocurren sólo una vez al inicio del día. Luego, repetidamente durante la jornada se fabrican dulces; este proceso requiere muchos pasos y cada uno de ellos ocurre muchas veces. Éstos conforman el ciclo detallado. Luego, al final del día, las máquinas se limpian y apagan; son las tareas de fin de trabajo.

No todos los programas adoptan el formato de la lógica que se muestra en la figura 2-6, pero muchos lo hacen. Tenga presente esta configuración general mientras piensa cómo organizaría muchos de ellos. Por ejemplo, en la figura 2-7 se presenta la muestra de un informe de nómina para una compañía pequeña. Un usuario introduce los nombres de los empleados hasta que ya no hay más; en este punto ingresa *XXX*. En tanto el nombre no sea *XXX*, el usuario introduce el pago bruto semanal del empleado. Las deducciones se calculan como 25% fijo del pago bruto y se da salida a las estadísticas para cada empleado. El usuario introduce otro nombre, y mientras no sea *XXX* el proceso continúa. Examine la lógica en la figura 2-8 para identificar los componentes en las tareas administrativas, de ciclo detallado y de fin del trabajo. Aprenderá más sobre el programa de informe de nómina en los siguientes capítulos. Por ahora, concéntrese en todo el panorama de cómo funciona una aplicación típica.

Informe de nómina			
Nombre	Bruto	Deducciones	Neto
Andrews	1000.00	250.00	750.00
Brown	1400.00	350.00	1050.00
Carter	1275.00	318.75	956.25
Young	1100.00	275.00	825.00

***Fin del informe

Figura 2-7 Muestra de informe de nómina

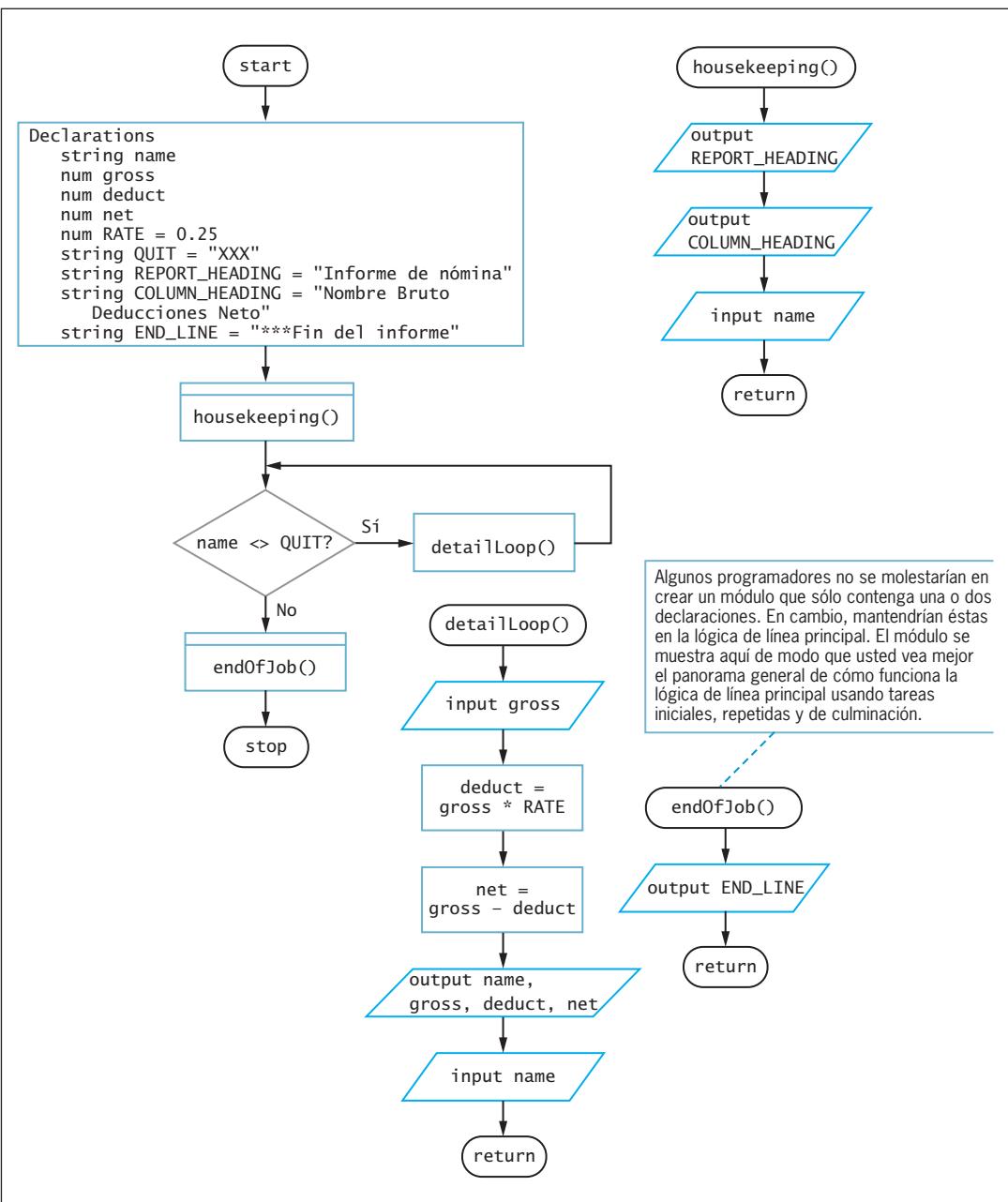


Figura 2-8 Lógica para el informe de nómina

DOS VERDADES Y UNA MENTIRA

Modularización de un programa

1. Un programa de llamada es el que llama al módulo cuando desea usarlo.
2. Siempre que un programa principal llama a un módulo la lógica se transfiere a éste; cuando el módulo termina, el programa finaliza.
3. Las tareas de administración incluyen todos los pasos que deben efectuarse una vez al principio del programa a fin de estar listo para el resto del mismo.

La afirmación falsa es la número 2. Cuando un módulo termina, el flujo lógico se **transfiere de vuelta** al programa principal que lo llamó y se reanuda donde se quedó.

Creación de gráficas de jerarquía

Quizá usted haya visto gráficas de jerarquía para las organizaciones, como la de la figura 2-9. La gráfica muestra quién reporta a quién, no cuándo o qué tan a menudo le reportan.

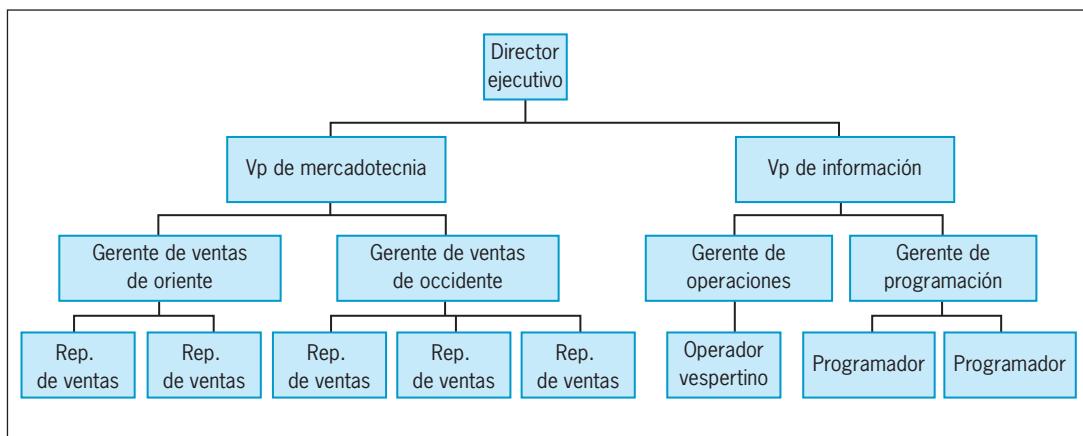


Figura 2-9 Una gráfica de jerarquía organizacional

Cuando un programa tiene varios módulos que llaman a otros, los programadores con frecuencia usan una **gráfica de jerarquía** del programa que funciona de manera parecida para mostrar el panorama general de la forma en que los módulos se relacionan entre sí. Una gráfica de jerarquía no le dice qué tareas se ejecutarán *dentro* de un módulo, *cuándo* se llaman los módulos, *cómo* se ejecuta un módulo o *por qué* se llaman (esta información está en el diagrama de flujo o en el pseudocódigo). Sólo le indica *cuáles* módulos existen dentro de un programa y *cuáles* llaman a otros. La gráfica de jerarquía en la figura 2-8 se ve como la

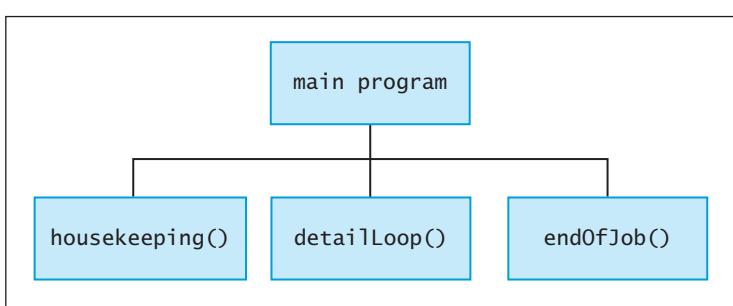


Figura 2-10 Gráfica de jerarquía del programa de informe de nómina de la figura 2-8

La figura 2-11 muestra el ejemplo de una gráfica de jerarquía para el programa de facturación de una compañía de pedidos por correo. La gráfica es para un programa más complicado, pero como la del informe de nómina en la figura 2-10, proporciona los nombres de los módulos y una visión general de las tareas que se realizarán, sin especificar detalles.

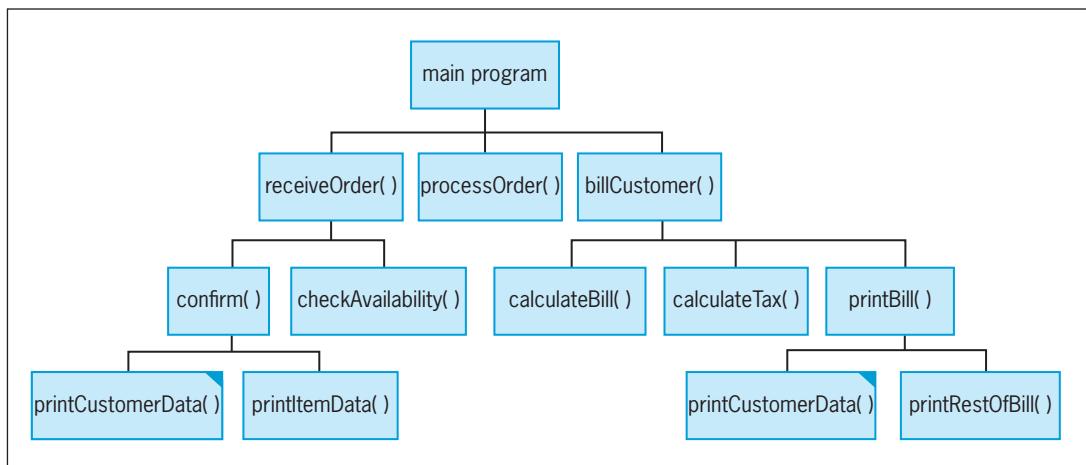


Figura 2-11 Gráfica de jerarquía del programa de facturación

Debido a que los módulos del programa son reutilizables, un módulo específico puede llamarse desde varias ubicaciones dentro de un programa. Por ejemplo, en la gráfica de jerarquía del programa de facturación en la figura 2-11 usted puede ver que el módulo `printCustomerData()` se usa dos veces. Por convención, se oscurece una esquina de cada cuadro que representa un módulo que se usa más de una vez. Esta acción alerta a los lectores acerca de que cualquier cambio en ese módulo tendría consecuencias en múltiples ubicaciones.

Una gráfica de jerarquía es una herramienta de planeación para desarrollar la relación general de los módulos del programa antes de escribirlos y una de documentación para ayudar a otros a ver cómo se relacionan los módulos después de que se escribe un programa. Por ejemplo, si cambia una ley fiscal, podría pedirse a un programador que reescriba el módulo `calculateTax()` en el diagrama del programa de facturación de la figura 2-11. Cuando el programador cambia el módulo `calculateTax()`, la gráfica de jerarquía muestra otros módulos dependientes que resultarían afectados. Una gráfica de jerarquía es útil para obtener el panorama general en un programa complejo.



Las gráficas de jerarquía se usan en la programación procedural, pero con frecuencia se utilizan otros tipos de diagramas en los ambientes orientados hacia los objetos.

DOS VERDADES Y UNA MENTIRA

Creación de gráficas de jerarquía

1. Usted puede usar una gráfica de jerarquía para ilustrar las relaciones de los módulos.
2. Una gráfica de jerarquía le dice cuáles tareas se realizarán dentro de un módulo.
3. Una gráfica de jerarquía sólo le indica cuáles módulos llaman a otros módulos.

Los módulos entran si.

Las afirmaciones falsas es la número 2. Una gráfica de jerarquía no le indica nada sobre las tareas que se realizan dentro de un módulo; sólo describe cómo se relacionan

Características de un buen diseño de programa

Conforme sus programas se vuelven más extensos y complicados, aumenta la necesidad de la planeación y el diseño adecuados. Piense en una aplicación que use, como un procesador de palabras o una hoja de cálculo. El número y la variedad de opciones de usuario son asombrosos. No sólo sería imposible que un solo programador escribiera una aplicación así sino que además, si faltaran una planeación y un diseño minuciosos, los componentes nunca trabajarían juntos en forma apropiada. De manera ideal, cada módulo del programa que diseña necesita funcionar bien como componente autónomo y como elemento de sistemas más grandes. Igual que una casa con una plomería deficiente o un automóvil con frenos que fallan están fatalmente defectuosos, una aplicación basada en la computadora puede ser funcional sólo si cada componente está bien diseñado. Recorrer la lógica de su programa en papel (lo que se conoce como prueba de escritorio, como aprendió en el capítulo 1) es un paso importante para diseñar programas superiores. Asimismo, usted puede implementar varias características de diseño mientras crea programas que son más fáciles de escribir y mantener. Para crear programas adecuados deberá hacer lo siguiente:

- Usar comentarios al programa donde sea apropiado.
- Elegir los identificadores de manera minuciosa.
- Esforzarse por diseñar declaraciones precisas en sus programas y módulos.
- Escribir indicadores y entradas con eco claros.
- Conservar los hábitos adecuados conforme mejora sus habilidades de programación.

Uso de comentarios del programa

Cuando usted escribe programas quizá desea insertar comentarios en ellos. Los **comentarios del programa** son explicaciones escritas que no forman parte de la lógica del mismo pero que sirven como documentación para los lectores. En otras palabras, son declaraciones que no se ejecutan y que ayudan a los lectores a entender las declaraciones de programación. Los lectores podrían ser usuarios que le ayudan a probar el programa u otros programadores que tal vez tengan que modificarlo en el futuro. Incluso usted, como autor del mismo, apreciará los comentarios cuando haga modificaciones y olvide por qué construyó de cierta manera una declaración.

La sintaxis que se usa para crear los comentarios del programa difiere entre los lenguajes de programación. En este libro los comentarios en seudocódigo comienzan con dos diagonales frontales. Por ejemplo, la figura 2-12 contiene comentarios que explican los orígenes y propósitos de las variables en un programa de bienes raíces.



Los comentarios del programa son un tipo de **documentación interna**. Este término los distingue de los documentos de soporte fuera del programa, que se conocen como **documentación externa**. El apéndice D explica otros tipos de documentación.

```
Declarations
num sqFeet
    //sqFeet es una estimación proporcionada por el vendedor de la propiedad
num pricePerFoot
    // pricePerFoot lo determinan las condiciones actuales del mercado
num lotPremium
    // lotPremium depende de servicios tales como si el lote es costero
```

Figura 2-12 Seudocódigo que declara variables e incluye comentarios

En un diagrama de flujo usted puede usar un símbolo de anotación que contenga información que amplía lo que se almacena dentro de otro símbolo del diagrama de flujo. Un **símbolo de anotación** casi siempre se representa con un cuadro de tres lados unido con una línea punteada al paso al que hace referencia. Los símbolos de anotación se usan para incluir comentarios o en ocasiones declaraciones que por ser demasiado largas no caben en un símbolo del diagrama de flujo. Por ejemplo, la figura 2-13 muestra cómo un programador usaría algunos símbolos de anotación en un diagrama de flujo para un programa de nómina.

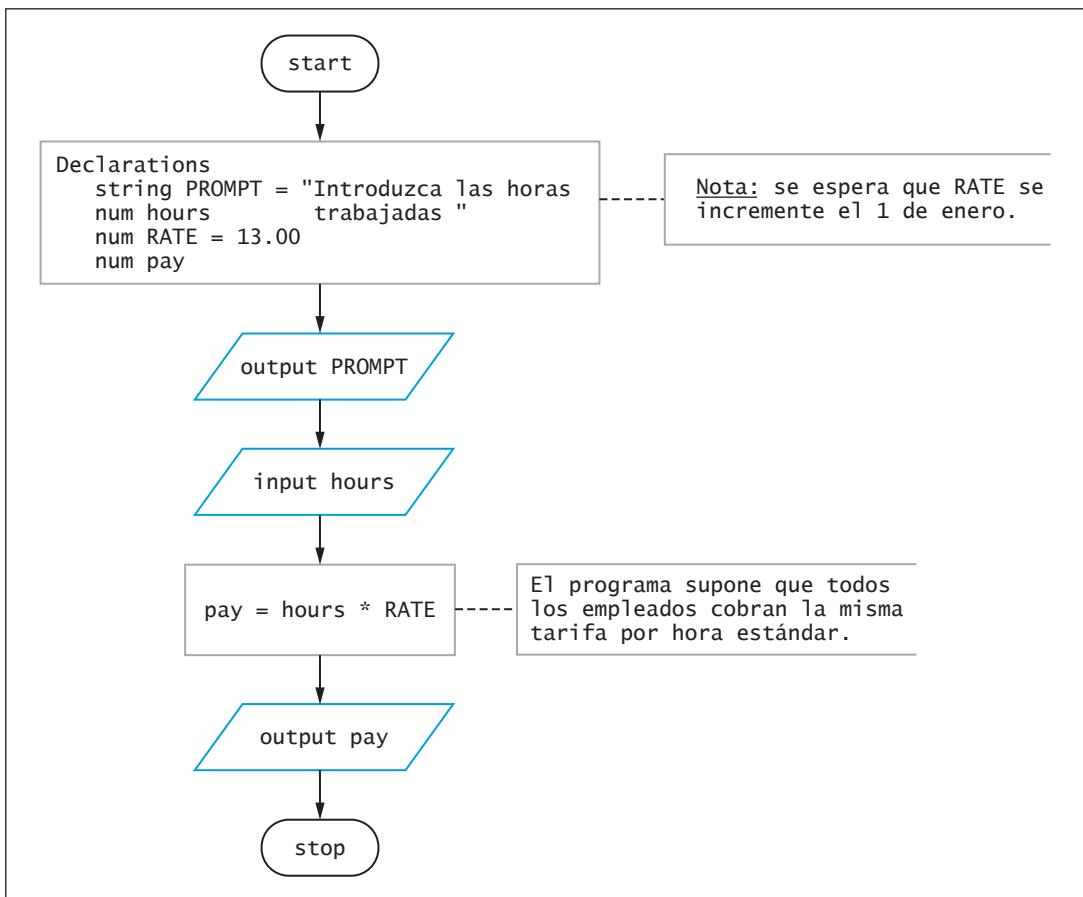


Figura 2-13 Diagrama de flujo que incluye símbolos de anotación



Es probable que usted use comentarios en sus programas codificados con mayor frecuencia de lo que los usa en el seudocódigo o los diagramas de flujo. Por una parte, los diagramas de flujo y el seudocódigo son más parecidos al inglés que el código en algunos lenguajes, así que sus declaraciones serían menos confusas. Además, sus comentarios permanecerán en el programa como parte de la documentación del mismo, pero es probable que sus herramientas de planeación se desechen una vez que el programa entra en producción.

No es necesario incluir comentarios para crear un programa que sea funcional, pero pueden ayudarle a recordar el propósito de las variables o explicar cálculos complicados. A algunos estudiantes no les gusta incluir comentarios porque teclearlos toma tiempo y no son parte del programa “real”. Pero es probable que los programas que usted escriba en el futuro los requieran. Cuando reciba su primer trabajo de programación y modifique un programa que alguien más haya escrito apreciará los comentarios bien colocados que expliquen las secciones complicadas del código.



Una desventaja de los comentarios es que deben actualizarse conforme el programa se modifica. Los comentarios obsoletos pueden aportar información engañosa sobre el estado del programa.

66

Elección de identificadores

La selección de identificadores adecuados es un paso que con frecuencia se pasa por alto en el diseño del programa. Cuando usted escribe programas, elige identificadores para las variables, las constantes y los módulos. Antes en este capítulo usted aprendió las reglas para nombrar variables y módulos: cada uno debe ser una sola palabra sin espacios y debe comenzar con una letra. Estas sencillas reglas brindan gran libertad de acción para nombrar los elementos del programa, pero no todos los identificadores son igual de buenos. La elección de los que sí lo son simplifica su labor de programación y facilita que otras personas entiendan su trabajo.

Algunos lineamientos generales son los siguientes:

- Aunque no es necesario en todos los lenguajes de programación, por lo general tiene sentido dar a una variable o constante un nombre que sea un sustantivo (o una combinación de un adjetivo y un sustantivo) debido a que representa una cosa. Del mismo modo, tiene sentido dar a un módulo un identificador que sea un verbo, o un verbo y un sustantivo combinados, debido a que un módulo emprende una acción.
- Use nombres significativos. Crear un elemento de datos que se ha nombrado `someData` o un módulo llamado `firstModule()` hace que un programa sea confuso. No sólo otras personas encontrarán difícil la lectura de los programas que usted haya escrito, sino que además usted podría olvidar el propósito de haber incluido estos identificadores. Todos los programadores usan nombres breves no descriptivos como `x` o `temp` en un programa rápido; sin embargo, en la mayoría de los casos, los nombres de los datos y el módulo deben ser significativos. Los programadores se refieren a los programas que contienen nombres significativos como **autodocumentados**. Esto significa que, aun sin información adicional, el código se explica por sí solo a los lectores.
- Use nombres pronunciables. Un nombre de variable como `pzf` no es pronunciable ni significativo; uno que parezca significativo cuando usted lo escribe podría no serlo cuando alguien más lo lea; por ejemplo, `preparead()` quizás signifique "Prepare ad" (Preparar anuncio) para usted, pero "Prep a read" (Preparar una lectura) para otras personas. Observe sus nombres en forma crítica para asegurarse de que es posible pronunciarlos. Las abreviaturas estándar no tienen que ser pronunciables. Por ejemplo, la mayoría de las personas de negocios interpretaría `isr` como impuesto sobre la renta.
- No olvide que no todos los programadores comparten su cultura. Una abreviatura cuyo significado parezca obvio para usted podría ser confusa para alguna persona en una parte distinta del mundo, o incluso de su país. Por ejemplo, usted podría nombrar `roi` a una variable a fin de que contenga un valor para el *rendimiento sobre la inversión*, pero alguien que hable francés interpretaría el significado como *rey*.
- Sea juicioso cuando use abreviaturas. Usted puede ahorrar algunos golpes de tecla cuando cree un módulo llamado `getStat()` pero, ¿el propósito del módulo es encontrar el estado en el que se localiza una ciudad, introducir algunas estadísticas o determinar el estado de algunas variables? Del mismo modo, ¿una variable que se ha nombrado `fn` pretende contener un primer nombre, un número de archivo o algo más? Las abreviaturas también

pueden confundir a las personas que se encuentran en diferentes líneas de trabajo: AKA podría sugerir una hermandad femenina (Alpha Kappa Alpha) para un administrador universitario, un registro (American Kennel Association) para un criador de perros o un alias (also known as [también conocido como]) para un detective de policía.



Para ahorrar tiempo en la mecanografía cuando desarrolle un programa, usted puede usar un nombre breve como `efn`. Una vez que el programa opera correctamente, puede usar las herramientas Buscar y Reemplazar de un editor de textos para cambiar su nombre codificado por otro más significativo como `employeeFirstName`.



Muchos IDE soportan una característica para completar declaraciones en forma automática que ahorra tiempo de mecanografía. Después de la primera vez que usted usa un nombre como `employeeFirstName`, sólo necesita teclear las primeras letras antes de que el editor del compilador le muestre una lista de nombres disponibles entre los que puede elegir. La lista se conforma con todos los nombres que ha usado y que empiecen con los mismos caracteres.

- En general, evite los dígitos. Los ceros se confunden con la letra *O*, y la letra *l* minúscula con el número 1. Por supuesto, use su juicio: `budgetFor2014` probablemente no se malinterpretará.
- Use el sistema que su lenguaje permita para separar las palabras en los nombres de variables largos que contengan muchas. Por ejemplo, si el lenguaje de programación que utiliza acepta guiones o subrayados, entonces use un nombre de módulo como `initialize-data()` o `initialize_data()`, lo cual es más fácil de leer que `initializedata()`. Otra opción es usar la notación de camelCase para crear un identificador como `initializeData()`. Si utiliza un lenguaje que es sensible a las mayúsculas y minúsculas, es legal pero confuso usar nombres de variables que sólo difieran en las mayúsculas y minúsculas. Por ejemplo, si un solo programa contiene `empName`, `EmpName` y `Empname`, de seguro habrá confusión.
- Considere incluir una forma del verbo *to be*, como *is* o *are*, en los nombres para las variables que se espera que contengan un estado. Por ejemplo, use `isFinished` como una variable de cadena que contiene una *Y* o *N* para indicar si un archivo está agotado. Hay más probabilidades de que el nombre más breve, `finished`, se confunda con un módulo que se ejecuta cuando acaba un programa. (Muchos lenguajes soportan un tipo de datos booleano, que se asigna a las variables que se espera que contengan sólo verdadero o falso. Es apropiado usar una forma de *to be* en los identificadores para las variables booleanas.)
- Muchos programadores siguen la convención de nombrar constantes usando sólo letras mayúsculas, insertando subrayados entre palabras para mejor legibilidad. En este capítulo vio ejemplos como `SALES_TAX_RATE`.
- Las organizaciones a veces imponen diferentes reglas para que los programadores las sigan cuando nombran los componentes del programa. Es su responsabilidad averiguar las convenciones que se utilizan en su organización y apegarse a ellas.



Los programadores en ocasiones crean un **diccionario de datos**, que es una lista de todos los nombres de variables que se han usado en un programa, junto con su tipo, tamaño y descripción. Cuando se crea este diccionario se vuelve parte de la documentación del programa.

Cuando usted comience a escribir programas, quizás la determinación de cuáles variables de datos, constantes y módulos necesita y cómo nombrarlos le parezca abrumador. Sin embargo, el proceso de diseño es crucial. Cuando reciba su primera asignación de programación profesional, el diseño bien podría haberse completado ya. Lo más probable es que su primera

asignación será escribir o modificar un pequeño módulo que forma parte de una aplicación mucho más grande. Entre más se apeguen los programadores originales a los lineamientos para nombrar, será mejor el diseño original y más fácil su labor de modificación.

Diseño de declaraciones precisas

68

Además de usar comentarios del programa y seleccionar buenos identificadores, usted puede aplicar las siguientes tácticas para contribuir a la precisión de las declaraciones dentro de sus programas:

- Evite cortes de línea confusos.
- Use variables temporales para aclarar declaraciones largas.

Evite cortes de línea confusos

Algunos lenguajes de programación antiguos requieren que las declaraciones se coloquen en columnas específicas. La mayor parte de los lenguajes modernos son de estilo libre; usted puede ordenar sus líneas de código en cualquier forma que considere adecuada. Como en la vida real, con la libertad viene la responsabilidad; cuando tiene flexibilidad para ordenar sus líneas de código debe asegurarse de que su significado sea claro. Con el código de estilo libre se permite a los programadores colocar dos o tres declaraciones en una línea o, a la inversa, extender una sola declaración a lo largo de múltiples líneas. Ambos casos hacen que sea más difícil leer los programas. Todos los ejemplos de seudocódigo en este libro usan un espaciado y cortes de línea claros y apropiados.

Use variables temporales para clarificar las declaraciones largas

Cuando necesite varias operaciones matemáticas para determinar un resultado considere la utilización de una serie de variables temporales para contener los resultados intermedios. Una **variable temporal** (o una **variable de trabajo**) no se usa para la entrada o salida, sino que es sólo una variable funcional que se emplea durante la ejecución de un programa. Por ejemplo, la figura 2-14 muestra dos formas de calcular un valor para una variable `salespersonCommission` de bienes raíces. Cada módulo obtiene el mismo resultado: la comisión de la persona se basa en los pies cuadrados multiplicados por el precio por pie cuadrado, más cualquier prima por un lote con características especiales, como uno arbolado o costero. Sin embargo, el segundo ejemplo usa dos variables temporales: `basePropertyPrice` y `totalSalePrice`. Cuando el cálculo se divide en pasos individuales menos complicados es más fácil ver cómo se calcula el precio total. En cálculos que tienen aún más pasos, la realización de la aritmética en etapas se volverá cada vez más útil.

```
// Usar una sola declaración para calcular la comisión  
salespersonCommission = (sqFeet * pricePerFoot + lotPremium) * commissionRate  
  
// Usar múltiples declaraciones para calcular la comisión  
basePropertyPrice = sqFeet * pricePerFoot  
totalSalePrice = basePropertyPrice + lotPremium  
salespersonCommission = totalSalePrice * commissionRate
```

Figura 2-14 Dos formas de lograr el mismo resultado `salespersonCommission`



Los programadores quizá mencionen que el uso de variables temporales, como el segundo ejemplo en la figura 2-14, es *barato*. Cuando se ejecuta una declaración aritmética larga, aun si no se nombran explícitamente las variables temporales, el compilador del lenguaje de programación las crea tras bambalinas (aunque sin nombres descriptivos), así que declararlas usted mismo no cuesta mucho en términos del tiempo de ejecución del programa.

Escritura de indicadores claros y entradas con eco

Cuando la entrada para el programa debe recuperarse de un usuario, usted casi siempre deseará proporcionar un indicador para este último. Un **indicador** es un mensaje que se despliega en el monitor para pedir al usuario una respuesta y quizá explicar cómo ésta se formateará. Los indicadores se usan tanto en los programas de línea de comandos como en los GUI interactivos.

Por ejemplo, suponga que un programa pide a un usuario que introduzca un número de catálogo para un artículo que está ordenando. El siguiente indicador no es muy útil:

Por favor introduzca un número.

El siguiente indicador es más adecuado:

Por favor introduzca un número de pedido por catálogo de cinco dígitos.

El siguiente indicador es aún más apropiado:

El número de pedido por catálogo de cinco dígitos aparece a la derecha de la imagen del artículo en el catálogo. Por favor introduzcalo ahora.

Cuando la entrada para el programa viene de un archivo almacenado y no de un usuario, no se necesitan indicadores. Sin embargo, cuando un programa espera una respuesta del usuario los indicadores son valiosos. Por ejemplo, la figura 2-15 muestra el diagrama de flujo y el seudocódigo para el inicio del programa generador de facturas que se mostró antes en este capítulo. Si la entrada proviene de un archivo de datos no se requerirán indicadores y la lógica se vería como la que se muestra en la figura 2-15.

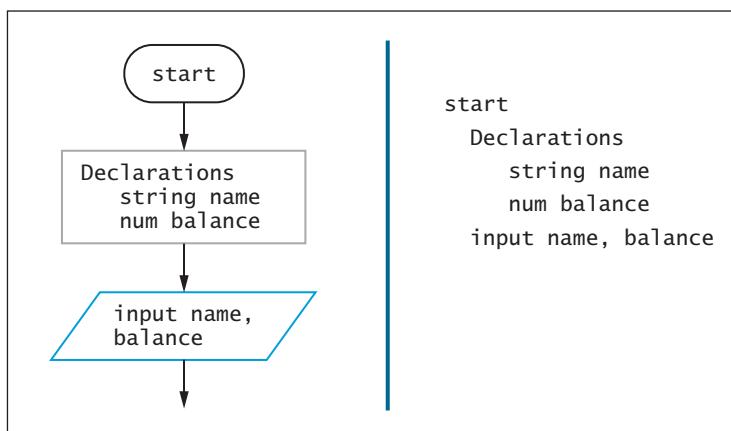


Figura 2-15 Inicio de un programa que acepta un nombre y saldo como entrada

Sin embargo, si la entrada provino de un usuario sería útil incluir indicadores. Usted podría suministrar un solo indicador como *Por favor introduzca un nombre de cliente y saldo deudor*, pero insertar más solicitudes en un indicador por lo general hace menos probable que el usuario recuerde que debe introducir todas las partes o que tiene que hacerlo en el orden correcto. Casi siempre es mejor incluir un indicador independiente para cada elemento que se introducirá. La figura 2-16 muestra un ejemplo.

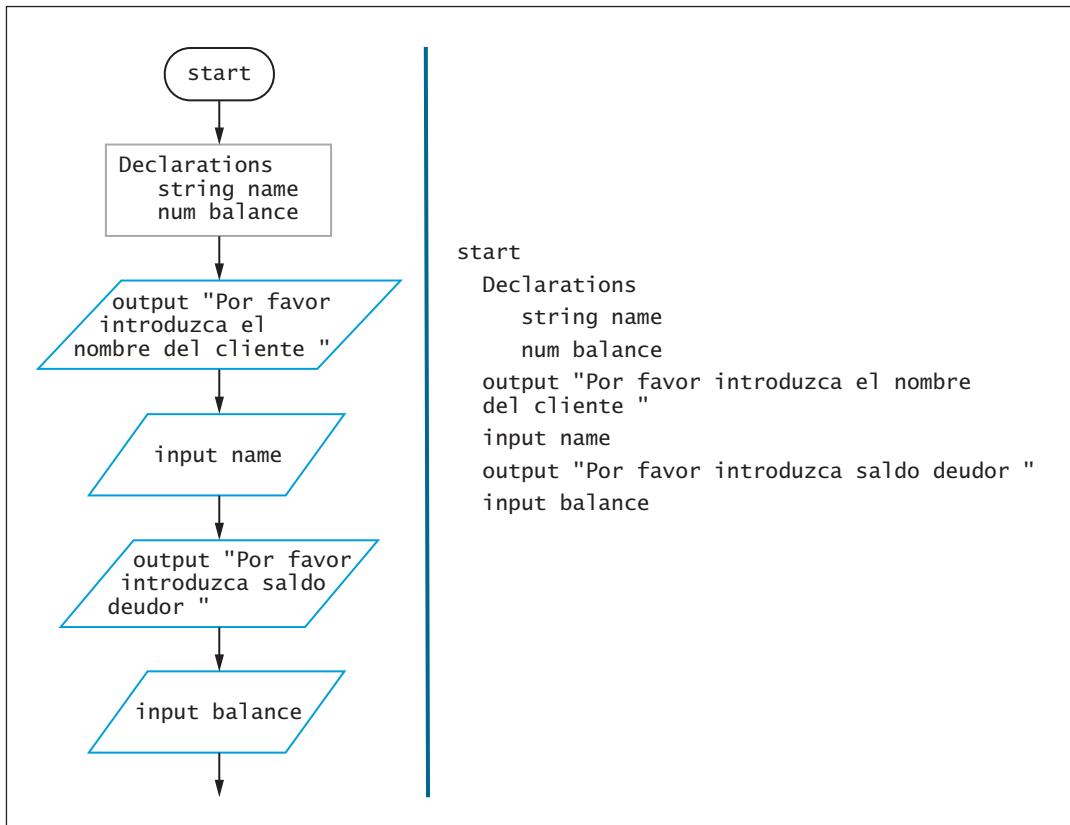


Figura 2-16 Inicio de un programa que acepta un nombre y saldo como entrada y usa un indicador independiente para cada elemento

También es útil para los usuarios que usted haga eco a su entrada. La **entrada con eco** es la acción de repetir la entrada al usuario, ya sea en un indicador subsiguiente o en la salida. Por ejemplo, la figura 2-17 muestra cómo el segundo indicador en la figura 2-16 puede mejorarse haciendo eco a la primera pieza de datos de entrada del usuario en el segundo indicador. Cuando un usuario corre el programa que se inicia en la figura 2-17 e introduce *Green* en el nombre del cliente, el segundo indicador no será *Por favor introduzca el saldo deudor* sino *Por favor introduzca el saldo deudor de Green*. Digamos, si un empleado estuviera a punto de introducir el saldo del cliente equivocado, la mención de *Green* sería suficiente para alertarlo de un error potencial.

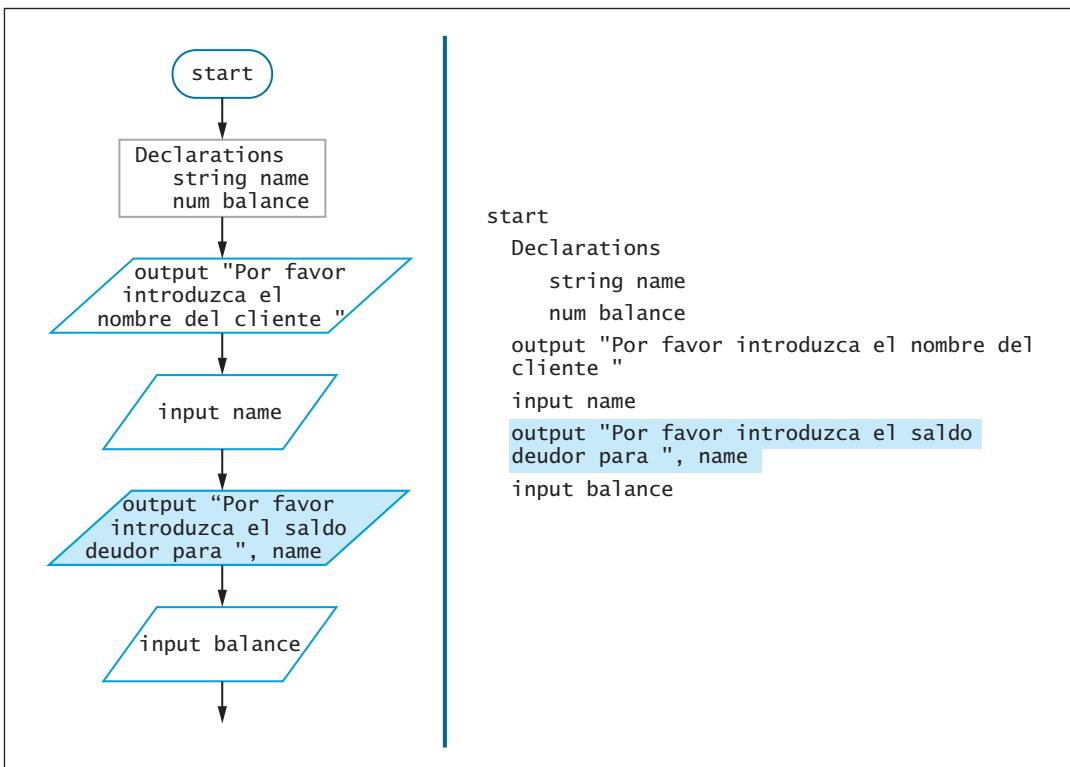


Figura 2-17 Inicio de un programa que acepta el nombre de un cliente y lo usa en el segundo indicador



Note el espacio que hay antes de las comillas en el indicador que pide al usuario un saldo deudor. El espacio aparecerá entre *para* y el apellido.

Mantener buenos hábitos de programación

Cuando usted aprende un lenguaje de programación y comienza a escribir las líneas de código de un programa es fácil que olvide los principios que ha aprendido en este texto. Tener algún conocimiento de programación y un teclado en sus manos puede animarle a escribir dichas líneas antes de pensarlas con cuidado; pero será mejor si antes planea el código de cada programa. Mantener el hábito de trazar primero los diagramas de flujo o escribir el seudocódigo, como ha aprendido aquí, hará que sus proyectos de programación futuros marchen sin complicaciones. Si hace su prueba de escritorio en papel para conocer la lógica antes de codificar las declaraciones en un lenguaje de programación, sus programas se ejecutarán en forma correcta más rápido. Si piensa con cuidado en los nombres de las variables y los módulos que elija y diseña declaraciones que sean fáciles de leer y usar, será más sencillo desarrollar y mantener sus programas.

DOS VERDADES Y UNA MENTIRA

Características de un diseño de programa apropiado

1. Un comentario del programa es un mensaje que se despliega en un monitor para pedir al usuario una respuesta y quizá explica cómo deberá formatearse la respuesta.
2. Por lo general tiene sentido dar a cada variable un nombre que contenga un sustantivo y a cada módulo uno que contenga un verbo.
3. La entrada con eco puede ayudar al usuario a confirmar que los elementos de datos se introdujeron en forma correcta.

La afirmación falsa es la número 1. Un comentario del programa es una explicación escrita que no forma parte de la lógica del mismo pero que sirve como documentación para quienes lo lean. Un indicador es un mensaje que se despliega en el monitor para pedir al usuario una respuesta y quizás explique cómo se formateará.

Resumen del capítulo

- Los programas contienen datos en tres formas diferentes: literales (o constantes literales), variables y constantes nombradas; cada uno de estos tipos puede ser numérico o de cadena. Las variables son ubicaciones de memoria nombradas cuyo contenido puede variar. Una declaración de variable incluye un tipo de datos y un identificador, y de manera opcional también una inicialización. Cada lenguaje de programación tiene su propio conjunto de reglas para nombrar variables; sin embargo, todos los nombres deben escribirse como una palabra sin espacios y tener un significado apropiado. Una constante nombrada es similar a una variable, excepto que sólo es posible asignarle un valor una vez.
- La mayoría de los lenguajes de programación usan $+$, $-$, $*$ y $/$ como los cuatro operadores aritméticos estándar. Cada operador sigue reglas de precedencia que dictan el orden en que se realizan las operaciones en la misma declaración; la multiplicación y la división siempre tienen precedencia sobre la suma y la resta. Estas reglas de precedencia pueden anularse usando paréntesis.
- Los programadores dividen los problemas en unidades cohesivas más pequeñas llamadas módulos, subrutinas, procedimientos, funciones o métodos. Para ejecutar un módulo, se le llama desde otro programa o módulo. Cualquier programa puede contener un número ilimitado de ellos y cada uno puede llamarse un número ilimitado de veces. La modularización proporciona abstracción, permite que varios programadores trabajen en un problema y facilita la reutilización de su trabajo.

- Cuando usted crea un módulo le asigna un encabezado, un cuerpo y una declaración `return`. Un programa o módulo llama al nombre de un módulo para ejecutarlo. Usted puede colocar cualquier declaración dentro de los módulos, incluyendo las que son locales para ellos. Las variables y constantes globales son aquellas que se conocen para el programa entero. La lógica de línea principal de casi todos los programas procedimentales puede seguir una estructura general que consiste en tres partes distintas: tareas de administración, de ciclo detallado y de fin de trabajo.
- Una gráfica de jerarquía ilustra los módulos y sus relaciones; indica cuáles existen dentro de un programa y cuáles llaman a otros.
- Conforme los programas se vuelven más extensos y complicados aumenta la necesidad de una planeación y un diseño adecuados. Usted debe usar comentarios del programa donde se requieran. Elija identificadores de manera sensata, esfuérzese por diseñar declaraciones precisas en sus programas y módulos, escriba indicadores y entradas con eco claras y conserve los buenos hábitos de programación conforme desarrolle sus habilidades de programación.

Términos clave

El término **numérico** describe los datos que consisten en números.

La palabra **cadena** se refiere a los datos que no son numéricos.

Un **entero** es un número entero.

Un número de **punto flotante** es un número con lugares decimales.

Los **números reales** son números de punto flotante.

Una **constante numérica** (o **constante numérica literal**) es un valor numérico específico.

Una **constante de cadena** (o **constante de cadena literal**) es un grupo específico de caracteres encerrados entre comillas.

Los **valores alfanuméricos** pueden contener caracteres alfabéticos, números y signos de puntuación.

Una **constante literal** es un valor numérico o de cadena literal.

Una **declaración** es un enunciado que proporciona un tipo de datos y un identificador para una variable.

Un **identificador** es el nombre de un componente del programa.

El **tipo de datos** de un elemento de datos es una clasificación que describe qué valores pueden asignarse, cómo se almacena el elemento y qué tipos de operaciones es posible realizar con el elemento.

Inicializar una variable es la asignación de su primer valor, con frecuencia al mismo tiempo que se crea la variable.

Basura es el valor desconocido almacenado en una variable no asignada.

Las **palabras clave** comprenden el conjunto limitado de palabras que se reservan en un lenguaje.

La **notación de camelio** es una convención para nombrar variables en la que la letra inicial es minúscula, los nombres de variable con múltiples palabras se escriben juntos y cada nueva palabra dentro del nombre de la variable comienza con una letra mayúscula.

La **caja de Pascal** es una convención para nombrar variables en la que la letra inicial es mayúscula, los nombres con múltiples palabras se escriben juntos y cada nueva palabra del nombre comienza con letra mayúscula.

La **notación húngara** es una convención para nombrar variables en la que el tipo de datos de las mismas u otra información se almacenan como parte de su nombre.

Una **declaración de asignación** asigna un valor desde la derecha de un operador de asignación hacia la variable o constante a la izquierda de dicho operador.

El **operador de asignación** es el signo de igual; se usa para asignar un valor a la variable o constante a su izquierda.

Un **operador binario** es aquel que requiere dos operandos, uno de cada lado.

La **asociatividad derecha** y la **asociatividad de derecha a izquierda** describen operadores que evalúan primero la expresión de la derecha.

Un **lvalue** es el identificador de la dirección de memoria a la izquierda de un operador de asignación.

Una **variable numérica** es aquella que puede contener dígitos, hacer que se realicen operaciones matemáticas en ella y por lo general contener un punto decimal y un signo que indique positivo o negativo.

Una **variable de cadena** puede contener texto que incluya letras, dígitos y caracteres especiales como signos de puntuación.

Seguridad de tipo es la característica de los lenguajes de programación que previene la asignación de valores de un tipo de datos incorrecto.

Una **constante nombrada** es similar a una variable, con excepción de que su valor no puede cambiar después de la primera asignación.

Un **número mágico** es una constante literal cuyo propósito no es evidente de inmediato.

El término **carga adicional** describe los recursos extra que una tarea requiere.

Las **reglas de precedencia** dictan el orden en que se realizan las operaciones en la misma declaración.

El **orden de las operaciones** describe las reglas de precedencia.

La **asociatividad de izquierda a derecha** describe operadores que evalúan primero la expresión de la izquierda.

Los **módulos** son unidades pequeñas que pueden usarse juntas para hacer un programa. Los programadores también se refieren a ellos como **subrutinas, procedimientos, funciones o métodos**.

Llamar a un módulo es usar el nombre del mismo para atraerlo, causando que se ejecute.

La **modularización** es el proceso de dividir un programa en módulos.

La **descomposición funcional** es la reducción de un programa grande en módulos más manejables.

La **abstracción** es el proceso de poner atención a las propiedades importantes mientras se ignoran los detalles no esenciales.

La **reutilización** es la característica de los programas modulares que permite que los módulos individuales se usen en diversas aplicaciones.

La **confiabilidad** es la característica de los programas modulares que asegura que un módulo se ha probado y funciona en forma correcta.

Un **programa principal** corre de principio a fin y llama a otros módulos.

La **lógica de línea principal** es la que aparece en el módulo principal de un programa; llama a otros módulos.

El **encabezado del módulo** incluye el identificador del mismo y quizá otra información de identificación necesaria.

El **cuerpo del módulo** contiene todas las declaraciones en este último.

La **declaración return del módulo** marca el final del mismo e identifica el punto en que el control regresa al programa o módulo que lo llamó.

El **encapsulamiento** es la acción de contener las instrucciones de una tarea en un módulo.

Una **pila** es una ubicación de memoria en la que la computadora sigue el rastro a la dirección de memoria correcta a la que deberá regresar después de ejecutar un módulo.

La **cohesión funcional** de un módulo es una medida del grado en que todas las declaraciones de éste contribuyen a la misma tarea.

La palabra **visible** describe el estado de los elementos de datos cuando un módulo puede reconocerlos.

El término **en ámbito** designa el estado de los datos que son visibles.

La palabra **local** se refiere a las variables que se declaran en el módulo que las usa.

Un módulo **portátil** es aquel que puede reutilizarse con más facilidad en múltiples programas.

El término **global** describe las variables que un programa entero conoce.

Las variables globales se declaran en el **nivel de programa**.

Las **tareas de administración** incluyen los pasos que usted debe efectuar al inicio de un programa a fin de estar listo para el resto del mismo.

Las **tareas de ciclo detallado** incluyen los pasos que se repiten para cada conjunto de datos de entrada.

Las **tareas de fin de trabajo** son los pasos que usted sigue al final del programa para terminar la aplicación.

Una **gráfica de jerarquía** es un diagrama que ilustra las relaciones recíprocas entre los módulos.

Los **comentarios del programa** son explicaciones escritas que no forman parte de la lógica del programa pero que sirven como documentación para quienes lo lean.

La **documentación interna** es la que está dentro de un programa codificado.

La **documentación externa** es la que está fuera de un programa codificado.

Un **símbolo de anotación** contiene información que amplía lo que aparece en otro símbolo del diagrama de flujo; con mayor frecuencia se representa con un cuadro abierto con tres lados que está conectado al paso al que hace referencia con una línea punteada.

Los programas con **autodocumentación** son aquellos que contienen datos significativos y nombres de módulos que describen su propósito.

Un **diccionario de datos** es una lista de los nombres de todas las variables que se usan en un programa, junto con su tipo, tamaño y descripción.

Una **variable temporal** (o **variable de trabajo**) es una variable activa que se usa para contener resultados intermedios durante la ejecución de un programa.

Un **indicador** es un mensaje que se despliega en un monitor para pedir al usuario una respuesta y quizás explicar cómo ésta debe formatearse.

La **entrada con eco** es la acción de repetir la entrada a un usuario ya sea en un indicador siguiente o en una salida.

Preguntas de repaso

1. ¿Qué proporciona una declaración a una variable?
 - a) un nombre
 - b) un tipo de datos
 - c) los dos anteriores
 - d) ninguno de los anteriores
2. El tipo de datos de una variable describe todo lo siguiente *excepto* _____.
 - a) qué valores puede contener ésta
 - b) cómo se almacena en la memoria
 - c) qué operaciones pueden realizarse con ella
 - d) el ámbito de la misma
3. El valor almacenado en una variable no inicializada es _____.
 - a) basura
 - b) nulo
 - c) abono
 - d) su identificador
4. El valor 3 es una _____.
 - a) variable numérica
 - b) constante numérica
 - c) variable de cadena
 - d) constante de cadena
5. El operador de asignación _____.
 - a) es un operador binario
 - b) tiene asociatividad de izquierda a derecha

- c) con más frecuencia se representa con dos puntos
d) dos de los anteriores

6. ¿Cuál de los siguientes términos es verdadero respecto a la precedencia aritmética?

 - La multiplicación tiene una precedencia mayor que la división.
 - Los operadores con menor precedencia siempre tienen asociatividad de izquierda a derecha.
 - La división tiene mayor precedencia que la resta.
 - Todas las anteriores.

7. ¿Cuál de los siguientes es un término que se usa como sinónimo para *módulo* en algunos lenguajes de programación?

 - método
 - procedimiento
 - ambos
 - ninguno de éstos

8. ¿Cuál de las siguientes es una razón para usar la modularización?

 - Que evita la abstracción.
 - Que reduce la carga adicional.
 - Que permite reutilizar el trabajo con mayor facilidad.
 - Que elimina la necesidad de sintaxis.

9. ¿Cuál es el nombre para el proceso de poner atención a las propiedades importantes mientras se ignoran los detalles no esenciales?

 - abstracción
 - extracción
 - extinción
 - modularización

10. Todos los módulos tienen todo lo siguiente *excepto* _____.

 - un encabezado
 - variables locales
 - un cuerpo
 - una declaración `return`

11. Los programadores dicen que un módulo puede _____ a otro, lo que significa que el primero causa que el segundo se ejecute.

 - declarar
 - definir
 - promulgar
 - llamar

12. Entre más contribuyen las declaraciones de un módulo a una misma labor, es mayor _____ del módulo.

 - la estructura
 - la modularidad
 - la cohesión funcional
 - el tamaño

Ejercicios

1. Explique por qué cada uno de los siguientes puede ser o no un nombre de variable adecuado para usted.
 - a) d
 - b) dsctamt
 - c) discountAmount
 - d) discount Amount
 - e) discount
 - f) discountAmountForEachNewCustomer
 - g) discountYear2013
 - h) 2013Discountyear
2. Si `productCost` y `productPrice` son variables numéricas y `productName` es una variable de cadena, ¿cuáles de las siguientes declaraciones son asignaciones válidas? Si una declaración no lo es, explique por qué no.
 - a) `productCost = 100`
 - b) `productPrice = productCost`
 - c) `productPrice = productName`
 - d) `productPrice = "24.95"`
 - e) `15.67 = productCost`
 - f) `productCost = $1,345.52`
 - g) `productCost = productPrice - 10`
 - h) `productName = "tapete para ratón"`
 - i) `productCost + 20 = productPrice`
 - j) `productName = 3-inch nails`
 - k) `productName = 43`
 - l) `productName = "44"`
 - m) `"99" = productName`
 - n) `productName = brush`
 - o) `battery = productName`
 - p) `productPrice = productPrice`
 - q) `productName = productCost`

3. Suponga que `income = 8` y `expense = 6`. ¿Cuál es el valor de cada una de las siguientes expresiones?
 - a) `income + expense * 2`
 - b) `income + 4 - expense / 2`
 - c) `(income + expense) * 2`
 - d) `income - 3 * 2 + expense`
 - e) `4 * ((income - expense) + 2) + 10`
4. Trace una gráfica de jerarquía típica para un programa que genera una factura mensual para un cliente de telefonía celular. Trate de pensar al menos en 10 módulos separados que se incluirían. Por ejemplo, uno de ellos podría calcular el cargo por minutos diurnos que se han usado.
5. a) Trace la gráfica de jerarquía y luego planee la lógica para un programa que el gerente de ventas de The Henry User Car Dealership necesita. El programa determinará la ganancia sobre cualquier automóvil vendido; la entrada incluye el precio de venta y el de compra real para un vehículo. La salida es la ganancia, que es el precio de venta menos el precio de compra. Use tres módulos. El programa principal declara variables globales y llama a módulos de administración, detallado y de fin de trabajo; el módulo de administración pide y acepta un precio de venta; el detallado pide y acepta el precio de compra, calcula la ganancia y despliega el resultado, y el de fin de trabajo despliega el mensaje *Gracias por usar este programa*.
b) Revise el programa que determina la ganancia de modo que corra en forma continua para cualquier cantidad de automóviles. El ciclo detallado se ejecuta en forma continua mientras el precio de venta no sea 0; además de calcular la ganancia, pide al usuario el siguiente precio de venta y lo obtiene. El módulo de fin de trabajo se ejecuta después de que se introduce 0 para el precio de venta.
6. a) Trace la gráfica de jerarquía y luego planee la lógica para un programa que calcule el índice de masa corporal (IMC) de una persona. El IMC es una medida estadística que compara el peso y la estatura. El programa usa tres módulos; el primero pide al usuario su estatura en pulgadas y la acepta; el segundo módulo acepta el peso del usuario en libras y convierte su estatura en metros y el peso en kilogramos. Luego, calcula el IMC como el peso en kilogramos por la estatura en metros al cuadrado y despliega los resultados. Hay 2.54 centímetros en una pulgada, 100 centímetros en un metro, 453.59 gramos en una libra y 1,000 gramos en un kilogramo. Use constantes nombradas siempre que las considere apropiadas. El último módulo despliega el mensaje *Fin del trabajo*.
b) Revise el programa que determina el IMC para que se ejecute en forma continua hasta que el usuario introduzca 0 para la estatura en pulgadas.
7. Trace la gráfica de jerarquía y diseñe la lógica para un programa que calcule los cargos por servicio de Hazel's Housecleaning. El programa contiene módulos de administración, de ciclo detallado y de fin de trabajo. El programa principal declara cualquier variable y constante globales necesarias y llama a los otros módulos. El módulo de

administración despliega un indicador y acepta el apellido de un cliente. Mientras el usuario no introduzca ZZZZ para el nombre, el ciclo detallado acepta el número de baños y el de otras habitaciones que se limpiarán. El cargo por servicio se calcula como \$40 más \$15 por cada baño y \$10 por cada una de las otras habitaciones. El ciclo detallado también despliega el cargo por servicio y luego pide al usuario el nombre del siguiente cliente. El módulo fin de trabajo, que se ejecuta después de que el usuario introduce el valor centinela para el nombre, despliega un mensaje que indica que el programa está completo.

8. Trace la gráfica de jerarquía y diseñe la lógica para un programa que calcule el costo proyectado de un viaje en automóvil. Suponga que el vehículo del usuario viaja a 8 kilómetros por litro de gasolina. Diseñe un programa que pida al usuario el número de kilómetros recorridos y el costo actual por litro. El programa calcula y despliega el costo del viaje al igual que el costo si los precios de la gasolina aumentan 10%; acepta datos en forma continua hasta que se introduce 0 para el número de kilómetros. Use los módulos apropiados, incluyendo uno que despliegue *Fin del programa* cuando éste termine.
9. a) Trace la gráfica de jerarquía y diseñe la lógica para un programa que necesita el gerente del equipo de softball del condado Stengel, quien desea calcular los porcentajes de *slugging* para sus jugadores. Un porcentaje de *slugging* es el total de bases alcanzadas dividido entre el número de turnos al bate del jugador. Diseñe un programa que pida al usuario el número de camiseta de un jugador, el de bases alcanzadas y el de turnos al bate y luego despliegue todos los datos, incluyendo el promedio de *slugging* calculado. El programa acepta jugadores en forma continua hasta que se introduce 0 para el número de camiseta. Use módulos apropiados, incluyendo uno que despliegue *Fin de trabajo* después de que se introduce el centinela para el número de camiseta.
 b) Modifique el programa de porcentaje de *slugging* para calcular también el porcentaje de veces que se embasa un jugador. Un porcentaje de veces que se embasa se calcula sumando los hits y bases por bolas de un jugador y luego se divide entre la suma de turnos al bate, bases por bolas y elevados de sacrificio. Pida al usuario con un indicador todos los datos adicionales necesarios y despliegue todos los datos para cada jugador.
 c) Modifique el programa de softball de modo que también calcule un promedio de producción bruta (*GPA, gross production average*) para cada jugador. Un *GPA* se calcula multiplicando el porcentaje de veces que cada uno se embasa por 1.8, luego sumando el porcentaje de *slugging* del jugador y luego dividiendo entre cuatro.



Encuentre los errores

10. Sus archivos descargables para el capítulo 2 incluyen DEBUG02-01.txt, DEBUG02-02.txt y DEBUG02-03.txt. Cada archivo empieza con algunos comentarios que describen el problema. Los comentarios son líneas que comienzan con dos diagonales (//). Después de los comentarios, cada archivo contiene seudocódigo que tiene uno o más errores que usted debe encontrar y corregir. (NOTA: Estos archivos se encuentran disponibles sólo para la versión original en inglés.)

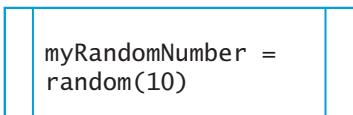


Zona de juegos

11. Para que los juegos mantengan su interés, casi siempre incluyen algún comportamiento aleatorio impredecible. Por ejemplo, uno en el que dispara a los asteroides pierde algo de su diversión si éstos siguen la misma trayectoria predecible cada vez que juega. Por consiguiente, generar valores aleatorios es un componente clave en la creación de juegos de computadora más interesantes. En muchos lenguajes de programación se ha incorporado un módulo que usted puede usar para generar números aleatorios. La sintaxis varía en cada lenguaje, pero en general es algo como esto:

```
myRandomNumber = random(10)
```

En esta declaración, `myRandomNumber` es una variable numérica que usted ha declarado y la expresión `random(10)` significa “llamar a un método que genere y devuelva un número aleatorio entre 1 y 10”. Por convención, en un diagrama de flujo colocaría una declaración como ésta en un símbolo de procesamiento con dos franjas verticales en los bordes, como se muestra a continuación.



Cree un diagrama de flujo o un seudocódigo que muestre la lógica para un programa que genere un número aleatorio, luego pida al usuario que piense en un número entre 1 y 10. Después despliegue el número generado aleatoriamente de modo que el usuario vea si su hipótesis fue correcta. (En capítulos futuros, mejorará este juego de modo que el usuario pueda introducir una suposición y el programa determine si estuvo en lo correcto.)



Para discusión

12. En la web se publican muchas guías de estilo de programación que sugieren buenos identificadores, explican las reglas de sangrado estándar e identifican cuestiones de estilo en lenguajes de programación específicos. Encuentre guías de estilo para al menos dos lenguajes (por ejemplo, C++, Java, Visual Basic o C#) y enumere cualquier diferencia que note.
13. ¿Qué ventajas hay en requerir que las variables tengan un tipo de datos?
14. Como se menciona en este capítulo, algunos lenguajes de programación requieren que a las constantes nombradas se les asigne un valor cuando se declaran; otros permiten que el valor de una constante se asigne más adelante en un programa. ¿Cuál requisito piensa que es mejor? ¿Por qué?
15. ¿Preferiría escribir un programa extenso por sí solo o trabajar en un equipo en el que cada programador elabora uno o más módulos? ¿Por qué?
16. La programación extrema es un sistema para desarrollar software con rapidez. Uno de sus principios es que todo el código de producción sea escrito por dos programadores sentados frente a una máquina. ¿Esto es buena idea? ¿Trabajar así es atractivo para usted como programador? ¿Por qué sí o por qué no?

CAPÍTULO 3

Comprender la estructura

En este capítulo usted aprenderá sobre:

- ◎ Las desventajas del código espagueti no estructurado
- ◎ Las tres estructuras básicas: secuencia, selección y ciclo
- ◎ Usar una entrada anticipada para estructurar un programa
- ◎ La necesidad de estructura
- ◎ Reconocer la estructura
- ◎ Estructurar y modularizar lógica no estructurada

Las desventajas del código espagueti no estructurado

Las aplicaciones de negocios profesionales por lo general son mucho más complejas que los ejemplos que ha visto hasta ahora en los capítulos 1 y 2. Imagine la cantidad de instrucciones en los programas que guían el vuelo de un avión o auditán una devolución del impuesto sobre la renta. Incluso el que genera el cheque de su salario en el trabajo contiene muchas, muchas instrucciones. Diseñar la lógica para un programa así puede ser una tarea que requiere tiempo. Cuando usted agrega varios miles de instrucciones a un programa, incluyendo varios cientos de decisiones, es fácil crear un desorden. El nombre común con que se designa a las declaraciones que se encuentran enmarañadas desde el punto de vista lógico es **código espagueti**, porque es difícil seguir la lógica, tal como ocurre con un espagueti en un plato de pasta. No sólo el código espagueti es confuso, los programas que lo contienen son propensos al error, es difícil reutilizarlos y usarlos como bloques para aplicaciones más grandes. Los **programas no estructurados** usan el código espagueti; es decir, no siguen las reglas de la lógica estructurada que aprenderá en este capítulo. Los **programas estructurados** *siguen* esas reglas y eliminan los problemas causados por el código espagueti.

Por ejemplo, suponga que empieza un trabajo como bañador de perros y recibe las instrucciones que se muestran en la figura 3-1. Este diagrama de flujo es un ejemplo del código espagueti no estructurado. Un programa de computadora que esté estructurado de manera similar podría “funcionar”; es decir, produciría resultados correctos, pero su lectura, su mantenimiento y el seguimiento de su lógica serían difíciles.

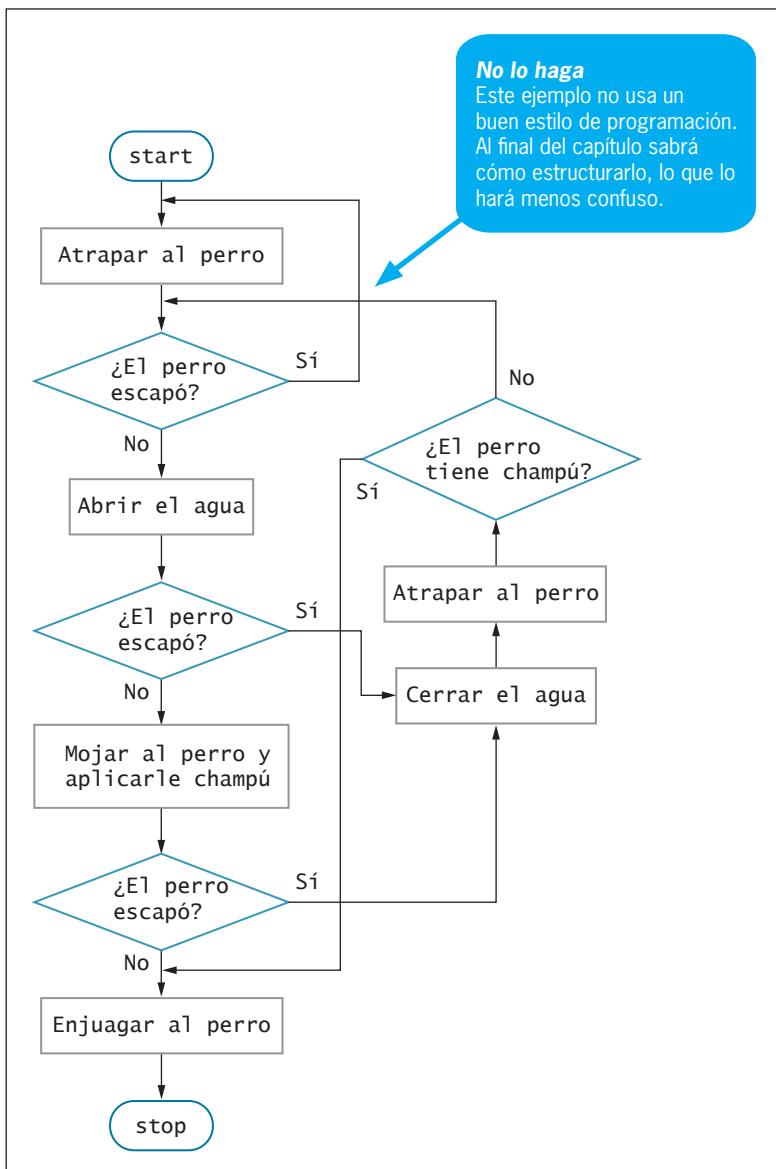


Figura 3-1 Lógica de código espagueti para bañar a un perro

Podría seguir la lógica del procedimiento para bañar perros de la figura 3-1 por dos razones:

- Es probable que ya sepa cómo bañar a un perro.
- El diagrama de flujo contiene un número limitado de pasos.

Sin embargo, imagine que no estuviera familiarizado con el baño de perros o que el proceso fuera mucho más complicado (imagine que debe bañar 100 perros al mismo tiempo mientras les aplica medicamento contra pulgas y garrapatas, haciéndoles cortes de pelo e investigando su genealogía). Describir una lógica más complicada en una forma no estructurada sería

engorroso. Al finalizar el estudio de este capítulo entenderá cómo hacer que el proceso no estructurado de la figura 3-1 sea más claro y menos propenso a errores.



Los creadores de software dicen que un programa que contiene código espagueti tiene una vida más breve que uno cuyo código es estructurado. Esto significa que los programas que usan código espagueti existen por menos tiempo como programas de producción en una organización. Modificarlos es tan difícil que cuando se requieren mejoras los desarrolladores con frecuencia encuentran más fácil abandonar los ya existentes y empezar desde cero. Esto requiere tiempo extra y cuesta más dinero.

DOS VERDADES Y UNA MENTIRA

Las desventajas del código espagueti no estructurado

1. El nombre común para las declaraciones de un programa que desde el punto de vista lógico se encuentran enmarañadas es código espagueti.
2. Los programas escritos que usan código espagueti no pueden producir resultados correctos.
3. Los programas escritos que usan código espagueti son más difíciles de seguir que otros programas.

La afirmación falsa es la número 2. Los programas escritos que usan código espagueti pueden producir resultados correctos, pero son más difíciles de entender y mantener que los que usan técnicas estructuradas.

Comprendión de las tres estructuras básicas

A mediados de la década de 1960, los matemáticos demostraron que cualquier programa, sin importar cuán complicado sea, puede construirse usando una o más de sólo tres estructuras. Una **estructura** es una unidad básica de lógica de programación; cada estructura es una de las siguientes:

- secuencia
- selección
- ciclo

Con sólo estas tres estructuras usted puede diagramar cualquier tarea, desde duplicar un número hasta realizar una cirugía cerebral. Puede diagramar cada estructura con una configuración específica de símbolos de diagrama de flujo.

La primera de estas tres estructuras básicas es una secuencia, como se muestra en la figura 3-2. Con una **estructura de secuencia**, se ejecuta una acción o tarea, y luego se realiza la siguiente acción, en orden. Una secuencia puede contener cualquier cantidad de tareas, pero no hay opción de ramificar y saltarse alguna de éstas. Una vez que comienza una serie de acciones en una secuencia, debe continuar paso a paso hasta que la secuencia termina.

Como ejemplo, las instrucciones para conducir con frecuencia se listan como una secuencia. Para decir a un amigo cómo llegar a la casa de usted desde la escuela, podría proporcionar la siguiente secuencia, en la que un paso sigue al otro y no pueden saltarse:

ve al norte sobre First Avenue por 3 kilómetros
da vuelta a la izquierda en Washington Boulevard
ve al oeste sobre Washington por 2 kilómetros
detente en el 634 de Washington

Como se muestra en la figura 3-3 la segunda de las tres es una **estructura de selección** o **estructura de decisión**.

Con ella, usted hace una pregunta y dependiendo de la respuesta toma un curso de acción. Luego, sin importar cuál ruta siga, continúa con la siguiente tarea. (En otras palabras, un diagrama de flujo que describe una estructura de selección debe comenzar con un símbolo de decisión, y las ramas de la decisión deben unirse en la parte inferior de la estructura. El seudocódigo que describe una estructura de selección debe empezar con `if`. El seudocódigo usa la **declaración de fin de estructura** `endif` para mostrar con claridad dónde termina la estructura.)

Algunas personas llaman a la estructura de selección una **if-then-else** debido a que concuerda con la siguiente declaración:

```
if someCondition is true then
    do oneProcess
else
    do theOtherProcess
endif
```

Por ejemplo, usted podría proporcionar parte de las instrucciones para llegar a su casa como sigue:

```
if hay tráfico en Washington Boulevard then
    continúa 1 cuadra sobre First Avenue y da vuelta a la izquierda en Adams
    Lane
else
    da vuelta a la izquierda en Washington Boulevard
endif
```

Del mismo modo, un programa de nómina podría incluir una declaración como ésta:

```
if hoursWorked es mayor que 40 then
    calculate regularPay y overtimePay
else
    calculate regularPay
endif
```

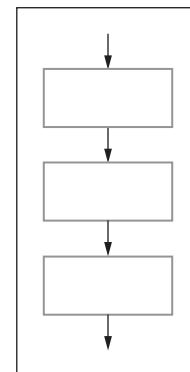


Figura 3-2 Estructura de secuencia

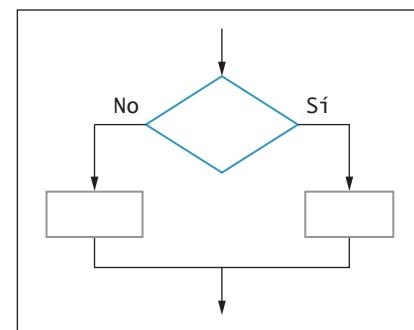


Figura 3-3 Estructura de selección

Estos ejemplos `if-else` también pueden llamarse **if de alternativa dual** (o **selecciones de alternativa dual**), porque contienen dos opciones: la acción emprendida cuando la condición probada es verdadera y la acción emprendida cuando esta última es falsa. Note que es perfectamente correcto que una rama de la selección sea “do nothing” (“no hacer nada”). En cada uno de los ejemplos siguientes sólo se emprende una acción cuando la condición probada es verdadera:

```
if está lloviendo then
    usar una sombrilla
endif
if el empleado participa en un
    programa dental then
        deducir $40 de su sueldo base
endif
```

Los ejemplos anteriores sin cláusulas `else` son **if de alternativa única** (o **selecciones de alternativa única**); en la figura 3-4 se muestra un diagrama de su estructura. En estos casos, usted no emprende alguna acción especial si no está lloviendo o si el empleado no pertenece al plan dental. El caso en el que no se hace nada con frecuencia se llama **caso nulo**.

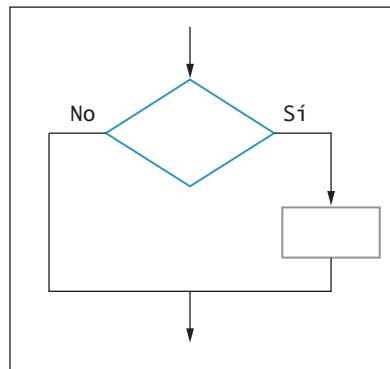


Figura 3-4 Estructura de selección de alternativa única

La última de las tres estructuras básicas, que se muestra en la figura 3-5, es un ciclo. En una **estructura en ciclo**, usted continúa repitiendo las acciones mientras una condición sigue siendo verdadera. La acción o acciones que ocurren dentro del ciclo son el **cuerpo del ciclo**. En el tipo más común de ciclo se evalúa una condición; si la respuesta es verdadera, se ejecuta el cuerpo del ciclo y se evalúa de nuevo la condición. Si ésta todavía es verdadera, se ejecuta de nuevo el cuerpo del ciclo y luego se reevalúa la condición original. Esto continúa hasta que la condición se vuelve falsa y entonces sale de la estructura. (En otras palabras, un diagrama de flujo que describe una estructura de ciclo siempre comienza con un símbolo de decisión que tiene una rama que regresa a un punto anterior a la decisión. El pseudocódigo que describe un ciclo empieza con `while` y termina con la declaración de fin de estructura `endwhile`.) Usted quizás escuche a los programadores referirse a los ciclos como **repeticIÓN** o **iteraciÓN**.

Algunos programadores llaman a esta estructura **while...do**, o simplemente, **ciclo while**, debido a que concuerda con la siguiente declaración:

```
while testCondition continue to be true
    do someProcess
endwhile
```

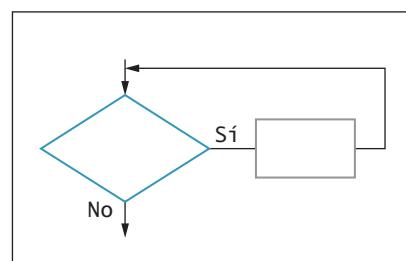


Figura 3-5 Estructura en ciclo

Cuando usted proporciona las instrucciones para llegar a su casa, algunas de ellas podrían ser:

```
while la dirección de las casas que va pasando sean menores que 634
    siga su recorrido hasta la siguiente casa
    vea la dirección de la casa
endwhile
```

Se encuentran ejemplos de ciclos todos los días, como en los siguientes:

```
while continúe teniendo hambre
    tome otro bocado de alimento
    determine si todavía siente hambre
endwhile
while queden páginas sin leer en la tarea de lectura
    lea otra página que no ha leído
    determine si ya no hay más páginas por leer
endwhile
```

Todos los problemas lógicos pueden resolverse usando sólo estas tres estructuras: secuencia, selección y ciclo. Las estructuras pueden combinarse en un número infinito de formas. Por ejemplo, usted puede tener una secuencia de tareas seguida por una selección, o un ciclo seguido por una secuencia. Las estructuras unidas por sus extremos se llaman **estructuras apiladas**. Por ejemplo, la figura 3-6 muestra un diagrama de flujo estructurado que se logra apilando estructuras y muestra el seudocódigo que sigue la lógica del diagrama de flujo.

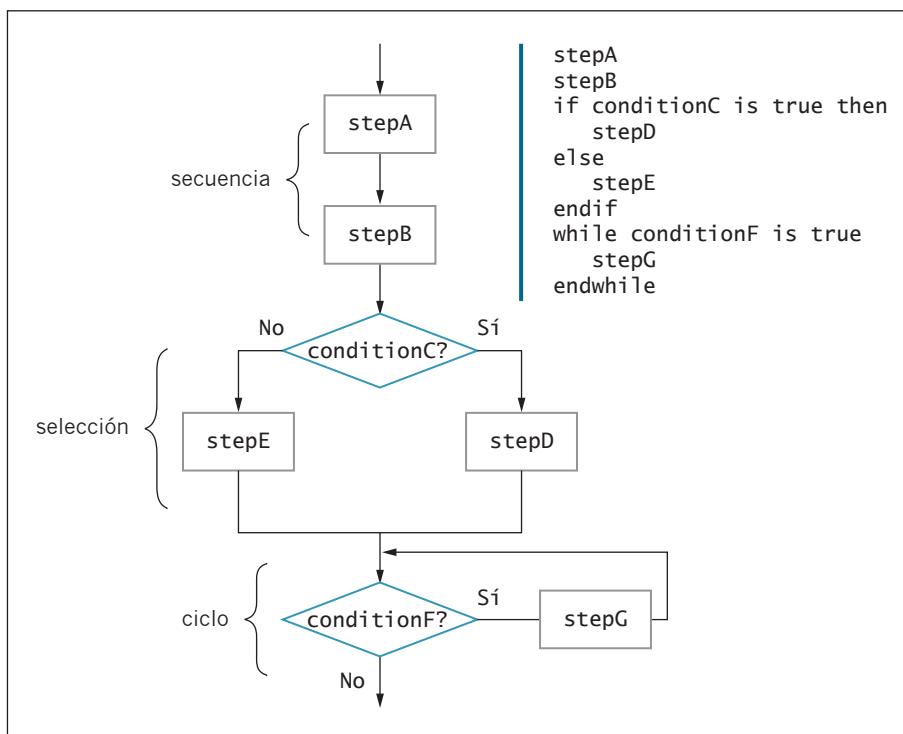


Figura 3-6 Diagrama de flujo estructurado y seudocódigo con tres estructuras apiladas



Ya sea que usted trace un diagrama de flujo o escriba un seudocódigo, puede usar cualquiera de los siguientes pares para representar los resultados de la decisión: Yes y No (Sí y No) o true y false (verdadero y falso). Este libro sigue la convención de usar Sí y No en los diagramas de flujo y verdadero y falso en el seudocódigo.

El seudocódigo en la figura 3-6 muestra una secuencia, seguida por una selección y después por un ciclo. Primero `stepA` y `stepB` se ejecutan en secuencia. Entonces inicia una estructura de selección con la prueba de `conditionC`. La instrucción que sigue a la cláusula `if` (`stepD`) ocurre cuando su condición probada es verdadera, la instrucción que sigue a `else` (`stepE`) se presenta cuando la condición probada es falsa, y cualesquier instrucciones que siguen a `endif` ocurren en cualquier caso. En otras palabras, las declaraciones más allá de la declaración `endif` están “fuera” de la estructura de decisión. Del mismo modo, la declaración `endwhile` muestra dónde termina la estructura de ciclo. En la figura 3-6, mientras `conditionF` sigue siendo verdadera, `stepG` continúa ejecutándose. Si cualesquier declaraciones siguen a la declaración `endwhile`, estarían fuera del ciclo y no serían parte de él.

Además de apilar estructuras, usted puede reemplazar todos los pasos individuales en un diagrama de flujo estructurado o un seudocódigo con estructuras adicionales. En otras palabras, cualquier secuencia, selección o ciclo puede contener otras secuencias, selecciones o ciclos. Por ejemplo, usted puede tener una secuencia de tres tareas en un lado de una selección, como se muestra en la figura 3-7. Si se coloca una estructura dentro de otra el conjunto se llama **estructuras anidadas**.

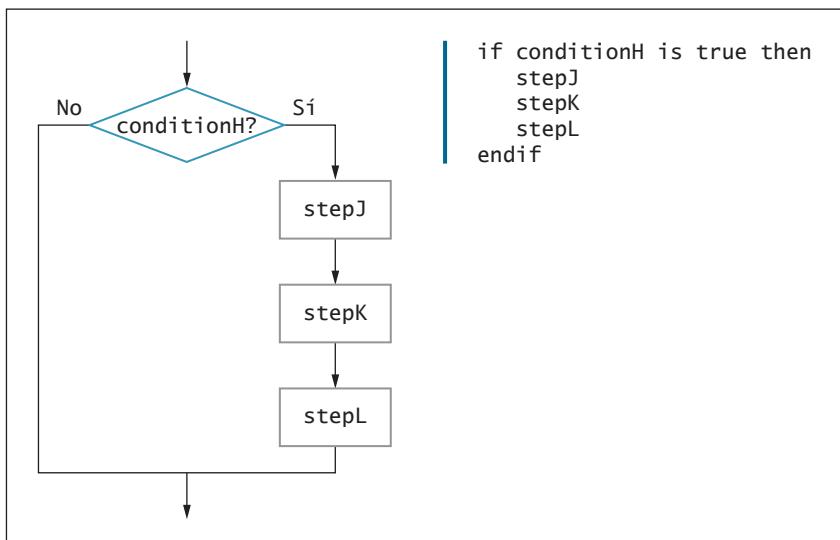


Figura 3-7 Diagrama de flujo y seudocódigo que muestran estructuras anidadas: una secuencia anidada dentro de una selección

En el seudocódigo para la lógica que se muestra en la figura 3-7, la sangría representa que las tres declaraciones (`stepJ`, `stepK` y `stepL`) deben ejecutarse si `conditionH` es verdadera. Las tres declaraciones constituyen un **bloque**, o un grupo de declaraciones que se ejecuta como una unidad.

En lugar de uno de los pasos en la secuencia de la figura 3-7, usted puede insertar otra estructura. En la figura 3-8, el proceso cuyo nombre es `stepK` se ha reemplazado por una estructura de ciclo que comienza con una prueba de la condición llamada `conditionM`.

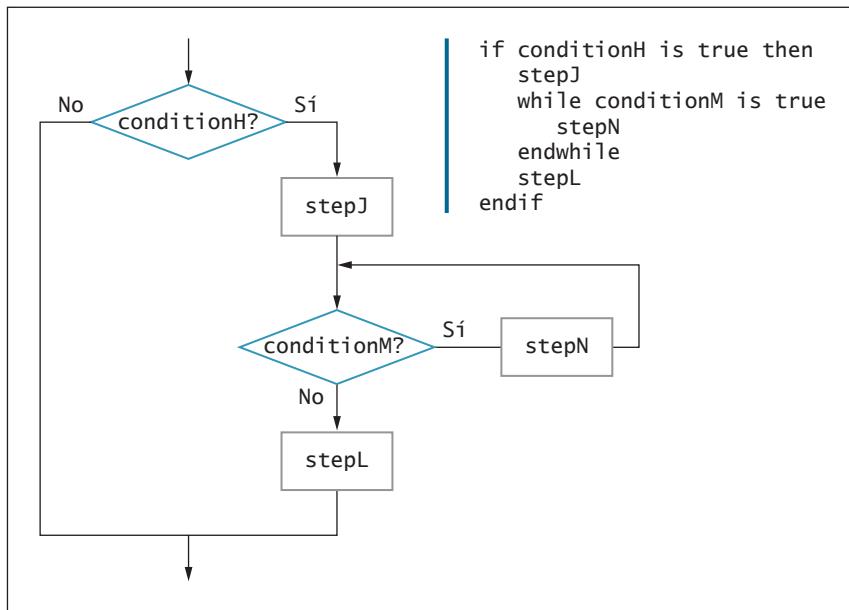


Figura 3-8 Diagrama de flujo y seudocódigo que muestran estructuras anidadas: un ciclo anidado dentro de una secuencia, anidada dentro de una selección

En el seudocódigo de la figura 3-8, note que `if` y `endif` están alineadas verticalmente. Esto muestra que están “en el mismo nivel”. Del mismo modo, `stepJ`, `while`, `endwhile` y `stepL` están alineadas y tienen la misma sangría. En el diagrama de flujo de la figura 3-8, usted podrá trazar una línea vertical a lo largo de los símbolos que contienen `stepJ`, los puntos de entrada y salida del ciclo `while` y `stepL`. El diagrama de flujo y el seudocódigo representan exactamente la misma lógica.

Cuando usted anide estructuras, las declaraciones que empiezan y terminan una estructura siempre están en el mismo nivel y en pares. Las estructuras no pueden superponerse. Por ejemplo, si tiene un `if` que contiene un `while`, entonces la declaración `endwhile` vendrá antes de `endif`. Por otra parte, si tiene un `while` que contiene un `if`, entonces la declaración `endif` vendrá antes de `endwhile`.

No hay límite para el número de niveles que puede crear cuando anide y apile estructuras. Por ejemplo, la figura 3-9 muestra lógica que se ha hecho más complicada al reemplazar `stepN` con una selección. La estructura que ejecuta `stepP` o `stepQ` con base en el resultado de `conditionO` está anidada dentro del ciclo que es controlado por `conditionM`. En el seudocódigo de la figura 3-9 note cómo los `if`, `else` y `endif` que describen la condición de selección están alineados entre sí y dentro de la estructura `while`, que es controlada por `conditionM`. Como antes, la sangría que se usa en el seudocódigo refleja la lógica trazada gráficamente en el diagrama de flujo.

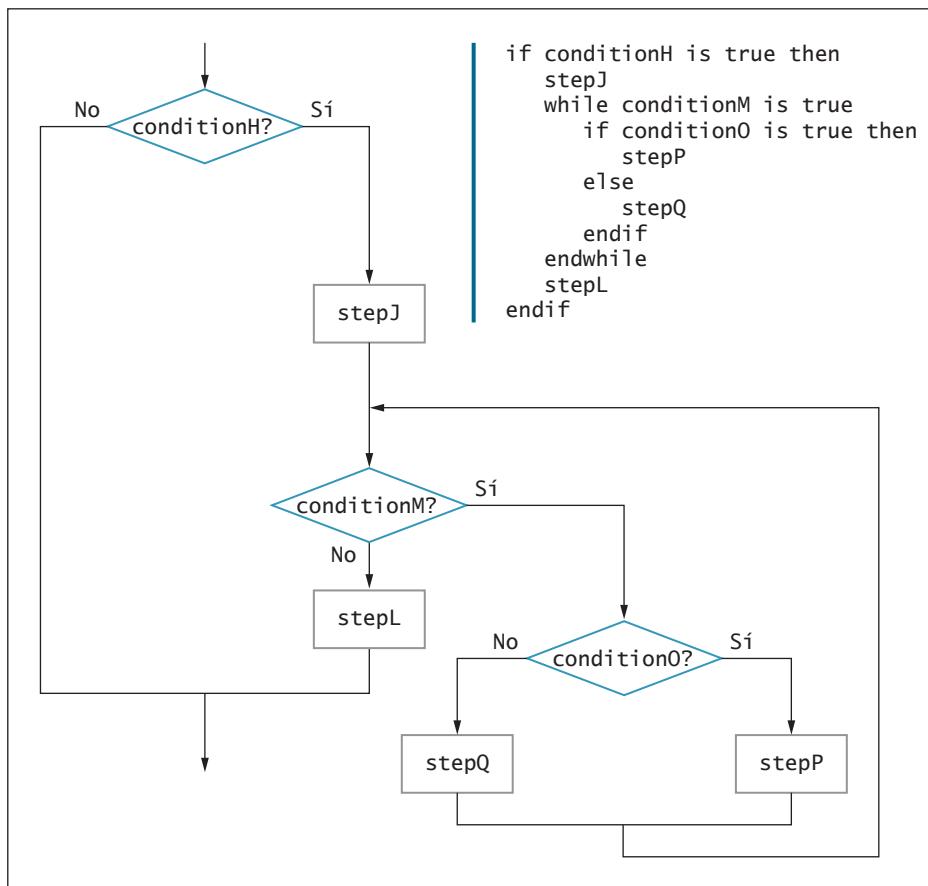


Figura 3-9 Diagrama de flujo y seudocódigo para una selección en el interior de un ciclo dentro de una secuencia contenida en una selección

Muchos de los ejemplos precedentes son genéricos de modo que usted puede enfocarse en las relaciones de las formas sin preocuparse por lo que hacen. Tenga en cuenta que las instrucciones genéricas como `stepA` y las condiciones genéricas como `conditionC` representan cualquier cosa. Por ejemplo, la figura 3-10 muestra el proceso de comprar y plantar flores al aire libre en la primavera después de que pasó el peligro de las heladas. Las estructuras del diagrama de flujo y el seudocódigo son idénticas a las de la figura 3-9. En los ejercicios al final de este capítulo, se le pedirá que desarrolle más escenarios que se ajusten al mismo patrón.

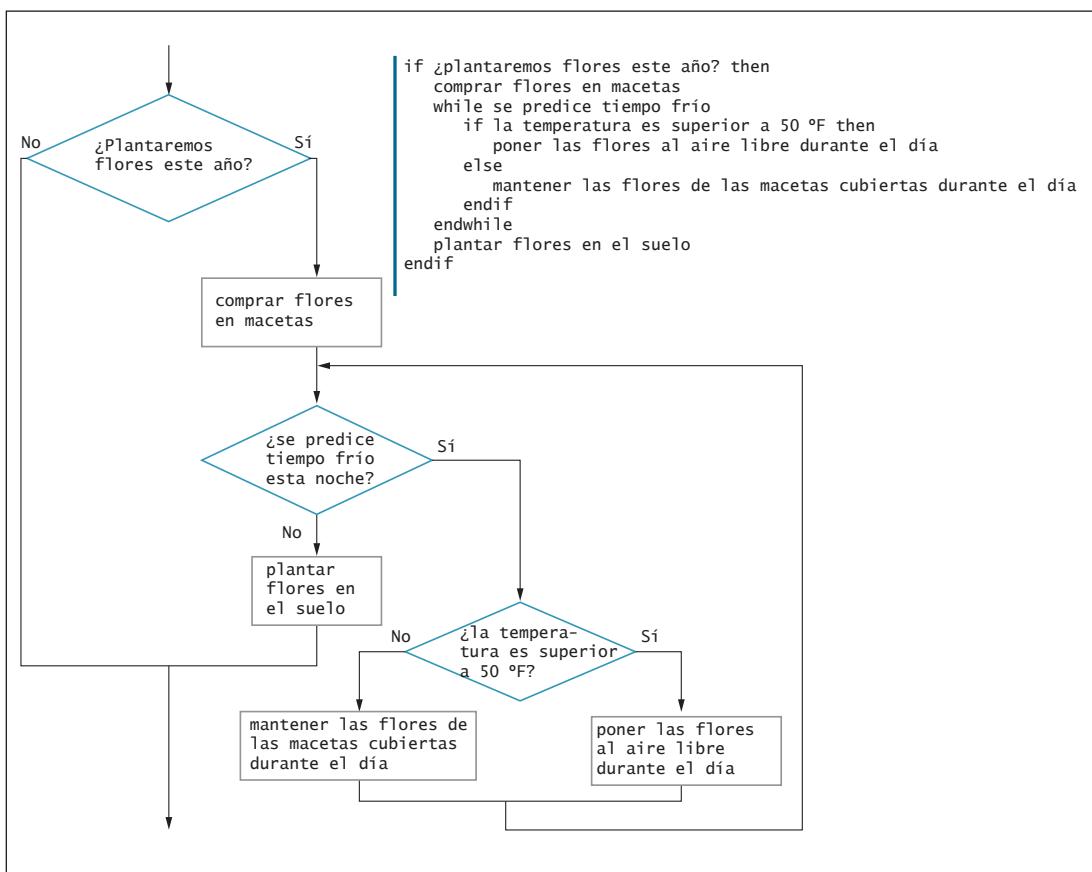


Figura 3-10 El proceso de comprar y plantar flores en la primavera

Las combinaciones posibles de las estructuras lógicas son infinitas, pero cada segmento de un programa estructurado es una secuencia, una selección o un ciclo. Las tres estructuras se muestran juntas en la figura 3-11. Note que cada estructura tiene un punto de entrada y un punto de salida. Una estructura sólo puede unirse a otra en uno de estos puntos.

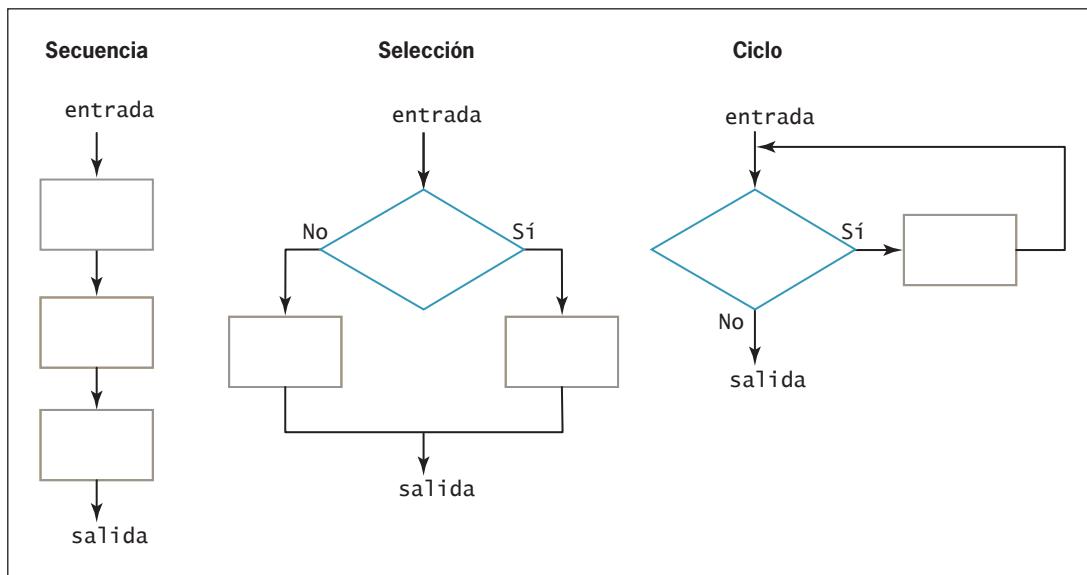


Figura 3-11 Las tres estructuras



Trate de imaginar que levanta físicamente cualquiera de las tres estructuras usando las “manijas” de entrada y salida. Éstos son los puntos en los que podría conectar una estructura con otra. Del mismo modo, cualquier estructura completa, desde su punto de entrada hasta el de salida, puede insertarse dentro del símbolo de proceso de cualquier otra estructura.

En resumen, un programa estructurado tiene las siguientes características:

- Un programa estructurado sólo incluye combinaciones de las tres estructuras básicas: secuencia, selección y ciclo. Cualquier programa estructurado podría contener uno, dos o los tres tipos de estructuras.
- Cada estructura tiene sólo un punto de entrada y uno de salida.
- Las estructuras pueden apilarse o conectarse entre sí sólo en sus puntos de entrada o salida.
- Cualquier estructura puede anidarse dentro de otra.



Nunca se requiere que un programa estructurado contenga ejemplos de las tres estructuras. Por ejemplo, muchos programas sencillos sólo contienen una secuencia de varias tareas que se ejecutan de principio a fin sin que se necesite alguna selección o ciclo. Como otro ejemplo, un programa podría desplegar una serie de números usando ciclos, pero nunca tomando alguna decisión sobre los números.

DOS VERDADES Y UNA MENTIRA

Comprensión de las tres estructuras básicas

1. Cada estructura en la programación estructurada es una secuencia, una selección o un ciclo.
2. Todos los problemas de lógica pueden resolverse usando sólo tres estructuras: secuencia, selección y ciclo.
3. Las tres estructuras no pueden combinarse en un solo programa.

en un número infinito de formas.

La afirmación falsa es la número 3. Las tres estructuras pueden aplicarse o anidarse

Uso de una entrada anticipada para estructurar un programa

Recuerde el programa para duplicar números que se mencionó en el capítulo 2; la figura 3-12 muestra un programa similar. El programa da entrada a un número y comprueba la condición de fin de archivo. Si la condición no se cumple, entonces el número se duplica, la respuesta se despliega y se introduce el siguiente número.

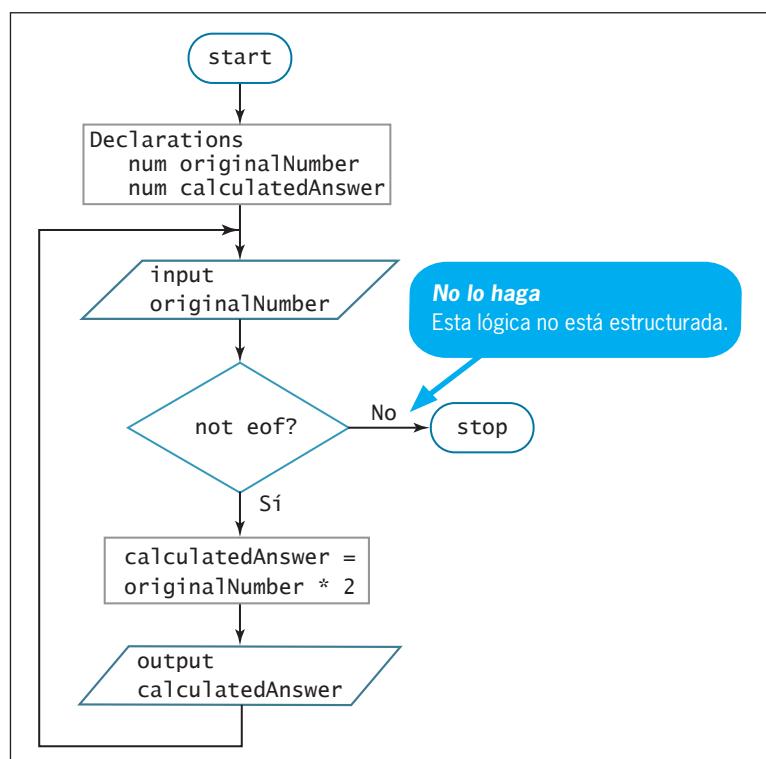


Figura 3-12 Diagrama de flujo no estructurado de un programa para duplicar números



Recuerde del capítulo 1 que este libro usa `eof` para representar una condición genérica de fin de datos cuando los parámetros exactos probados no son importantes para la exposición. En este ejemplo, la prueba es para `not eof?`, porque el procesamiento continuará mientras no se haya alcanzado el fin de los datos.

¿El programa representado por la figura 3-12 está estructurado? Al principio quizás sea difícil decirlo. Las tres estructuras permitidas se ilustraron en la figura 3-11 y el diagrama de flujo en la figura 3-12 no se ve exactamente como cualquiera de esas tres figuras. Sin embargo, debido a que usted puede apilar y anidar estructuras mientras conserva la estructura general, podría ser complicado determinar si un diagrama de flujo en conjunto está estructurado. Es más fácil analizar el diagrama de flujo en la figura 3-12 un paso a la vez. El inicio del diagrama se ve como la figura 3-13.

¿Esta porción del diagrama de flujo está estructurada? Sí, es una secuencia de dos eventos.

La adición de la siguiente pieza del diagrama de flujo se ve como la figura 3-14. La secuencia termina; empieza ya sea una selección o un ciclo. Quizás usted no sepa cuál, pero sabe que la secuencia no continúa porque las secuencias no pueden contener preguntas. Con una secuencia, cada tarea o paso debe seguir sin ninguna oportunidad de ramificarse. Así, ¿cuál tipo de estructura comienza con la pregunta en la figura 3-14? ¿Es una selección o un ciclo?

Las estructuras de selección y de ciclo difieren como sigue:

- En una estructura de selección, la lógica va en una dirección después de la pregunta, y luego el flujo regresa y se une; la pregunta no se hace por segunda vez dentro de la estructura.
- En un ciclo, si la respuesta a la pregunta da como resultado que el ciclo se introduzca y se ejecuten sus declaraciones, luego la lógica regresa a la pregunta que inició el ciclo. Cuando se ejecuta el cuerpo de un ciclo, la pregunta que controla al ciclo siempre se hace de nuevo.

Si la condición fin de archivo no se cumple en el problema de duplicar un número en la figura 3-12 original, entonces el resultado se calcula y se le da salida, se obtiene un número nuevo y la lógica regresa a la pregunta que prueba el final del archivo. En otras palabras, mientras la respuesta a la pregunta `not eof?` continúe siendo *Sí*, un conjunto de declaraciones sigue ejecutándose. Por consiguiente, la pregunta `not eof?` inicia una estructura que es más como un ciclo que como una selección.

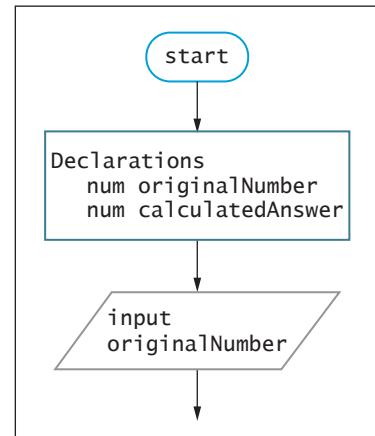


Figura 3-13 Inicio de un diagrama de flujo para duplicar un número

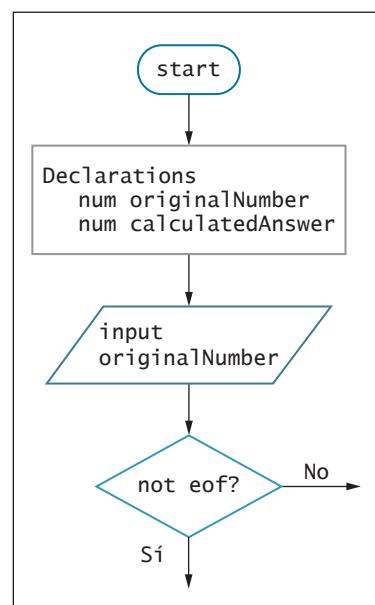


Figura 3-14 Continuación del diagrama de flujo para duplicar números

El problema para duplicar números *contiene* un ciclo, pero no es un ciclo estructurado. En un ciclo estructurado, las reglas son:

1. Usted hace una pregunta.
2. Si la respuesta le indica que debe ejecutar el cuerpo del ciclo, entonces lo hace.
3. Si ejecuta el cuerpo del ciclo, entonces debe ir directo a repetir la pregunta.

El diagrama de flujo en la figura 3-12 hace una pregunta. Si la respuesta es *Sí* (es decir, mientras `not eof?` sea verdadera), entonces el programa realiza dos tareas en el cuerpo del ciclo: hace la aritmética y despliega los resultados. Hacer dos cosas es aceptable porque dos tareas sin ramificación posible constituyen una secuencia, y está bien anidar una estructura dentro de otra. Sin embargo, cuando la secuencia termina la lógica no fluye de vuelta a la pregunta que controla el ciclo. En cambio, va *arriba* de la pregunta para obtener otro número. Para que el ciclo en la figura 3-12 sea estructurado, la lógica debe regresar a la pregunta `not eof?` cuando termina la secuencia incrustada.

El diagrama de flujo en la figura 3-15 muestra el flujo de la lógica que regresa a la pregunta `not eof?` inmediatamente después de la secuencia. La figura 3-15 muestra un diagrama de flujo estructurado, pero tiene un defecto importante, el diagrama de flujo no hace el trabajo de duplicar en forma continua diferentes números.

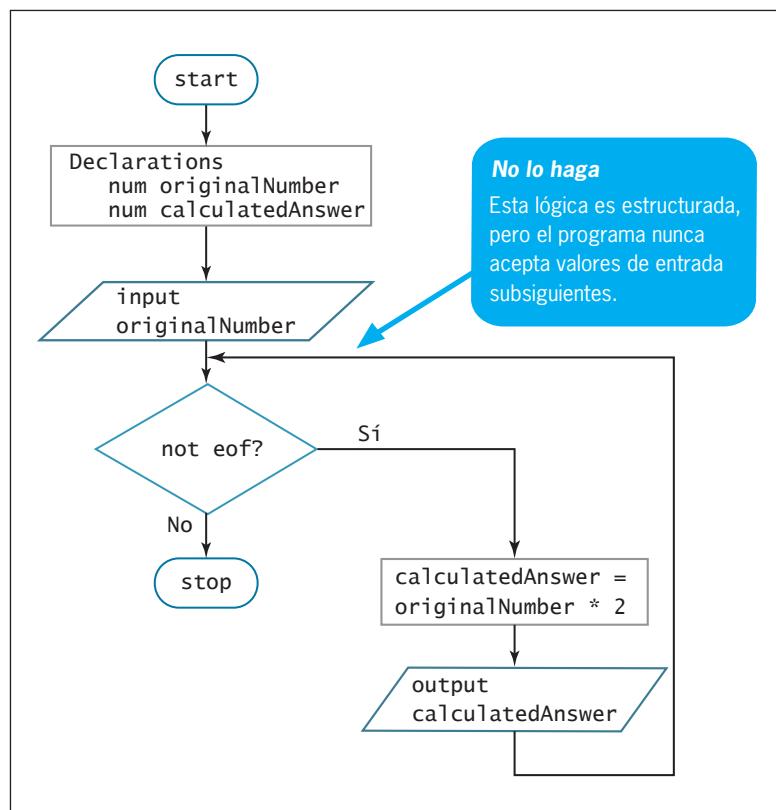


Figura 3-15 Diagrama de flujo estructurado, pero no funcional, del problema para duplicar números

Siga el diagrama de flujo a lo largo de una corrida típica del programa, suponiendo que la condición eof es un valor de entrada de 0. Suponga que cuando el programa empieza, el usuario introduce un 9 para el valor de originalNumber. Éste no es eof, así que el número se multiplica por 2 y se despliega 18 como el valor de calculatedAnswer. Entonces se hace de nuevo la pregunta not eof?. La condición not eof? debe ser verdadera todavía porque no puede introducirse un valor nuevo que represente el valor centinela (final). La lógica nunca regresa a la tarea input originalNumber, así que el valor de originalNumber nunca cambia. Por consiguiente, se duplica de nuevo 9 y la respuesta 18 se despliega de nuevo. El resultado de not eof? todavía es verdadero, así que se repiten los mismos pasos. Esto continúa *por siempre*, y la respuesta 18 es la salida de manera repetida. La lógica del programa que se muestra en la figura 3-15 es estructurada, pero no funciona como se pretende. A la inversa, el programa en la figura 3-16 funciona pero no es estructurado porque después de que las tareas se ejecutan dentro de un ciclo estructurado, el flujo de la lógica debe regresar en forma directa a la pregunta que controla el ciclo. En la figura 3-16, la lógica no regresa a esta pregunta; en cambio, va “demasiado alto” fuera del ciclo para repetir la tarea input originalNumber.

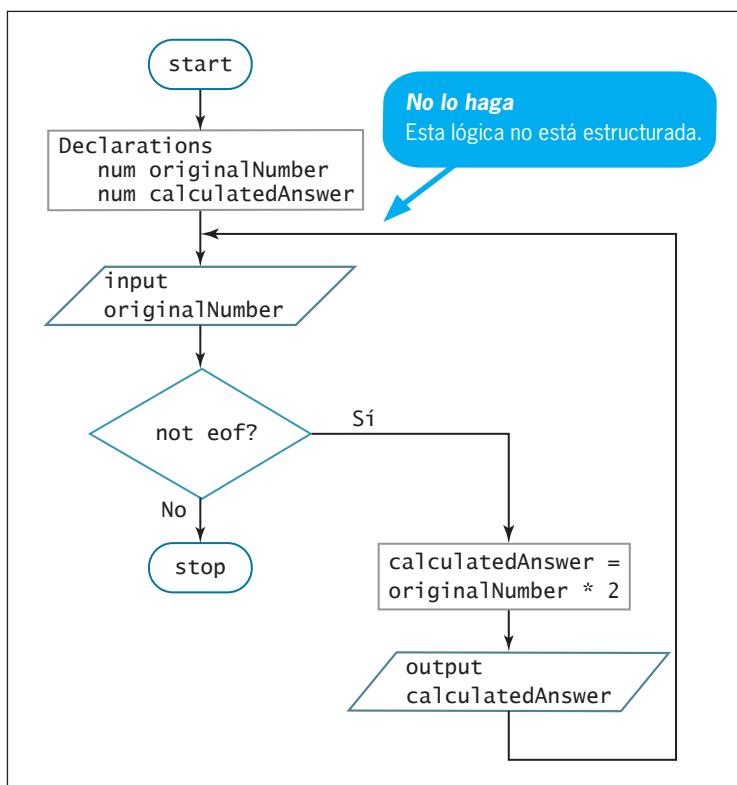


Figura 3-16 Diagrama de flujo funcional pero no estructurado

¿Cómo puede el problema para duplicar números estar estructurado y funcionar como se pretende? Con frecuencia, para que un programa sea estructurado, usted debe agregarle algo extra. En este caso, es un paso de entrada anticipada. Una **entrada anticipada** o **lectura anticipada** es una declaración agregada que obtiene el valor de la primera entrada en

un programa. Por ejemplo, si un programa recibirá 100 valores de datos como entrada, usted introduce el primer valor en una declaración que esté separada de las otras 99. Debe hacer esto para mantener estructurado el programa.

Considere la solución en la figura 3-17; está estructurada y hace lo que se supone que debe hacer. Contiene una declaración adicional `input originalNumber` sombreada. La lógica del programa contiene una secuencia y un ciclo. El ciclo contiene otra secuencia.

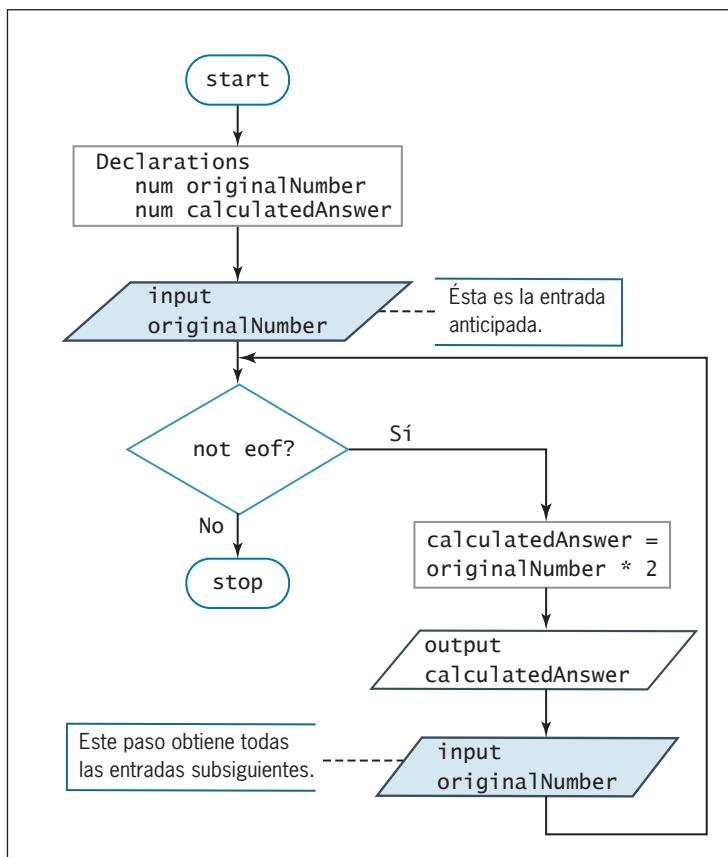


Figura 3-17 Diagrama de flujo estructurado y funcional para el problema de duplicar números

El paso adicional `input originalNumber` que se muestra en la figura 3-17 es típico en los programas estructurados. El primero de los dos pasos de entrada es la entrada anticipada. El término *anticipada* proviene del hecho de que se lee primero (inicia el proceso). El propósito del paso de entrada anticipada es controlar el ciclo próximo que comienza con la pregunta `not eof?` El último elemento del ciclo estructurado obtiene el siguiente valor de entrada y todos los subsiguientes. Esto también es típico en los ciclos estructurados: el último paso que se ejecuta dentro del ciclo altera la condición probada en la pregunta que los inicia, que en este caso es `not eof?`



En el capítulo 2 usted aprendió que el conjunto de tareas preliminares que establece el escenario para el trabajo principal de un programa se llama sección de administración. La lectura anticipada es un ejemplo de una tarea de administración.

La figura 3-18 muestra otra forma en la que usted podría trazar la lógica del programa para duplicar números. A primera vista parecería que la figura muestra una solución aceptable para el problema: es estructurada, contiene sólo un ciclo con una secuencia de tres pasos dentro de él y parece eliminar la necesidad de la declaración de entrada anticipada. Cuando el programa empieza, se plantea la pregunta `not eof?` y si no es el final de los datos de entrada, entonces el programa obtiene un número de entrada, lo duplica y lo despliega. Luego, si la condición `not eof?` sigue siendo verdadera, el programa obtiene otro número, lo duplica y lo despliega. El programa podría continuar mientras se introducen muchos números. La última vez que se ejecuta la declaración `input originalNumber`, encuentra `eof`, pero el programa no se detiene; en cambio, calcula y despliega un resultado una última vez. Dependiendo del lenguaje y del tipo de entrada que usted use, podría recibir un mensaje de error o dar basura como salida. En cualquier caso, esta última salida es extraña: no se deberá duplicar ningún valor ni darle salida después de que se encuentra la condición `eof`. Como regla general, una prueba de fin de archivo debe ir siempre inmediatamente después de una declaración de entrada porque la condición de fin de archivo se encontrará en la entrada. Por tanto, la mejor solución al problema de duplicar números sigue siendo la que se muestra en la figura 3-17, la que contiene la declaración de entrada anticipada.

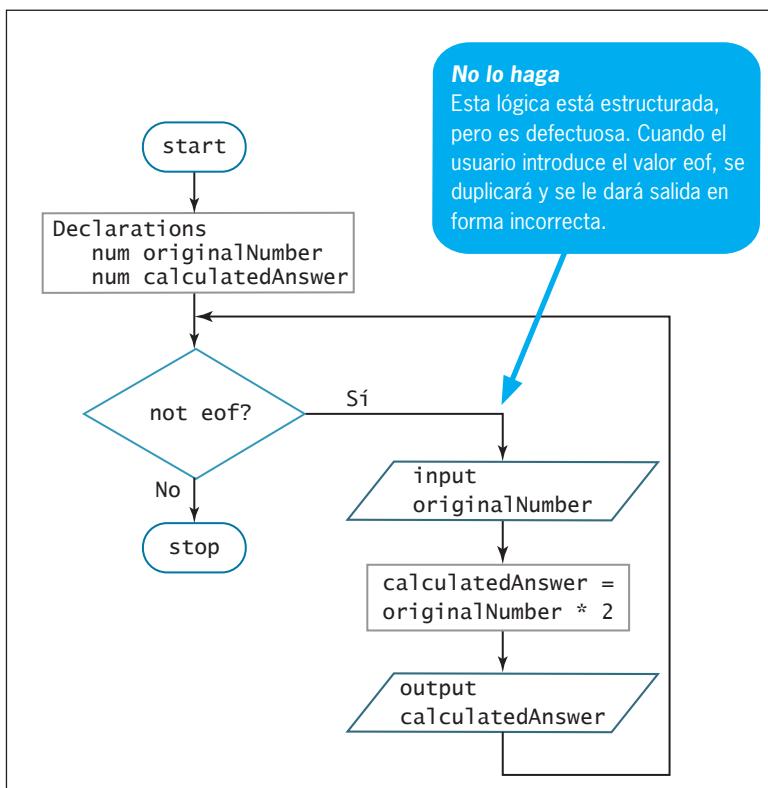


Figura 3-18 Solución estructurada pero incorrecta para el problema de duplicar números

DOS VERDADES Y UNA MENTIRA

Uso de una entrada anticipada para estructurar un programa

1. Una entrada anticipada es la declaración que obtiene de manera repetida todos los datos que son la entrada en un programa.
2. Un programa estructurado en ocasiones es más largo que uno no estructurado.
3. Un programa puede estar estructurado pero aun así ser incorrecto.

entrada.

La afirmación falsa es la número 1. Una entrada anticipada obtiene la primera

Comprensión de las razones para la estructura

En este punto, usted quizá diga: "Me gustaba el programa original para duplicar números de la figura 3-12. Podía seguirlo. Además, tenía un paso menos, así que era menos trabajo. ¿A quién le importa si un programa está estructurado?".

Hasta que tenga alguna experiencia en programación, es difícil apreciar las razones para usar sólo las tres estructuras: secuencia, selección y ciclo. Sin embargo, quedarse con estas tres es mejor por las siguientes razones:

- *Claridad.* El programa para duplicar números es pequeño. Conforme los programas se hacen más grandes, se vuelven más confusos si no están estructurados.
- *Profesionalismo.* Los demás programadores (y los profesores de programación que usted pudiera encontrar) esperan que sus programas estén estructurados; así se hacen las cosas en el ámbito profesional.
- *Eficiencia.* La mayoría de los lenguajes de computación más recientes soportan una estructura y usan sintaxis que permiten que usted aborde la secuencia, la selección y el ciclo con eficiencia. Los lenguajes más antiguos, como los ensambladores, COBOL y RPG, se desarrollaron antes de que se descubrieran los principios de la programación estructurada. Sin embargo, aun los programas que usan esos lenguajes antiguos pueden escribirse en forma estructurada. Los más recientes como C#, C++ y Java imponen la estructura por su sintaxis.



En los lenguajes más antiguos, usted podía abandonar una selección o ciclo antes de que se completara usando una declaración "go to". Ésta permitía a la lógica ir a ("go to") cualquier otra parte del programa, ya sea que estuviera dentro de la misma estructura o no. La programación estructurada en ocasiones se llama **programación sin goto**.

- *Mantenimiento.* Usted y otros programadores encontrarán más fácil modificar y dar mantenimiento a los programas estructurados cuando se requieran cambios en el futuro.
- *Modularidad.* Los programas estructurados pueden dividirse con facilidad en módulos que pueden asignarse a cualquier número de programadores. Las rutinas vuelven a unirse después como muebles modulares en cada entrada o punto de salida únicos de la rutina. Además, un módulo con frecuencia puede usarse en múltiples programas, ahorrando tiempo de desarrollo en el nuevo proyecto.

DOS VERDADES Y UNA MENTIRA

Comprensión de las razones para la estructura

1. Los programas estructurados son más claros que los no estructurados.
2. Usted y otros programadores encontrarán más fácil modificar y dar mantenimiento a los programas estructurados cuando se requieran cambios en el futuro.
3. Los programas estructurados no se dividen con facilidad en partes, lo que los hace menos propensos al error.

La afirmación falsa es la número 3. Los programas estructurados pueden dividirse con facilidad en módulos que pueden asegurarse a cualquier cantidad de programadores.

Reconocimiento de la estructura

Cuando comienza a aprender sobre el diseño de programas estructurados, es difícil que se percate de si un diagrama de flujo de la lógica de un programa está estructurado. Por ejemplo, ¿el segmento de diagrama de flujo en la figura 3-19 lo está?

Sí, lo está. Tiene la estructura de una secuencia y una selección.

¿El segmento de diagrama de flujo de la figura 3-20 está estructurado?

Sí, lo está. Tiene un ciclo, y dentro de él hay una selección.

¿El segmento de diagrama de flujo en la esquina superior izquierda de la figura 3-21 está estructurado?

No, no está construido a partir de las tres estructuras básicas. Una forma de arreglar un segmento de diagrama de flujo no estructurado es usar el método del “tazón de espagueti”; es decir, imagine el diagrama de flujo como un tazón de espagueti que debe desenredar. Suponga que puede tomar una pieza de pasta en la parte superior del tazón y jalarla. Conforme “jala” cada símbolo de la maraña puede desenredar las rutas separadas hasta que el segmento entero esté estructurado.

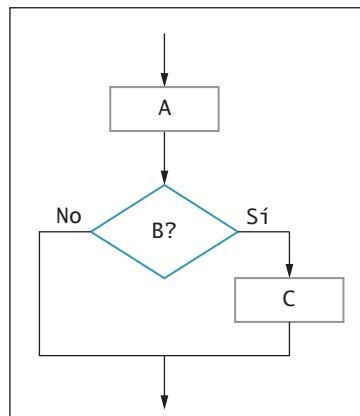


Figura 3-19 Ejemplo 1

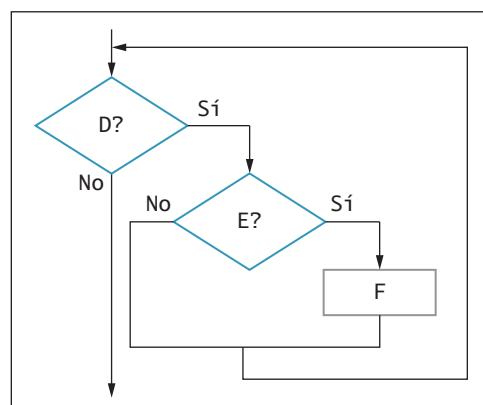


Figura 3-20 Ejemplo 2

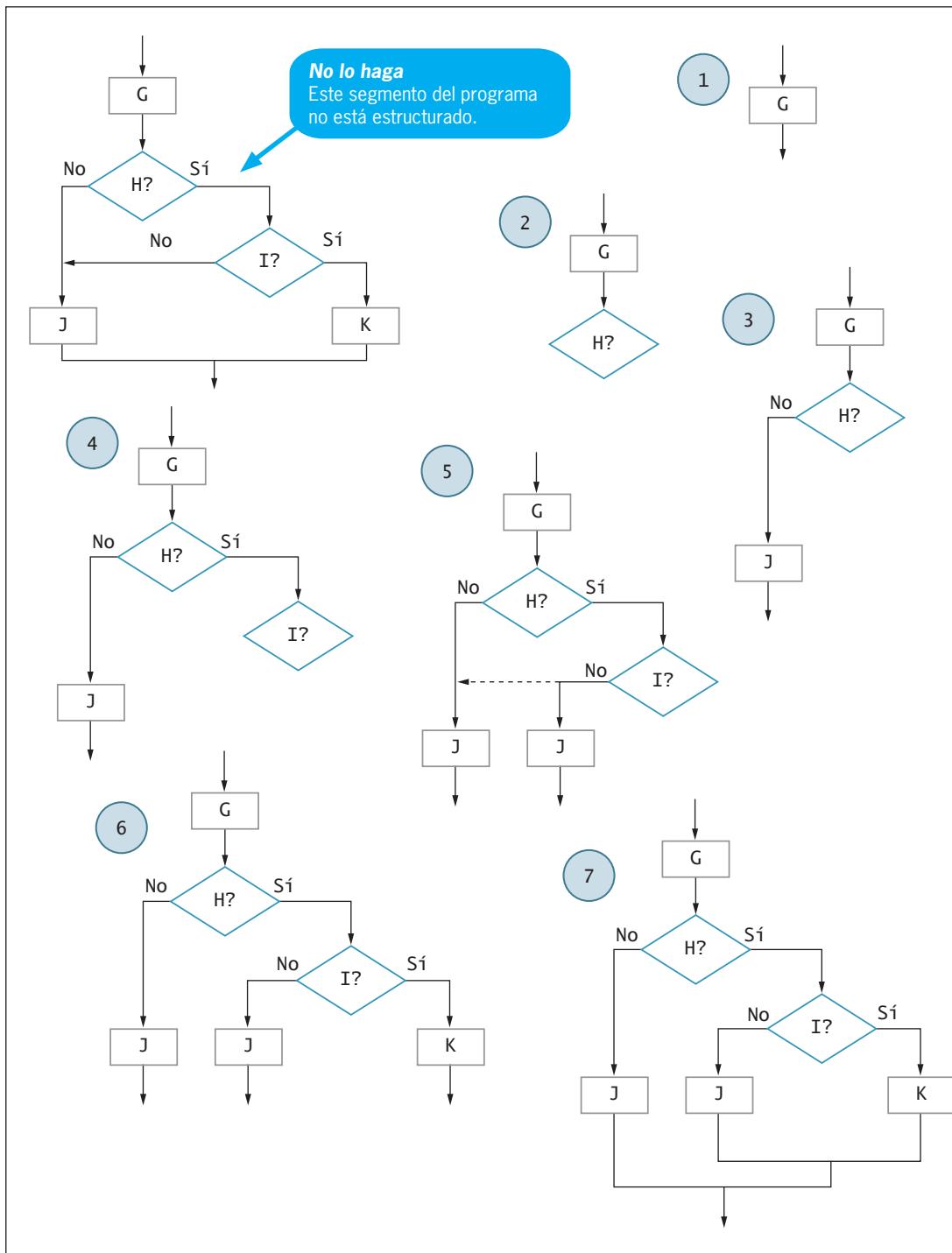


Figura 3-21 Ejemplo 3 y proceso para estructurarlo

Observe el diagrama en la esquina superior izquierda de la figura 3-21. Si pudiera jalar la flecha en la parte superior encontraría un cuadro de procedimiento etiquetado como G (véase la figura 3-21, paso 1). Un solo proceso como G es parte de una estructura aceptable; constituye al menos el comienzo de una estructura de secuencia.

Imagine que continúa jalando símbolos del segmento enredado. El siguiente elemento en el diagrama de flujo es una pregunta que prueba una condición etiquetada con H, como se observa en el paso 2 de la figura 3-21. En este punto usted sabe que la secuencia que comenzó con G ha terminado. Las secuencias nunca tienen decisiones en ellas, así que la que comenzó con G ha finalizado; una selección o un ciclo inician con la pregunta H. Un ciclo debe regresar a la pregunta que lo controla en algún punto posterior. Usted puede ver a partir de la lógica original que si la respuesta a H es *Sí* o *No*, la lógica nunca regresa a H. Por consiguiente, H comienza una estructura de selección, no una de ciclo.

Para continuar desenredando la lógica, usted jalaría la línea de flujo que surge del lado izquierdo (el lado *No*) de la pregunta H y encontraría J, como se muestra en el paso 3 de la figura 3-21. Cuando continúa más allá de J, llega al final del diagrama de flujo.

Ahora puede poner su atención en el lado *Sí* (el lado derecho) de la condición probada en H. Cuando jala el lado derecho, encuentra la pregunta I. (Véase el paso 4 de la figura 3-21.)

En la versión original del diagrama de flujo de la figura 3-21, siga la línea del lado izquierdo de la pregunta I. La línea que surge del lado izquierdo de la selección I está unida a J, que está fuera de la estructura de selección. Usted podría decir que la selección controlada por I se enreda con la selección controlada por H, así que debe desenmarañar las estructuras repitiendo el paso que genera el enredo. (En este ejemplo, repite el paso J para desenredarlo del otro uso de J.) Continúe jalando la línea de flujo que surge de J hasta que alcance el final del segmento de programa, como se muestra en el paso 5 de la figura 3-21.

Ahora jale el lado derecho de la pregunta I. El proceso K aparece, como se muestra en el paso 6 de la figura 3-21; entonces llega al final.

En este punto, el diagrama de flujo desenredado tiene tres extremos sueltos. Éstos pueden unirse para formar una estructura de selección; entonces es posible unir los extremos sueltos de la pregunta H para formar otra estructura de selección. El resultado es el diagrama de flujo que se muestra en el paso 7 de la figura 3-21. El segmento del diagrama de flujo entero está estructurado; tiene una secuencia a la que sigue una selección dentro de una selección.



Si desea estructurar un ejemplo más difícil de un programa no estructurado, vea el apéndice E.

DOS VERDADES Y UNA MENTIRA

Reconocimiento de la estructura

1. Algunos procesos no pueden expresarse en un formato estructurado.
2. Un diagrama de flujo no estructurado puede lograr resultados correctos.
3. Cualquier diagrama de flujo no estructurado puede “desenredarse” para hacerlo estructurado.

expresarse en un formato estructurado.

La afirmación falsa es la número 1. Cualquier conjunto de instrucciones puede

Estructuración y modularización de la lógica no estructurada

Recuerde el proceso para bañar perros que se ilustró en la figura 3-1, al principio de este capítulo. Si lo ve ahora reconocerá que es un proceso no estructurado. ¿Puede reconfigurarse para efectuar precisamente las mismas tareas en una forma estructurada? ¡Por supuesto!

La figura 3-22 demuestra cómo podría usted enfocar la estructuración de la lógica para bañar perros. La parte 1 de la figura muestra el inicio del proceso. El primer paso, *Atrapar al perro*, es una secuencia simple. A este paso lo sigue una pregunta. Cuando ésta se encuentra la secuencia termina y empieza ya sea un ciclo o una selección. En este caso, después de que el perro escapa, usted debe atraparlo y determinar si se escapa de nuevo, así que comienza un ciclo. Para crear un ciclo estructurado como los que ha visto antes en este capítulo, puede repetir el proceso *Atrapar al perro* y regresar de inmediato a la pregunta *¿El perro escapó?*

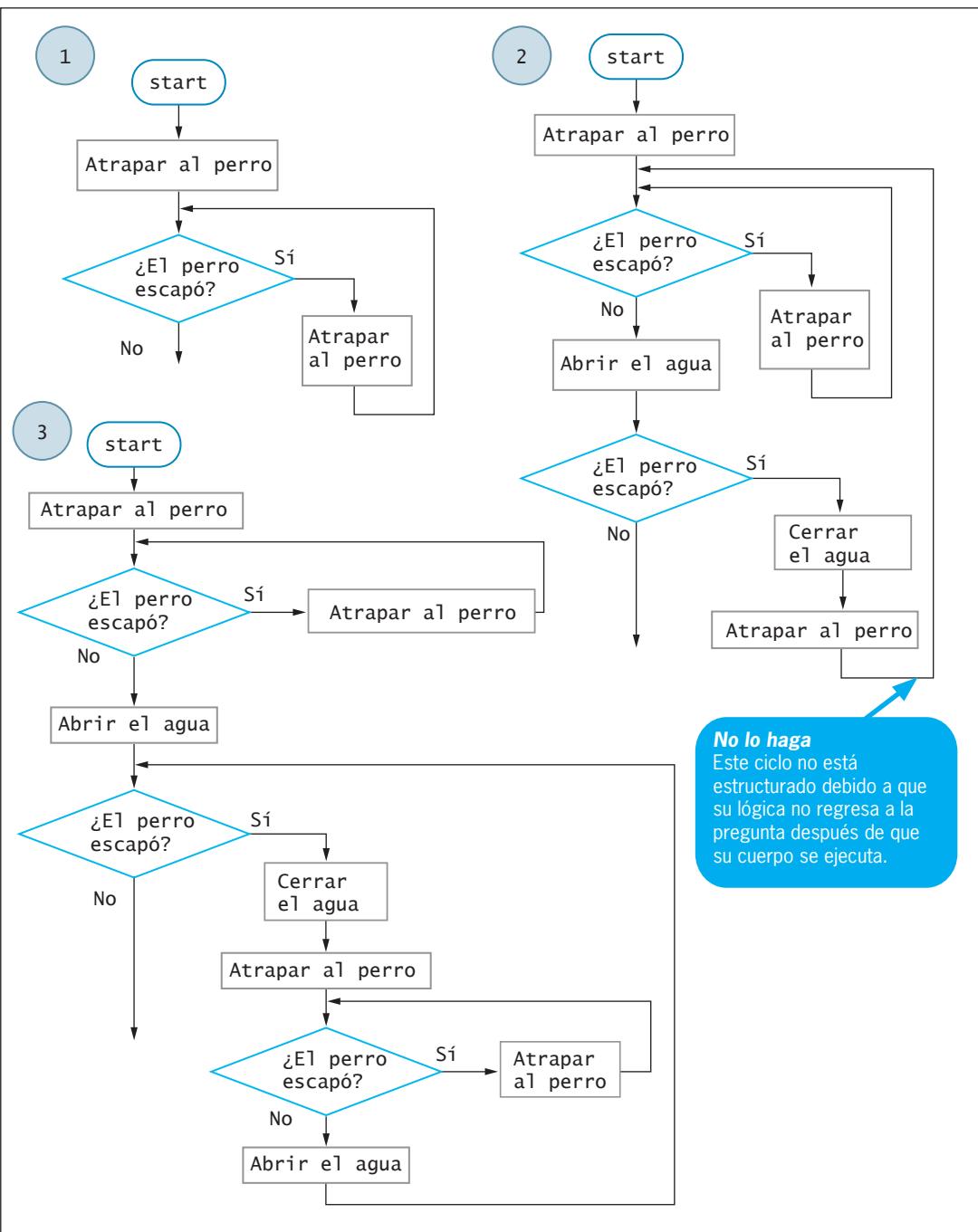


Figura 3-22 Pasos para estructurar el proceso para bañar perros

En el diagrama de flujo original en la figura 3-1, usted abre la llave del agua cuando el perro no escapa. Este paso es una secuencia simple, así que puede agregarse correctamente después del ciclo. Cuando se abre la llave del agua, la lógica original comprueba si el perro escapó después de este nuevo desarrollo. Esto inicia un ciclo. En el diagrama de flujo original las líneas se cruzan creando una maraña, de modo que usted repetirá tantos pasos como sean necesarios para desenredar las líneas. Después de que cierra la llave del agua y atrapa al perro, encuentra la pregunta *¿El perro tiene champú?* Debido a que la lógica no ha llegado aún al paso de poner champú, no hay necesidad de hacer esta pregunta; la respuesta en este punto siempre será *No*. Cuando no es posible recorrer alguna de las rutas lógicas que surgen de una pregunta, usted puede eliminar esta última. La parte 2 de la figura 3-22 muestra que si el perro escapa después de que usted abre la llave del agua, pero antes de que lo moje y le ponga champú, debe cerrar la llave, atrapar al perro y regresar al paso en el que se pregunta si el perro escapó.

La lógica en la parte 2 de la figura 3-22 no está estructurada debido a que el segundo ciclo que comienza con *¿El perro escapó?* no regresa de inmediato a la pregunta que controla al ciclo después de que se ejecuta el cuerpo del mismo. Así, para hacer el ciclo estructurado, usted puede repetir las acciones que ocurren antes de regresar a la pregunta que controla el ciclo. El segmento de diagrama de flujo en la parte 3 de la figura 3-22 está estructurado; contiene una secuencia, un ciclo, una secuencia y un ciclo final más grande. Este último contiene su propia secuencia, ciclo y secuencia.

Después de que se atrapa al perro y la llave del agua está abierta, usted lo moja y le pone champú. Luego, de acuerdo con el diagrama de flujo original en la figura 3-1, una vez más comprueba para ver si el perro se ha escapado. Si lo ha hecho, usted cierra la llave del agua y lo atrapa. Desde esta ubicación en la lógica, la respuesta a la pregunta *¿El perro tiene champú?* siempre será *Sí*; como antes, no hay necesidad de hacer una pregunta cuando sólo hay una respuesta posible. Así, si el perro escapa, se ejecuta el último ciclo. Cierra la llave, continúa atrapando al perro mientras se escapa repetidamente y abre la llave del agua. Cuando al fin atrapa al perro, lo enjuaga y termina el programa. La figura 3-23 muestra tanto el diagrama de flujo completo como el seudocódigo.

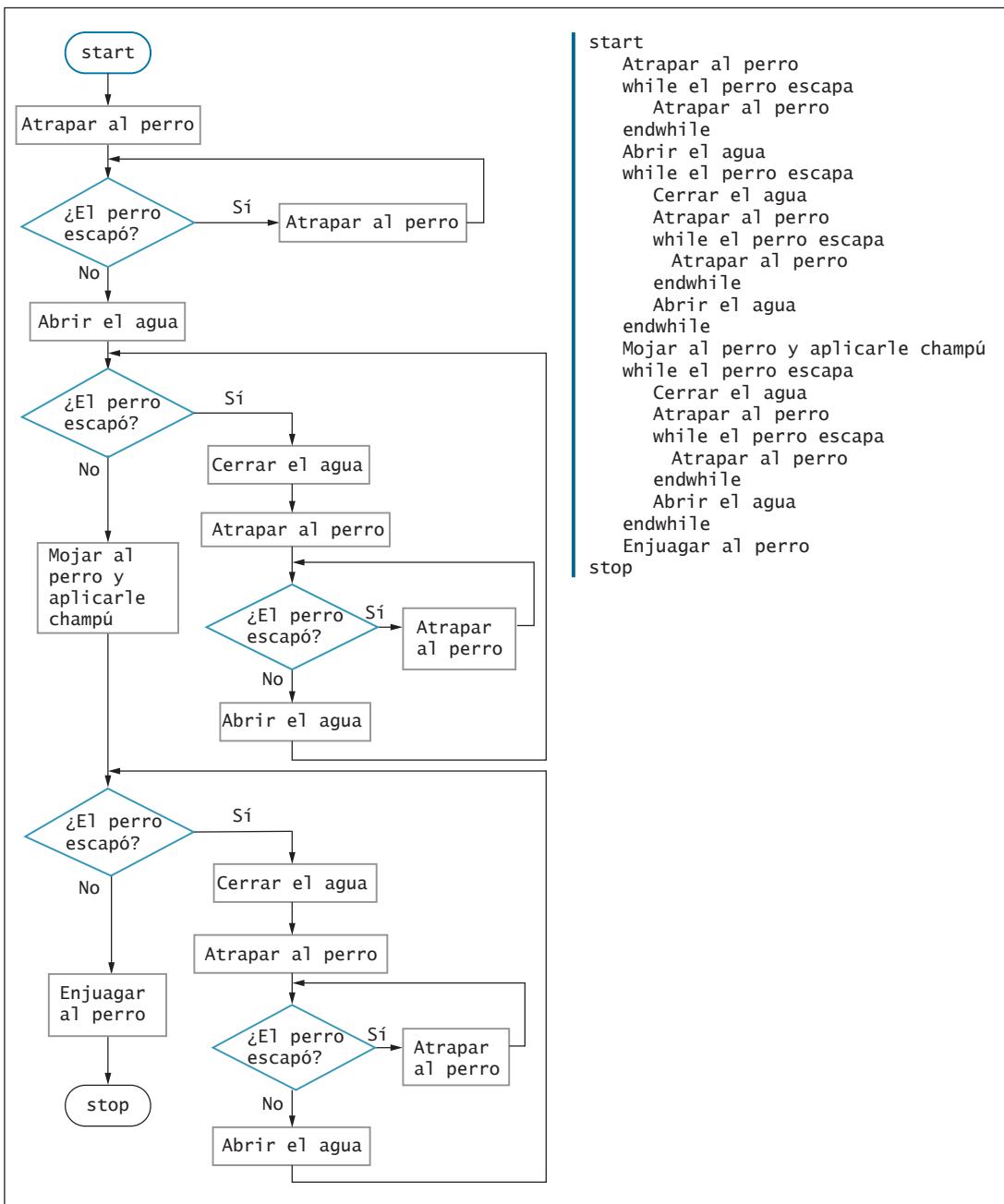


Figura 3-23 Diagrama de flujo y seudocódigo estructurados para bañar perros

El diagrama de flujo de la figura 3-23 está completo y estructurado. Contiene estructuras de secuencia y de ciclo alternadas.

La figura 3-23 incluye tres lugares donde la secuencia-ciclo-secuencia de atrapar al perro y abrir la llave del agua se repite. Si lo desea, podría modularizar las secciones duplicadas de modo que sus conjuntos de instrucciones se escriban una sola vez y queden contenidas en su propio módulo. La figura 3-24 muestra una versión modularizada del programa; las tres llamadas a módulos están sombreadas.

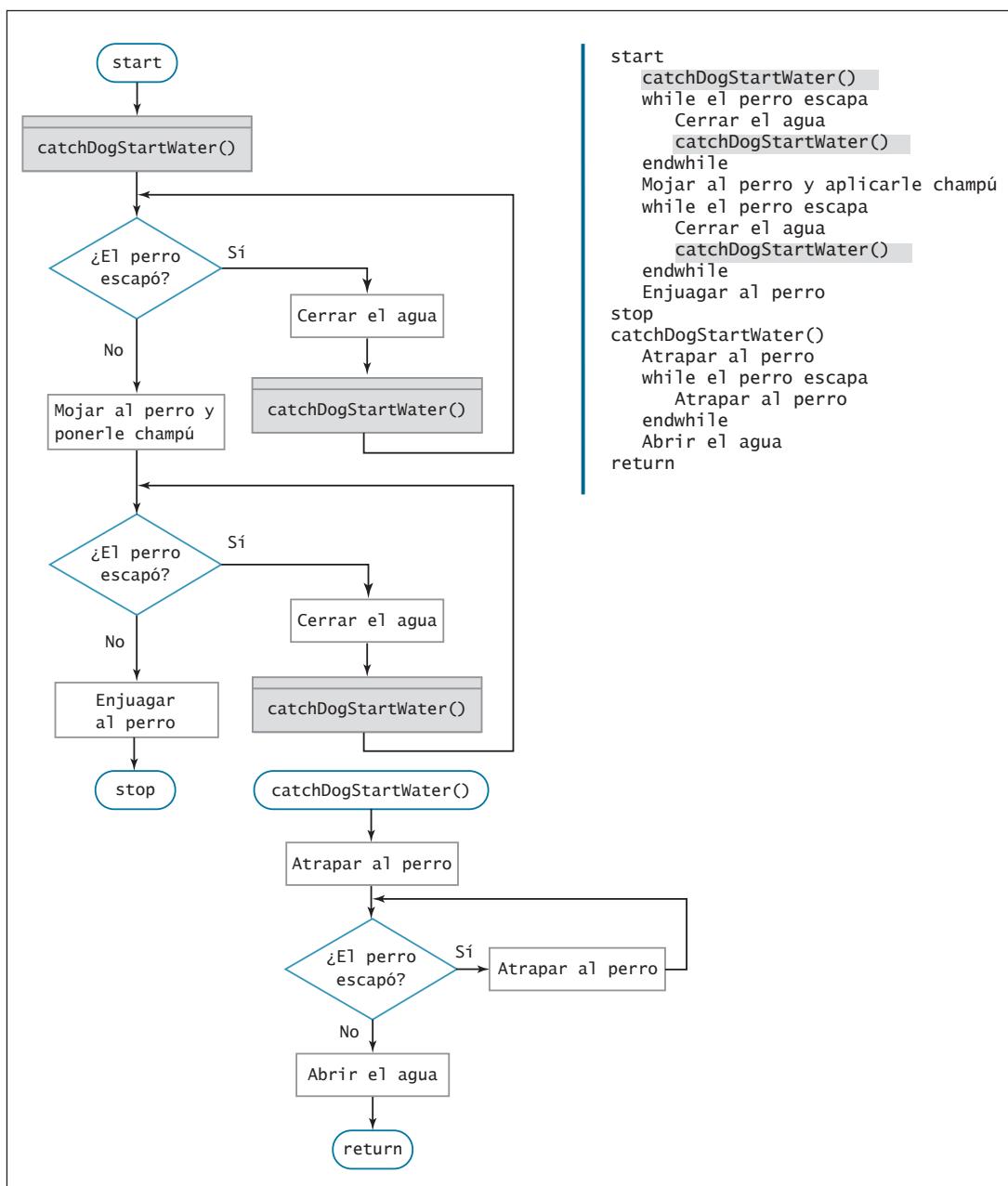


Figura 3-24 Versión modularizada del programa para bañar perros

Sin importar cuán complicado sea, cualquier conjunto de pasos puede reducirse siempre a combinaciones de las tres estructuras básicas de secuencia, selección y ciclo. Estas estructuras pueden anidarse y apilarse en un número infinito de formas para describir la lógica de cualquier proceso y crear la lógica para cada programa de computadora escrito en el pasado, el presente o el futuro.

110



Por conveniencia, muchos lenguajes de programación permiten dos variaciones de las tres estructuras básicas. La estructura **case** es una variación de la estructura selección y el ciclo **do** lo es del ciclo **while**. Usted puede aprender sobre ambas en el apéndice F. Aun cuando estas estructuras adicionales pueden usarse en casi todos los lenguajes de programación, es posible resolver todos los problemas lógicos sin ellas.

DOS VERDADES Y UNA MENTIRA

Estructuración y modularización de la lógica no estructurada

1. Cuando usted encuentra una pregunta en un diagrama lógico, una secuencia debe terminar.
2. En un ciclo estructurado, la lógica regresa a la pregunta que lo controla después de que se ejecuta el cuerpo del ciclo.
3. Si un diagrama de flujo o seudocódigo contiene una pregunta cuya respuesta nunca varía, es posible eliminar dicha pregunta.

La afirmación falsa es la número 1. Cuando encuentre una pregunta en un diagrama

lógica, deberá iniciar una selección o un ciclo. Cuálquier estructura podría terminar

antes de que se encuentre la pregunta.

Resumen del capítulo

- Código espagueti es el nombre común para las declaraciones no estructuradas que no siguen las reglas de la lógica estructurada.
- Es posible elaborar programas más claros usando sólo tres estructuras básicas: secuencia, selección y ciclo. Éstas pueden combinarse en un número infinito de formas apilándolas y anidándolas. Cada estructura tiene un punto de entrada y uno de salida; una estructura puede unirse a otra sólo en uno de estos puntos.
- Una entrada anticipada es la declaración que obtiene el primer valor de entrada antes de empezar un ciclo estructurado. El último paso dentro del ciclo obtiene el siguiente y todos los valores de entrada subsiguientes.
- Los programadores usan técnicas estructuradas para promover la claridad, el profesionalismo, la eficiencia y la modularidad.

- Una forma de ordenar un segmento de diagrama de flujo no estructurado es imaginarlo como un tazón de espagueti que usted debe desenredar.
- Cualquier conjunto de pasos lógicos puede reescribirse para que se ajuste a las tres estructuras.

Términos clave

El **código espagueti** es la lógica enredada y no estructurada de un programa.

Los **programas no estructurados** son aquellos que *no* siguen las reglas de la lógica estructurada.

Los **programas estructurados** son aquellos que siguen las reglas de la lógica estructurada.

Una **estructura** es una unidad básica de lógica de programación; cada estructura es una secuencia, una selección o un ciclo.

Una **estructura de secuencia** contiene una serie de pasos que se ejecutan en orden. Una secuencia puede contener cualquier cantidad de tareas, pero no hay opción para ramificar y omitir cualquiera de éstas.

Una **estructura de selección** o **estructura de decisión** contiene una pregunta y, dependiendo de la respuesta, toma algún curso de acción antes de continuar con la siguiente tarea.

Una **declaración de fin de estructura** designa el final de una estructura en pseudocódigo.

Un **if-then-else** es otro nombre para una estructura de selección.

Un **if de alternativa dual** (o **selecciones de alternativa dual**) define una acción que se efectuará cuando la condición probada es verdadera y otra acción cuando sea falsa.

Un **if de alternativa única** (o **selecciones de alternativa única**) efectúan una acción en una sola rama de la decisión.

El **caso nulo** es la rama de una decisión en la que no se emprende una acción.

Una **estructura de ciclo** continúa repitiendo acciones mientras una condición de prueba se mantiene verdadera.

Un **cuerpo de ciclo** es el conjunto de acciones que ocurren dentro de un ciclo.

Repetición e **iteración** son nombres alternativos para una estructura de ciclo.

En un **while... do**, o más simple, un **ciclo while**, un proceso continúa mientras alguna condición sigue siendo verdadera.

Las **estructuras apiladas** son el resultado de la unión de estructuras por sus extremos.

Las **estructuras anidadas** son el resultado de colocar una estructura dentro de otra.

Un **bloque** es un conjunto de declaraciones que se ejecuta como una unidad.

Una **entrada anticipada** o **lectura anticipada** es la declaración que lee el primer registro de datos de entrada antes de comenzar un ciclo estructurado.

Programación sin goto es un nombre que describe la programación estructurada, porque los programadores estructurados no usan una declaración “go to”.

Preguntas de repaso

17. Cuando se ejecuta un ciclo, la pregunta que controla la estructura _____.
a) se hace exactamente una vez
b) nunca se hace más de una vez
c) se hace antes o después de que se ejecuta el cuerpo del ciclo
d) se hace sólo si es verdadera y no se hace si es falsa
18. ¿Cuál de las siguientes *no* es una razón para aplicar las reglas de estructura en los programas de computadora?
a) Los programas estructurados son más claros que los no estructurados y es más fácil entenderlos.
b) Otros programadores profesionales esperan que los programas estén estructurados.
c) Los programas estructurados por lo general son más breves que los no estructurados.
d) Los programas estructurados pueden dividirse en módulos con facilidad.
19. ¿Cuál de los siguientes *no* es un beneficio de la modularización de los programas?
a) Es más fácil leer y entender los programas modulares que los que no lo son.
b) Si usted usa módulos, puede ignorar las reglas de estructura.
c) Los componentes modulares son reutilizables en otros programas.
d) Muchos programadores pueden trabajar en diferentes módulos al mismo tiempo.
20. ¿Cuál de las siguientes afirmaciones es verdadera en la lógica estructurada?
a) Usted puede usar la lógica estructurada con los lenguajes de programación más recientes, como Java y C#, pero no con los más antiguos.
b) Cualquier tarea puede describirse usando alguna combinación de las tres estructuras.
c) Los programas estructurados requieren que se divida el código en módulos fáciles de manejar, cada uno de ellos no contiene más de cinco acciones.
d) Todas las afirmaciones son verdaderas.

Ejercicios

1. En la figura 3-10 se mostró el proceso de comprar y plantar flores en primavera usando las mismas estructuras que en el ejemplo genérico de la figura 3-9. Use la misma estructura lógica de esa figura para crear un diagrama de flujo o seudocódigo que describa algún otro proceso que conozca.
2. Ningún segmento del diagrama de flujo en la figura 3-25 está estructurado. Vuelva a trazar cada uno de modo que haga lo mismo pero quede estructurado.

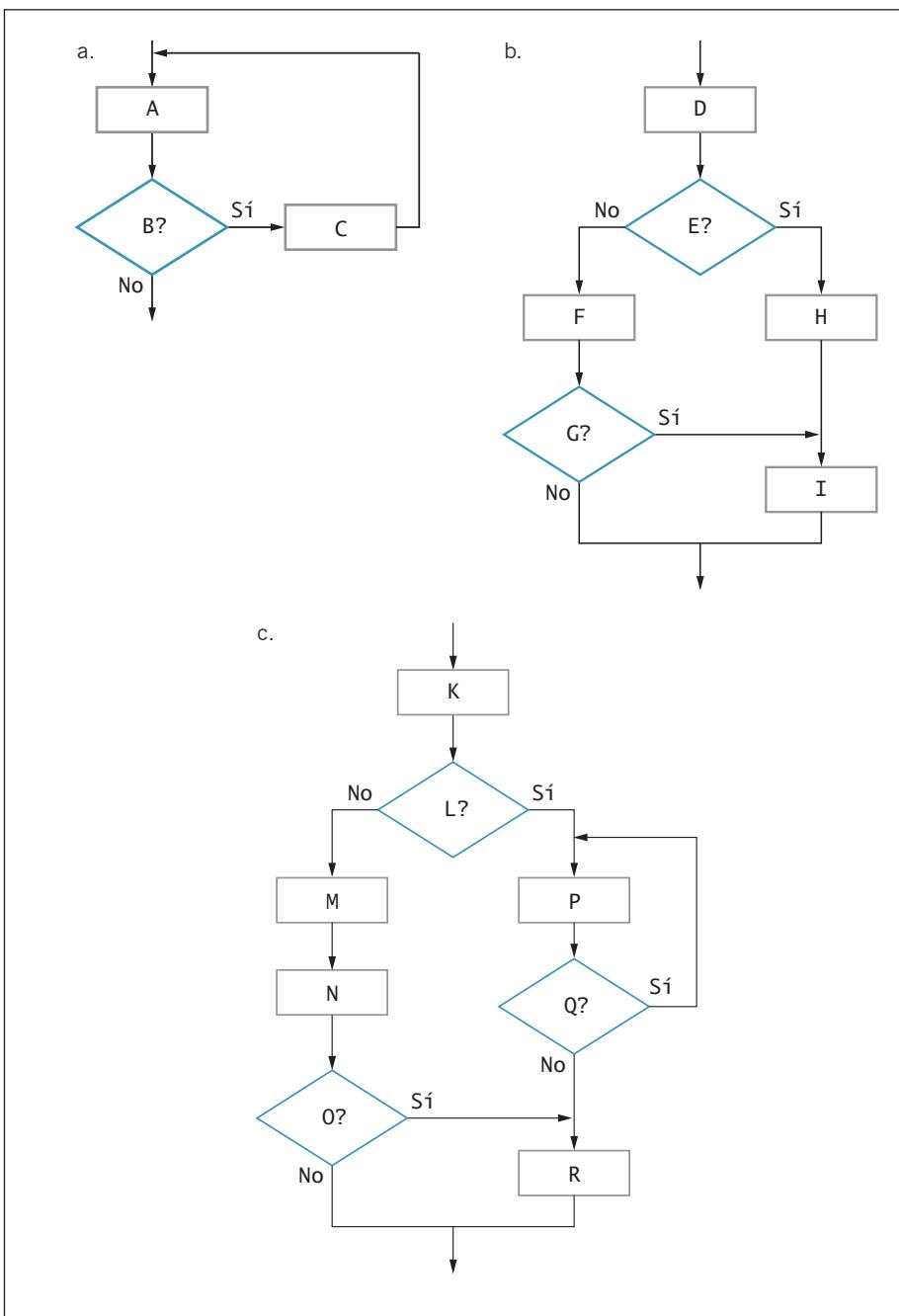


Figura 3-25 Diagramas de flujo para el ejercicio 2 (continúa)

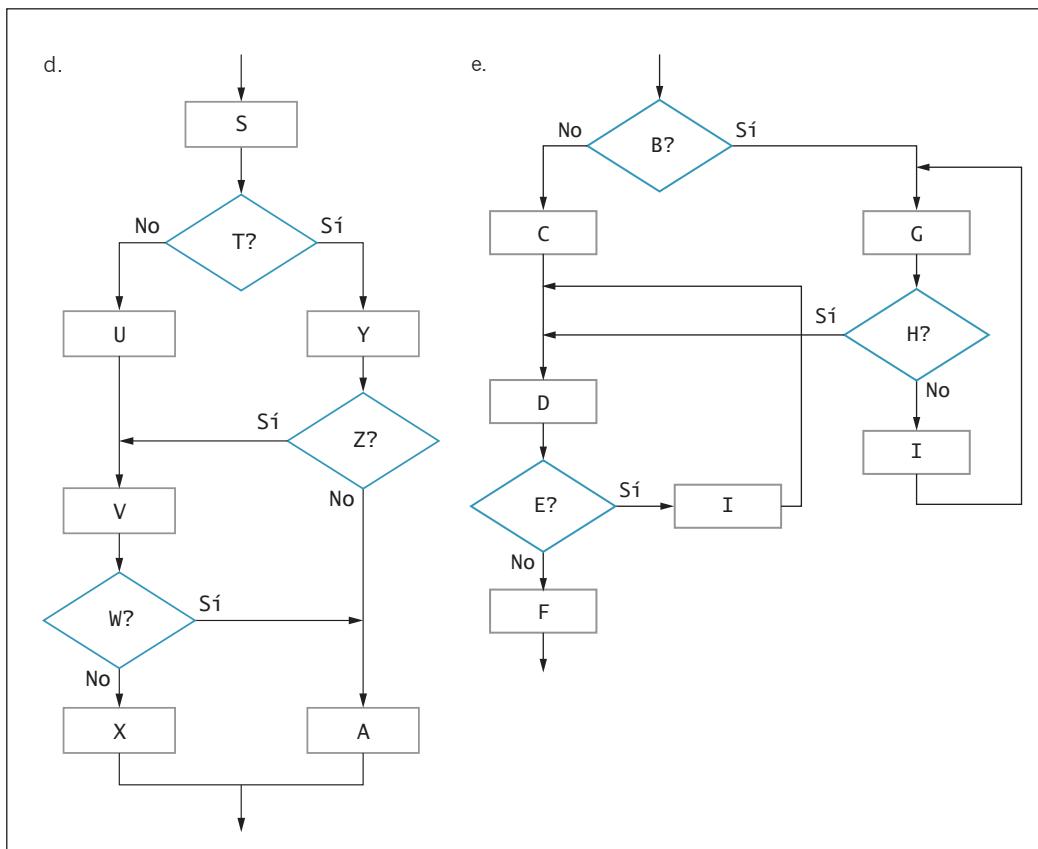


Figura 3-25 Diagramas de flujo para el ejercicio 2 (continuación)

3. Escriba un seudocódigo para cada ejemplo (de *a* hasta *e*) en el ejercicio 2, asegurándose de que su seudocódigo está estructurado pero efectúa las mismas tareas que el segmento de diagrama de flujo.
4. Suponga que ha creado un brazo mecánico que puede sostener un bolígrafo. El brazo puede realizar las siguientes tareas:
 - Bajar el bolígrafo hasta una hoja de papel.
 - Levantar el bolígrafo del papel.
 - Mover el bolígrafo 3 centímetros (o 1 pulgada) en línea recta. (Si el bolígrafo está abajo, esta acción traza una línea de 3 centímetros de izquierda a derecha; si está levantado, esta acción sólo reubica el bolígrafo 3 centímetros [o 1 pulgada] a la derecha.)
 - Girar 90° a la derecha.
 - Trazar un círculo que tenga 3 centímetros (o 1 pulgada) de diámetro.

Trace un diagrama de flujo estructurado o escriba un seudocódigo estructurado que explique la lógica que causaría que el brazo dibuje o escriba lo siguiente. Haga que

un(a) compañero(a) de estudios actúe como el brazo mecánico y lleve a cabo sus instrucciones. No revele el resultado deseado a su compañero(a) hasta que el ejercicio esté completo.

- a) un cuadrado de 3 centímetros (1 pulgada)
 - b) un rectángulo de 5 centímetros por 3 centímetros
 - c) una cadena de tres cuentas
 - d) una palabra corta (por ejemplo, *gato*)
 - e) un número de cuatro dígitos
5. Suponga que ha creado un robot mecánico que puede realizar las siguientes tareas:

- Ponerse de pie.
- Sentarse.
- Girar 90° a la izquierda.
- Girar 90° a la derecha.
- Dar un paso.

Además, el robot puede determinar la respuesta a una condición de prueba:

- ¿Estoy tocando algo?
 - a) Coloque dos sillas separadas 6 metros (o 20 pies), de modo que queden directamente una frente a la otra. Trace un diagrama de flujo estructurado o escriba un seudocódigo que expliquen la lógica que permitiría al robot empezar desde la posición de sentado en una silla, cruzar la habitación y sentarse en la otra silla. Haga que un compañero(a) de estudios actúe como el robot y siga sus instrucciones.
 - b) Trace un diagrama de flujo estructurado o escriba un seudocódigo que expliquen la lógica que permitiría al robot empezar desde la posición de sentado en una silla, ponerse de pie y dar vuelta a la silla, cruzar la habitación, dar vuelta a la otra silla, regresar a la primera silla y sentarse. Haga que un compañero(a) de estudios actúe como el robot y lleve a cabo sus instrucciones.
- 6. Trace un diagrama de flujo estructurado o escriba seudocódigo que expliquen el proceso de adivinar un número entre 1 y 100. Después de cada intento, se dice al jugador que la suposición es demasiado alta o demasiado baja. El proceso continúa hasta que el jugador adivina el número correcto. Escoja un número y haga que un compañero(a) trate de adivinarlo siguiendo sus instrucciones.
- 7. Buscar una palabra en un diccionario puede ser un proceso complicado. Por ejemplo, suponga que desea encontrar *lógica*. Podría abrir el diccionario en una página al azar y ver *jugo*. Sabe que esta palabra va alfabéticamente antes que *lógica*, así que avanza y ve *lagarto*. Todavía no es lo bastante adelante, así que avanza y ve *mono*. Se ha excedido, así que retrocede, y así sucesivamente. Trace un diagrama de flujo estructurado

o escriba un seudocódigo que expliquen el proceso de buscar una palabra en un diccionario. Escoja una al azar y haga que un(a) compañero(a) intente llevar a cabo sus instrucciones.

8. Trace un diagrama de flujo estructurado o escriba un seudocódigo estructurado que describan cómo hallar su salón de clases desde la entrada principal de la escuela. Incluya al menos dos decisiones y dos ciclos.
9. Trace un diagrama de flujo estructurado o escriba un seudocódigo estructurado que expliquen cómo arreglar un departamento. Incluya al menos dos decisiones y dos ciclos.
10. Trace un diagrama de flujo estructurado o escriba un seudocódigo estructurado que expliquen cómo envolver un regalo. Incluya al menos dos decisiones y dos ciclos.
11. Trace un diagrama de flujo estructurado o escriba un seudocódigo estructurado que expliquen los pasos que debe seguir el dependiente de una tienda de abarrotes para cobrarle a un cliente. Incluya al menos dos decisiones y dos ciclos.



Encuentre los errores

12. Sus archivos descargables para el capítulo 3 incluyen DEBUG03-01.txt, DEBUG03-02.txt y DEBUG03-03.txt. Cada archivo empieza con algunos comentarios que describen el problema. Los comentarios son líneas que comienzan con dos diagonales(//). Después de los comentarios, cada archivo contiene seudocódigo que tiene uno o más errores que debe encontrar y corregir. (NOTA: Estos archivos se encuentran disponibles sólo para la versión original en inglés.)



Zona de juegos

13. Elija un juego simple para niños y describa su lógica, usando un diagrama de flujo estructurado o un seudocódigo. Por ejemplo, podría explicar “Piedra, papel o tijeras”; el “Juego de las sillas”; el juego de naipes “Guerras”; o el juego de eliminación “De tin marín”.
14. Elija un programa de concurso de televisión como *Jeopardy!* o *100 mexicanos dijeron* y describa sus reglas usando un diagrama de flujo estructurado o un seudocódigo.
15. Elija un deporte como béisbol o fútbol americano y describa las acciones en un periodo limitado de juego (como un turno al bate en el béisbol o una posesión en el fútbol americano) usando un diagrama de flujo estructurado o un seudocódigo.



Para discusión

16. Encuentre más información sobre una de las siguientes personas y explique por qué él o ella es importante en el ámbito de la programación estructurada: Edsger Dijkstra, Corrado Bohm, Giuseppe Jacopini y Grace Hopper.
17. Los programas de computadora pueden contener unas estructuras dentro de otras y estructuras apiladas, con lo que se crean programas muy grandes. Las computadoras también pueden realizar millones de cálculos aritméticos en una hora. ¿Cómo es posible saber que los resultados son correctos?
18. Elabore una lista de verificación de reglas que puede usar para determinar si un segmento de diagrama de flujo o de seudocódigo están estructurados.

CAPÍTULO 4

Toma de decisiones

En este capítulo usted aprenderá sobre:

- ◎ Expresiones booleanas y la estructura de selección
- ◎ Los operadores de comparación relacional
- ◎ Lógica AND
- ◎ Lógica OR
- ◎ Hacer selecciones dentro de rangos
- ◎ Precedencia cuando se combinan operadores AND y OR

Expresiones booleanas y la estructura de selección

La razón por la que las personas piensan con frecuencia que las computadoras son inteligentes es la capacidad que tienen sus programas para tomar decisiones. Un programa de diagnóstico médico que puede decidir si los síntomas concuerdan con varios perfiles de enfermedades parece bastante inteligente, al igual que uno que ofrece diferentes rutas de vacaciones potenciales con base en su destino.

Cada decisión que usted toma en un programa de computadora implica la evaluación de una **expresión booleana**, cuyo valor sólo puede ser verdadero o falso. La evaluación verdadero/falso es natural desde el punto de vista de una computadora, porque su circuitería consiste en interruptores de encendido-apagado de dos estados, que a menudo se representan con 1 o 0. Cada decisión de la computadora produce un resultado verdadero o falso, sí o no, 1 o 0. En cada estructura de selección se usa una expresión booleana; dicha estructura no es nueva para usted, es una de las estructuras básicas sobre las que aprendió en el capítulo 3. Véanse las figuras 4-1 y 4-2.



El matemático George Boole (1815-1864) enfocó la lógica de manera más sencilla que sus predecesores, al expresar las selecciones lógicas con símbolos algebraicos comunes. Se considera el fundador de la lógica matemática y las expresiones booleanas (verdadero/falso) reciben su nombre de él.

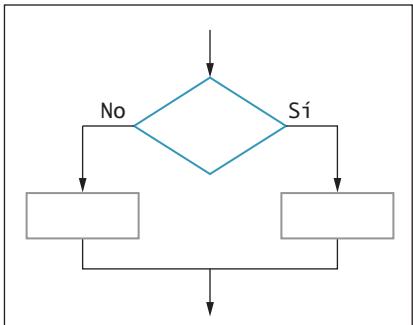


Figura 4-1 La estructura de selección de alternativa dual

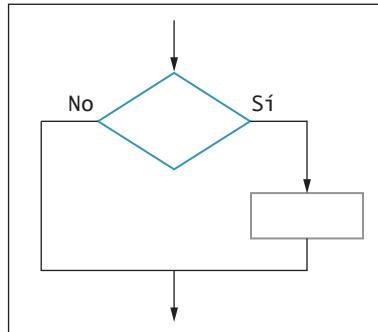


Figura 4-2 La estructura de selección de alternativa única

En el capítulo 3 usted aprendió que la estructura de la figura 4-1 es una selección de alternativa dual, o binaria, debido a que una acción se asocia con cada uno de los dos resultados posibles: dependiendo de la respuesta a la pregunta representada por el diamante, el flujo lógico procede ya sea a la rama izquierda de la estructura o a la derecha. Las opciones son mutuamente excluyentes; es decir, la lógica puede fluir sólo a una de las dos alternativas, nunca a ambas.



Este libro sigue la convención de que las dos rutas lógicas que salen de una decisión se trazan a la derecha y a la izquierda de un diamante en un diagrama de flujo. Algunos programadores trazan una de las líneas de flujo saliendo de la parte inferior del diamante. El formato exacto del diagrama no es tan importante como la idea de que una ruta lógica fluye hacia adentro de una selección y dos resultados posibles salen.

El segmento de diagrama de flujo en la figura 4-2 representa una selección de alternativa única en la que sólo se requiere acción para un resultado de la pregunta. Esta forma de la estructura de selección se llama **if-then**, porque no se necesita una acción alternativa o **else**.

Expresiones booleanas y la estructura de selección

La figura 4-3 muestra el diagrama de flujo y seudocódigo para un programa interactivo que calcula el pago de los empleados. El programa despliega el salario semanal para cada empleado con la misma tarifa por hora (\$10.00 [todas las cantidades se presentan en dólares estadounidenses]) y supone que no hay deducciones en la nómina. La lógica de línea principal llama a los módulos `housekeeping()`, `detailLoop()` y `finish()`. El módulo `detailLoop()` contiene una decisión `if-then-else` típica que determina si un empleado ha trabajado más que una semana laboral estándar (40 horas) y paga una y media veces la tarifa por hora establecida por las horas trabajadas en exceso de 40 por semana.

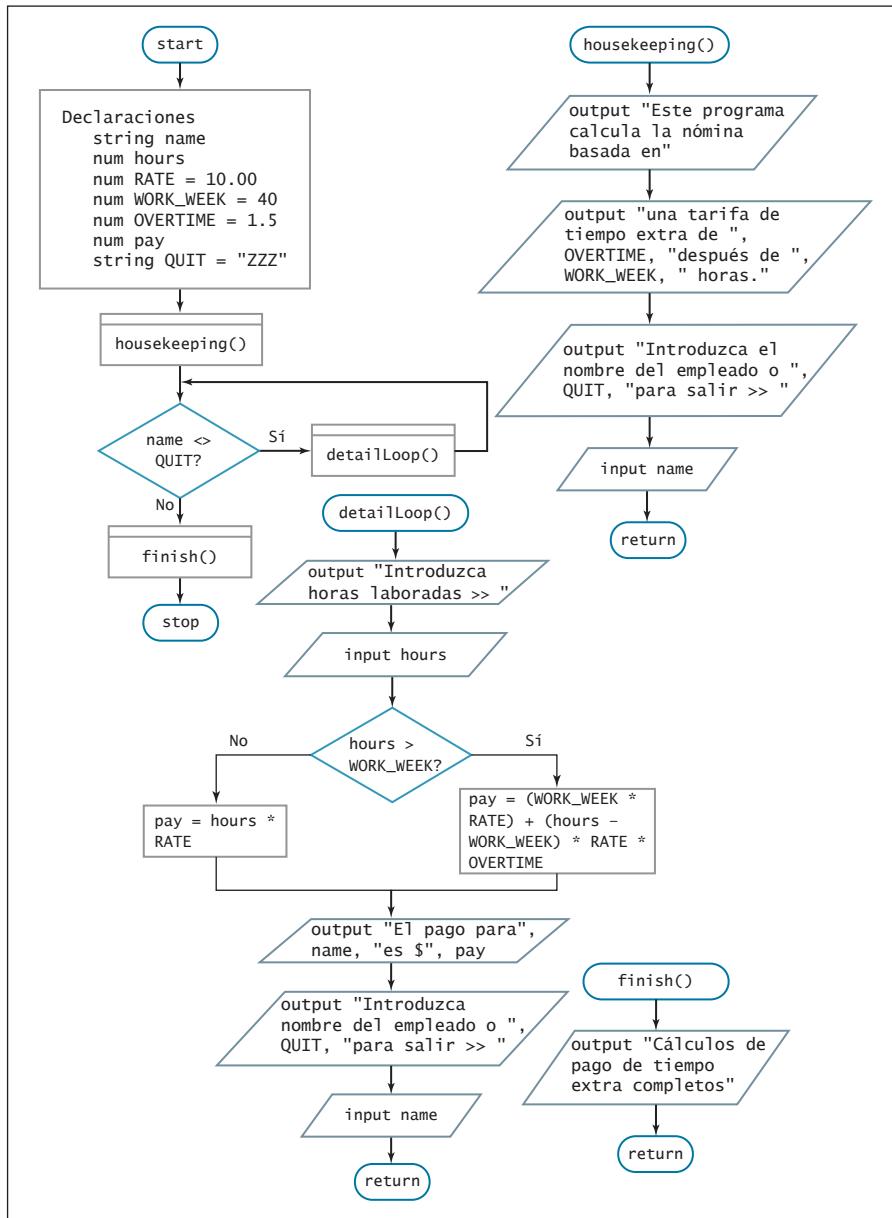


Figura 4-3 Diagrama de flujo y seudocódigo para un programa de nómina con tiempo extra (continúa)

```

start
  Declarations
    string name
    num hours
    num RATE = 10.00
    num WORK_WEEK = 40
    num OVERTIME = 1.5
    num pay
    string QUIT = "ZZZ"
  housekeeping()
  while name <> QUIT
    detailLoop()
  endwhile
  finish()
stop

housekeeping()
  output "Este programa calcula la nómina basada en "
  output "una tarifa de tiempo extra de ", OVERTIME, "después de ", WORK_WEEK, "horas."
  output "Introduzca el nombre del empleado o ", QUIT, "para salir >> "
  input name
return

detailLoop()
  output "Introduzca las horas trabajadas >> "
  input hours
  if hours > WORK_WEEK then
    pay = (WORK_WEEK * RATE) + (hours - WORK_WEEK) * RATE * OVERTIME
  else
    pay = hours * RATE
  endif
  output "El pago para ", name, "es $", pay
  output "Introduzca el nombre del empleado o ", QUIT, "para salir >> "
  input name
return

finish()
  output "Cálculos de pago de tiempo extra completos"
return

```

Figura 4-3 Diagrama de flujo y seudocódigo para un programa de nómina con tiempo extra (continuación)



A lo largo de este libro se presentan muchos ejemplos tanto en forma de diagrama de flujo como de seudocódigo. Cuando usted analice una solución, quizás encuentre más fácil concentrarse al principio en sólo una de las dos herramientas de diseño. Cuando entienda cómo funciona el programa usando una herramienta (por ejemplo, el diagrama de flujo), puede confirmar que la solución es idéntica usando la otra herramienta.

En el módulo `detailLoop()` del programa en la figura 4-3, la decisión contiene dos cláusulas:

- La **cláusula if-then** es la parte de la decisión que contiene la acción o acciones que se ejecutan cuando la condición probada en la decisión es verdadera. En este ejemplo, la cláusula contiene el cálculo de tiempo extra más largo.
- La **cláusula else** de la decisión es la parte que se ejecuta sólo cuando la condición probada en la decisión es falsa. En este ejemplo, la cláusula contiene el cálculo más breve.

La figura 4-4 muestra una ejecución común del programa en un ambiente de línea de comandos. Se introducen los valores de datos para tres empleados; los primeros dos empleados no trabajaron más que 40 horas, así que su salario se despliega simplemente como horas por \$10.00. El tercer empleado, sin embargo, ha trabajado una hora de tiempo extra, y por tanto gana \$15.00 por la última hora en lugar de \$10.00.

```

Este programa calcula la nómina basada en
una tarifa de tiempo extra de 1.5 después de 40 horas.
Introduzca el nombre del empleado o ZZZ para salir >> Wilson
Introduzca las horas laboradas >> 39
El pago para Wilson es $390
Introduzca el nombre del empleado o ZZZ para salir >> Dauphin
Introduzca las horas laboradas >> 40
El pago para Dauphin es $400
Introduzca el nombre del empleado o ZZZ para salir >> Penn
Introduzca las horas laboradas >> 41
El pago para Penn es $415
Introduzca el nombre del empleado o ZZZ para salir >> ZZZ
Cálculos de pago de tiempo extra completos
  
```

Figura 4-4 Ejecución típica del programa de nómina con tiempo extra de la figura 4-3

DOS VERDADES Y UNA MENTIRA

Expresiones booleanas y la estructura de selección

1. La cláusula **if-then** es la parte de una decisión que se ejecuta cuando una condición probada en una decisión es verdadera.
2. La cláusula **else** es la parte de una decisión que se ejecuta cuando una condición probada en una decisión es verdadera.
3. Una expresión booleana es aquella cuyo valor es verdadero o falso.

que se ejecuta cuando una condición probada en una decisión es falsa.

La afirmación falsa es la número 2. La cláusula **else** es la parte de una decisión

Uso de operadores de comparación relacionales

El cuadro 4-1 describe los seis **operadores de comparación relacionales** soportados por todos los lenguajes de programación modernos. Cada operador es binario; es decir, cada uno requiere dos operandos. Cuando se construye una expresión usando uno de los operadores que se han descrito en el cuadro 4-1, la expresión evalúa para verdadero o falso. (Note que algunos operadores se forman usando dos caracteres sin espacio entre ellos.) Por lo general, ambos operandos en una comparación deben ser del mismo tipo de datos; es decir, usted puede comparar valores numéricos con otros valores numéricos y cadenas de texto con otras cadenas. Algunos lenguajes de programación le permiten comparar un carácter con un número. Si usted lo hace, entonces en la comparación se usa sólo un código numérico de carácter. El apéndice A contiene más información sobre los sistemas de codificación. En este libro sólo se compararán los operandos del mismo tipo.

Operador	Nombre	Discusión
=	Operador de equivalencia	Evalúa como verdadero cuando sus operandos son equivalentes. Muchos lenguajes usan un signo de igual doble (==) para evitar la confusión con el operador de asignación.
>	Operador mayor que	Evalúa como verdadero cuando el operando izquierdo es mayor que el operando derecho.
<	Operador menor que	Evalúa como verdadero cuando el operando izquierdo es menor que el operando derecho.
>=	Operador mayor o igual que	Evalúa como verdadero cuando el operando izquierdo es mayor que o equivalente al operando de la derecha.
<=	Operador menor o igual que	Evalúa como verdadero cuando el operando izquierdo es menor que o equivalente al operando de la derecha.
<>	Operador no igual a	Evalúa como verdadero cuando sus operandos no son equivalentes. Algunos lenguajes usan un signo de admiración seguido por un signo de igual para indicar no igual a (!=).

Cuadro 4-1 Operadores de comparación relacionales

En cualquier expresión booleana, los dos valores comparados pueden ser variables o constantes. Por ejemplo, la expresión `currentTotal = 100?` compara una variable, `currentTotal`, con una constante numérica, 100. Dependiendo del valor `currentTotal`, la expresión es verdadera o falsa. En la expresión `currentTotal = previousTotal?`, ambos valores son variables, y el resultado también es verdadero o falso dependiendo de los valores almacenados en cada una de las dos variables. Aunque es legal, usted nunca usaría expresiones en las que compara dos constantes; por ejemplo, `20 = 20?` o `30 = 40?` Tales expresiones son **expresiones triviales** debido a que cada una siempre evalúa el mismo resultado: verdadero para `20 = 20?` y falso para `30 = 40?`

Algunos lenguajes requieren operaciones especiales para comparar cadenas, pero en este libro se supondrá que los operadores de comparación estándar funcionan en forma correcta con cadenas basadas en sus valores alfabeticos. Por ejemplo, la comparación `“azul” < “negro”?` se evaluaría como verdadera porque `“azul”` precede a `“negro”` alfabéticamente. En general, las variables de cadena no se consideran iguales a menos que sean idénticas, incluyendo el espacio y si aparecen en mayúsculas o minúsculas. Por ejemplo, `“pluma negra”` no es igual que `“plumanegra”`, `“PLUMA NEGRA”` o `“Pluma Negra”`.

Cualquier decisión puede hacerse usando combinaciones de sólo tres tipos de comparaciones: igual, mayor que y menor que. Usted nunca necesitará las tres comparaciones adicionales (mayor o igual que, menor o igual que o no igual), pero su uso frecuente hace que se tomen las decisiones más convenientes. Por ejemplo, suponga que necesita aplicar un descuento de 10% a cualquier cliente cuya edad sea de 65 años o mayor, y cobrar el precio completo a otros clientes. Puede usar el símbolo mayor o igual que para escribir la lógica como sigue:

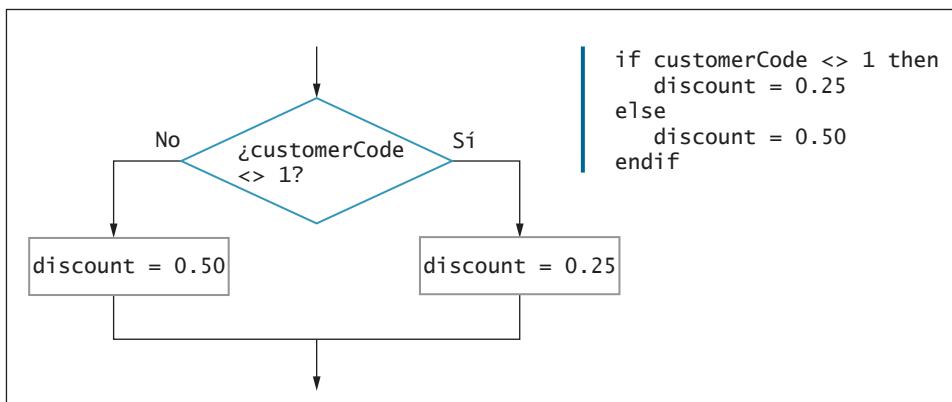
```
if customerAge >= 65 then
    discount = 0.10
else
    discount = 0
endif
```

Como alternativa, si no existe el operador `>=`, podría expresar la misma lógica escribiendo:

```
if customerAge < 65 then
    discount = 0
else
    discount = 0.10
endif
```

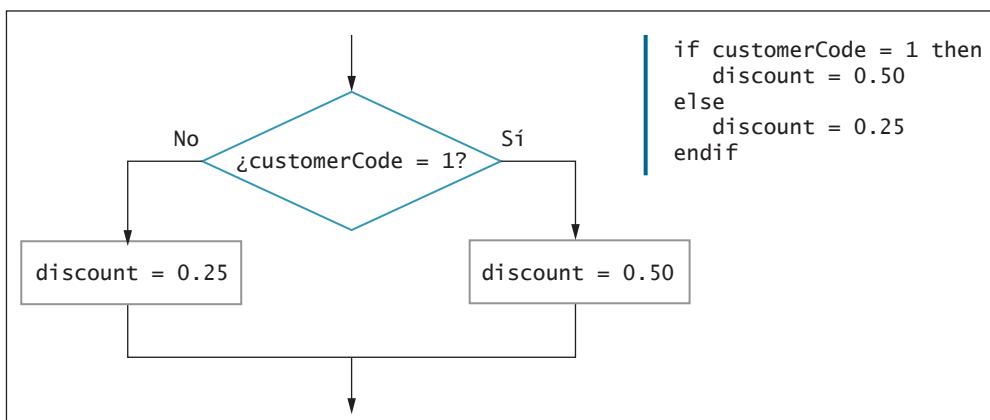
En cualquier decisión para la que `a >= b` es verdadera, entonces `a < b` es falsa. A la inversa, si `a >= b` es falsa, entonces `a < b` es verdadera. Al reformular la pregunta e intercambiar las acciones emprendidas con base en el resultado, usted puede hacer la misma decisión en múltiples formas. La ruta más clara con frecuencia es hacer una pregunta de tal manera que el resultado positivo o verdadero produzca la acción que fue su motivación para hacer la prueba. Cuando la política de su compañía es “proporcionar un descuento para aquellos que tengan 65 años y más”, la frase *mayor o igual que* viene a la mente, de modo que es más natural usarla. Por el contrario, si su política es “no aplicar descuento para aquellos menores de 65 años”, entonces es más natural usar la sintaxis *menor que*. De cualquier forma, las mismas personas recibirán el descuento.

La comparación de dos cantidades para decidir si *no* son iguales entre sí es la más confusa. Usar *no igual a* en las decisiones implica pensar en dobles negativos, que pueden hacer a usted propenso a incluir errores lógicos en sus programas. Por ejemplo, considere el segmento de diagrama de flujo en la figura 4-5.

**Figura 4-5** Uso de una comparación negativa

En la figura 4-5, si el valor de `customerCode` es igual a 1, el flujo lógico sigue la rama falsa de la selección. Si `customerCode <> 1` es verdadero, `discount` es 0.25; si `customerCode <> 1` no es verdadero, significa que `customerCode` es 1 y `discount` es 0.50. Incluso la lectura del enunciado “si `customerCode` no es igual a 1 no es verdadero” es incómoda.

La figura 4-6 muestra la misma decisión, esta vez planteada con lógica positiva. Tomar la decisión con base en cuál *es* `customerCode` es más claro que tratar de determinar qué *no* es `customerCode`.

**Figura 4-6** Uso del equivalente positivo de la comparación negativa de la figura 4-5

Aunque usar comparaciones negativas puede ser incómodo, su significado en ocasiones es más claro cuando se usan. Con frecuencia, esto ocurre cuando usa un `if` sin un `else`, emprendiendo la acción sólo cuando alguna comparación es falsa. Un ejemplo sería: `if customerZipCode <> LOCAL_ZIP_CODE then total = total + deliveryCharge`.

Evitar un error común con los operadores relacionales

Un error común que ocurre cuando se programa con operadores relacionales es usar uno equivocado y perder la frontera o límite requerido para una selección. Si usted usa el símbolo `>` para hacer una selección cuando debió usar `>=`, todos los casos que son iguales quedarían sin seleccionar. Por desgracia, las personas que solicitan programas no siempre hablan con tanta precisión como una computadora. Si, por ejemplo, su jefe dice: “Escriba un programa que seleccione a todos los empleados mayores de 65 años”, ¿quiere decir que incluya a los empleados que tienen 65 años o no? En otras palabras, la comparación es `age > 65` o `age >= 65`? Aunque la frase *más de 65* indica *mayor que 65*, las personas no siempre dicen lo que quieren decir y el mejor curso de acción es verificar dos veces el significado que se pretende con la persona que solicitó el programa; por ejemplo, el usuario final, su supervisor o su instructor. Frases similares que pueden causar malentendidos son *no más que*, *al menos* y *no menos*.

DOS VERDADES Y UNA MENTIRA

Uso de los operadores relacionales de comparación

1. Por lo general, usted sólo puede comparar valores que sean del mismo tipo de datos.
2. Una expresión booleana se define como aquella que decide si dos valores son iguales.
3. En cualquier expresión booleana, los dos valores comparados pueden ser variables o constantes.

La afirmación falsa es la número 2. Aunque decidir si dos valores son iguales es una expresión booleana, también lo es decidir si una es mayor que o menor que.

Comprensión de la lógica AND

A menudo usted necesita evaluar más de una expresión para determinar si debe tener lugar una acción. Cuando hace preguntas múltiples antes de que se determine un resultado crea una **condición compuesta**. Por ejemplo, suponga que trabaja para una compañía de telefonía celular que cobra a sus clientes como sigue:

- La factura mensual por el servicio básico es \$30 (las cantidades están en dólares estadounidenses).
- Se facturan \$20 adicionales a los clientes que hacen más de 100 llamadas que duran un total de más de 500 minutos.

La lógica necesaria para este programa de facturación incluye una **decisión AND**, en la que dos condiciones deben ser verdaderas para que una acción ocurra. En este caso es preciso hacer

un número mínimo de llamadas y consumir un número mínimo de minutos antes de que se cargue al cliente la cantidad adicional. Una decisión AND puede construirse al usar una **decisión anidada**, o un **if anidado**; es decir, una decisión dentro de la cláusula **if-then** o **else** de otra decisión. Una serie de declaraciones **if** anidadas también se llama **declaración if en cascada**. El diagrama de flujo y seudocódigo para el programa que determina los cargos para los clientes se muestran en la figura 4-7.

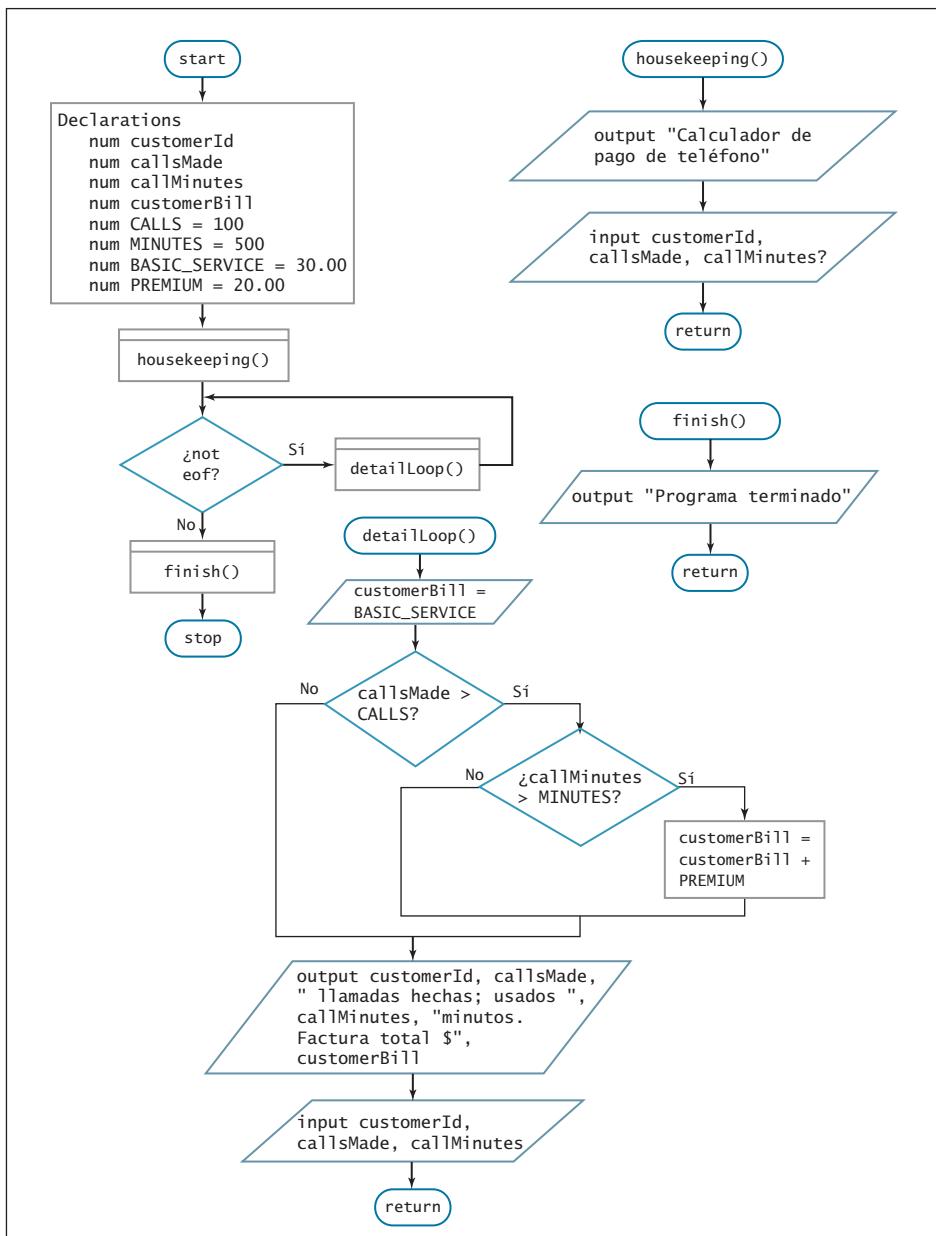


Figura 4-7 Diagrama de flujo y seudocódigo para el programa de facturación de telefonía celular (continúa)

(continuación)

```

start
Declarations
    num customerId
    num callsMade
    num callMinutes
    num customerBill
    num CALLS = 100
    num MINUTES = 500
    num BASIC_SERVICE = 30.00
    num PREMIUM = 20.00
housekeeping()
while not eof
    detailLoop()
endwhile
finish()
stop

housekeeping()
    output "Calculador de pago de teléfono"
    input customerId, callsMade, callMinutes
return

detailLoop()
    customerBill = BASIC_SERVICE
    if callsMade > CALLS then
        if callMinutes > MINUTES then
            customerBill = customerBill + PREMIUM
        endif
    endif
    output customerId, callsMade, " llamadas hechas; usados ",
        callMinutes, " minutos. Factor total $", customerBill
    input customerId, callsMade, callMinutes
return

finish()
    output "Programa terminado"
return

```

Figura 4-7 Diagrama de flujo y seudocódigo para el programa de facturación de telefonía celular

Usted aprendió sobre las estructuras anidadas en el capítulo 3. Siempre puede apilar y anidar cualquiera de las estructuras básicas.



En el programa de facturación de telefonía celular, los datos del cliente se recuperan de un archivo. Esto elimina la necesidad de indicadores y mantiene el programa más breve de modo que usted pueda concentrarse en el proceso de la toma de decisiones. Si éste fuera un programa interactivo, usted usaría un indicador antes de cada declaración de entrada. El capítulo 7 cubre el procesamiento de archivos y explica algunos pasos adicionales que puede dar cuando trabaje con archivos.

En la figura 4-7 se declaran las variables y constantes apropiadas, y luego el módulo `housekeeping()` despliega un encabezado introductorio y obtiene el primer conjunto de datos de entrada. Después de que el control regresa a la lógica de línea principal, se prueba la condición `eof` y si la entrada de datos no está completa se ejecuta el módulo `detailLoop()`. En el módulo `detailLoop()`, la factura del cliente se establece con la tarifa estándar y luego se ejecuta la decisión anidada. En la estructura `if` anidada en la figura 4-7 se evalúa primero la expresión `callsMade > CALLS?` Si la expresión es verdadera, sólo entonces se evalúa la segunda expresión booleana (`callMinutes > MINUTES?`). Si esta expresión también es verdadera, entonces se agrega la prima de \$20 a la factura del cliente. Si cualquiera de las condiciones probadas es falsa, el valor de la factura del cliente nunca se altera, conservando el valor de \$30 que se asignó al inicio.



La mayoría de los lenguajes permite usar una variación de la estructura de decisión llamada *estructura case* cuando se deba anidar una serie de decisiones sobre una sola variable. El apéndice F contiene información sobre la estructura *case*.

Anidar decisiones AND para la eficiencia

Cuando usted anida dos decisiones debe elegir cuál de ellas tomar primero. Lógicamente, cualquier expresión en una decisión AND puede evaluarse primero. Sin embargo, con frecuencia usted puede mejorar el desempeño de su programa eligiendo en forma correcta cuál de las dos selecciones hacer primero.

Por ejemplo, la figura 4-8 muestra dos maneras de diseñar la estructura de decisión anidada que asigna una prima a las facturas de los clientes si hacen más de 100 llamadas de teléfono celular y usan más de 500 minutos en un periodo de facturación. El programa puede preguntar primero sobre las llamadas hechas, eliminar a los clientes que no hayan hablado más del mínimo y preguntar sobre los minutos que se usaron sólo para los clientes que pasen la prueba de las llamadas mínimas (es decir, que se evalúen como verdaderos). O el programa podría preguntar primero sobre los minutos, eliminar a quienes no califiquen y preguntar sobre el número de llamadas sólo para los clientes que pasen la prueba de los minutos. De cualquier manera, sólo los clientes que excedan ambos límites deben pagar la prima. ¿Hay alguna diferencia dependiendo de cuál pregunta se haga primero? En lo que respecta al resultado, no. De cualquier manera, los mismos clientes pagan la prima: los que califican con base en ambos criterios. Sin embargo, en lo que respecta a la eficiencia del programa, *sí podría* haber una diferencia dependiendo de cuál pregunta se haga primero.

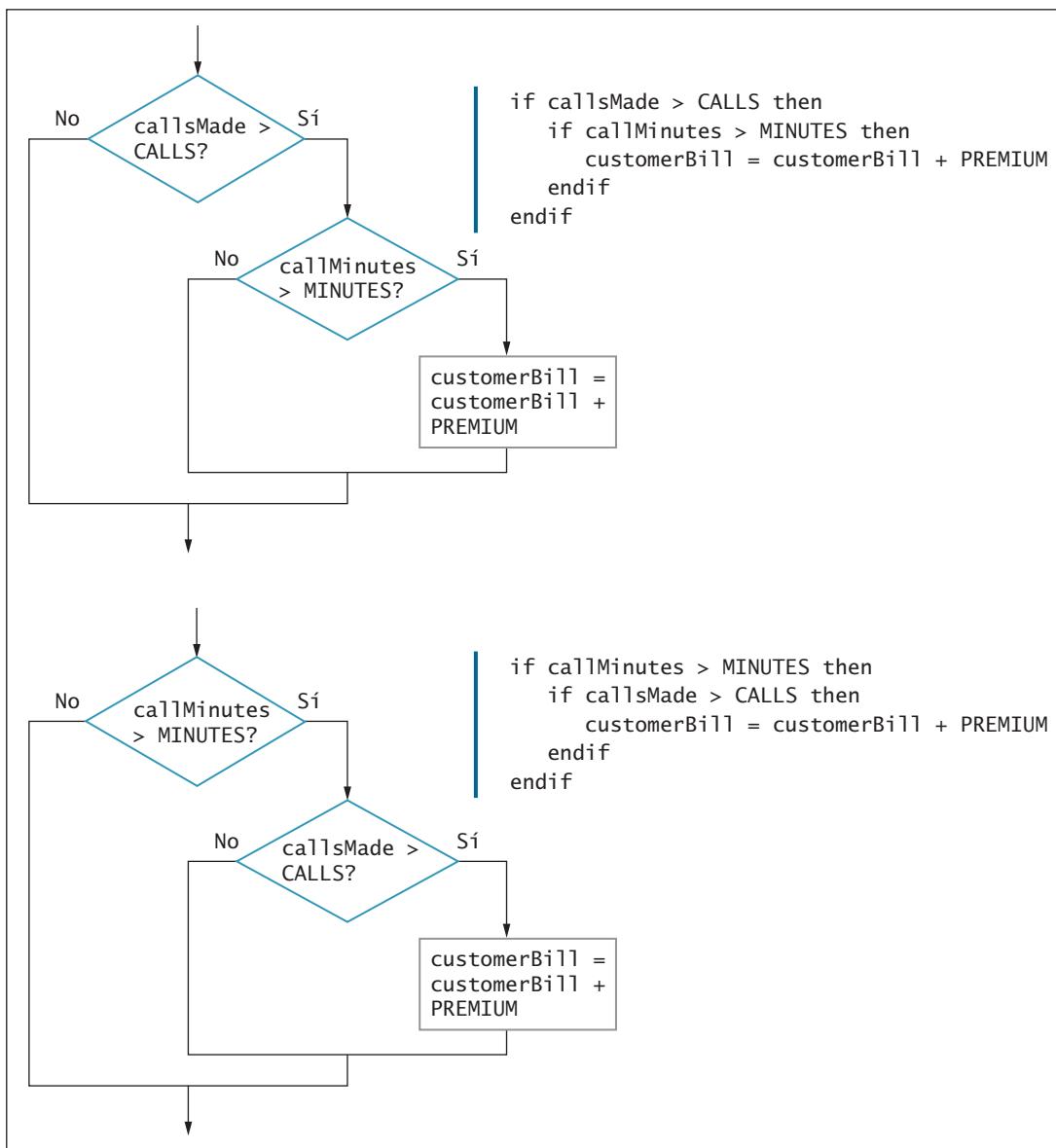


Figura 4-8 Dos formas de generar facturas de telefonía celular usando criterios idénticos

Suponga que sabe que de 1,000 clientes de telefonía celular, casi 90%, o 900, hacen más de 100 llamadas en un periodo de facturación. Imagine que sabe también que sólo alrededor de la mitad de los 1,000 clientes, o 500, usan más de 500 minutos de tiempo de llamada.

Si usa la lógica que se mostró primero en la figura 4-8 y necesita generar 100 facturas telefónicas, la primera pregunta, `callsMade > CALLS?`, se ejecutará 1,000 veces. Para aproximadamente 90% de los clientes, o 900, la respuesta es verdadera, así que se eliminan 100 clientes de

la asignación de prima y 900 continúan a la siguiente pregunta sobre los minutos consumidos. Sólo alrededor de la mitad de los clientes usan más de 500 minutos, así que 450 de los 900 pagan la prima, y se requieren 1,900 preguntas para identificarlos.

Usando la lógica alterna que se muestra en segundo lugar en la figura 4-8, la primera pregunta, `callMinutes > MINUTES?`, también se hará 1,000 veces, una vez para cada cliente. Debido a que sólo alrededor de la mitad de los clientes usan el número mayor de minutos, sólo 500 pasarán esta prueba y continuarán a la pregunta del número de llamadas realizadas. Entonces, casi 90% de los 500, o 450 clientes, pasarán la segunda prueba y se les facturará la cantidad de prima. Se requieren 1,500 preguntas para identificar a los 450 clientes que lo harán.

Ya sea que usted use el primero o el segundo órdenes de decisión de la figura 4-8, los mismos 450 clientes que satisfarán ambos criterios pagarán la prima. La diferencia es que cuando pregunta primero sobre el número de llamadas, el programa debe hacer 400 preguntas más que cuando pregunta primero sobre los minutos consumidos.

La diferencia de 400 preguntas entre el primer y el segundo conjuntos de decisiones no dura mucho tiempo en la mayoría de las computadoras. Pero toma *algún* tiempo, y si una corporación tiene cientos de miles de clientes en lugar de sólo 1,000, o si muchas de tales decisiones tienen que hacerse dentro de un programa, el tiempo de ejecución puede mejorar de manera significativa al hacer las preguntas en el orden más eficiente.

Con frecuencia cuando usted debe tomar decisiones anidadas, no tiene idea de cuál evento tiene mayores probabilidades de ocurrir; en este caso, de manera legítima puede hacer cualquier pregunta primero. Sin embargo, si conoce las probabilidades de las condiciones, o puede hacer una suposición razonable, la regla general es: *en una decisión AND, primero haga la pregunta que tiene menores probabilidades de ser verdadera*. Esto elimina la mayor cantidad posible de casos de la segunda decisión, lo que acelera el tiempo de procesamiento.

Uso del operador AND

La mayoría de los lenguajes de programación permiten hacer dos o más preguntas en una sola comparación si se usa un **operador condicional AND** o, más sencillo, un **operador AND** que une las decisiones en una sola declaración. Por ejemplo, si desea facturar una cantidad adicional a los clientes de telefonía celular que hacen más de 100 llamadas que den un total de más de 500 minutos en un periodo de facturación, puede usar decisiones anidadas, como se mostró en la sección anterior, o incluir ambas decisiones en una sola declaración escribiendo la siguiente pregunta:

```
callsMade > CALLS AND callMinutes > MINUTES?
```

Cuando usted usa uno o más operadores AND para combinar dos o más expresiones booleanas, cada una de estas últimas debe ser verdadera para que la expresión entera se evalúe como verdadera. Por ejemplo, si pregunta “¿Es usted un ciudadano que nació en Estados Unidos y tiene menos de 35 años de edad?”, la respuesta a ambas partes de la pregunta deben ser *sí* antes de que la respuesta pueda ser un solo *sí* que los resuma. Si cualquier parte de la expresión es falsa, entonces la expresión entera es falsa.



El operador condicional AND en Java, C++ y C# consiste en dos símbolos &, sin espacios entre ellos (&&). En Visual Basic, se usa la palabra And.

Una herramienta que puede ayudarle a entender el operador AND es una tabla de verdad. Las **tablas de verdad** son diagramas que se usan en las matemáticas y la lógica para describir la verdad de una expresión entera con base en la verdad de sus partes. El cuadro 4-2 muestra una tabla de verdad que lista todas las posibilidades con una decisión AND. Como muestra el cuadro, para cualesquiera dos expresiones x y y , la expresión x AND y ? es verdadera sólo si tanto x como y son verdaderas de manera individual. Si ya sea x o y sola es falsa, o si ambas son falsas, entonces la expresión x AND y ? es falsa.

<i>¿x?</i>	<i>¿y?</i>	<i>¿x AND y?</i>
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Falso
Falso	Verdadero	Falso
Falso	Falso	Falso

Cuadro 4-2 Tabla de verdad para el operador AND

Si el lenguaje de programación que usted usa permite un operador AND, debe observar que la pregunta que coloque primero (a la izquierda del operador) es la que se hará primero y los casos que se eliminan con base en la primera pregunta no procederán a la segunda. En otras palabras, cada parte de una expresión que usa un operador AND sólo se evalúa hasta donde sea necesario para determinar si la expresión entera es verdadera o falsa. Esta característica se llama **evaluación de cortocircuito**. La computadora sólo puede hacer una pregunta a la vez; aun cuando su seudocódigo se vea como el primer ejemplo en la figura 4-9, la computadora ejecutará la lógica que se muestra en el segundo ejemplo. Aun cuando use un operador AND, la computadora toma una decisión a la vez y lo hace en el orden en que usted las hizo. Si la primera pregunta en una expresión AND se evalúa como falsa, entonces la expresión entera es falsa, y la segunda pregunta nunca se prueba.

Nunca se requiere que use el operador AND debido a que el uso de declaraciones *if* anidadas siempre puede lograr el mismo resultado. Sin embargo, usar el operador AND con frecuencia hace que su código sea más conciso, menos propenso a errores y más fácil de entender.

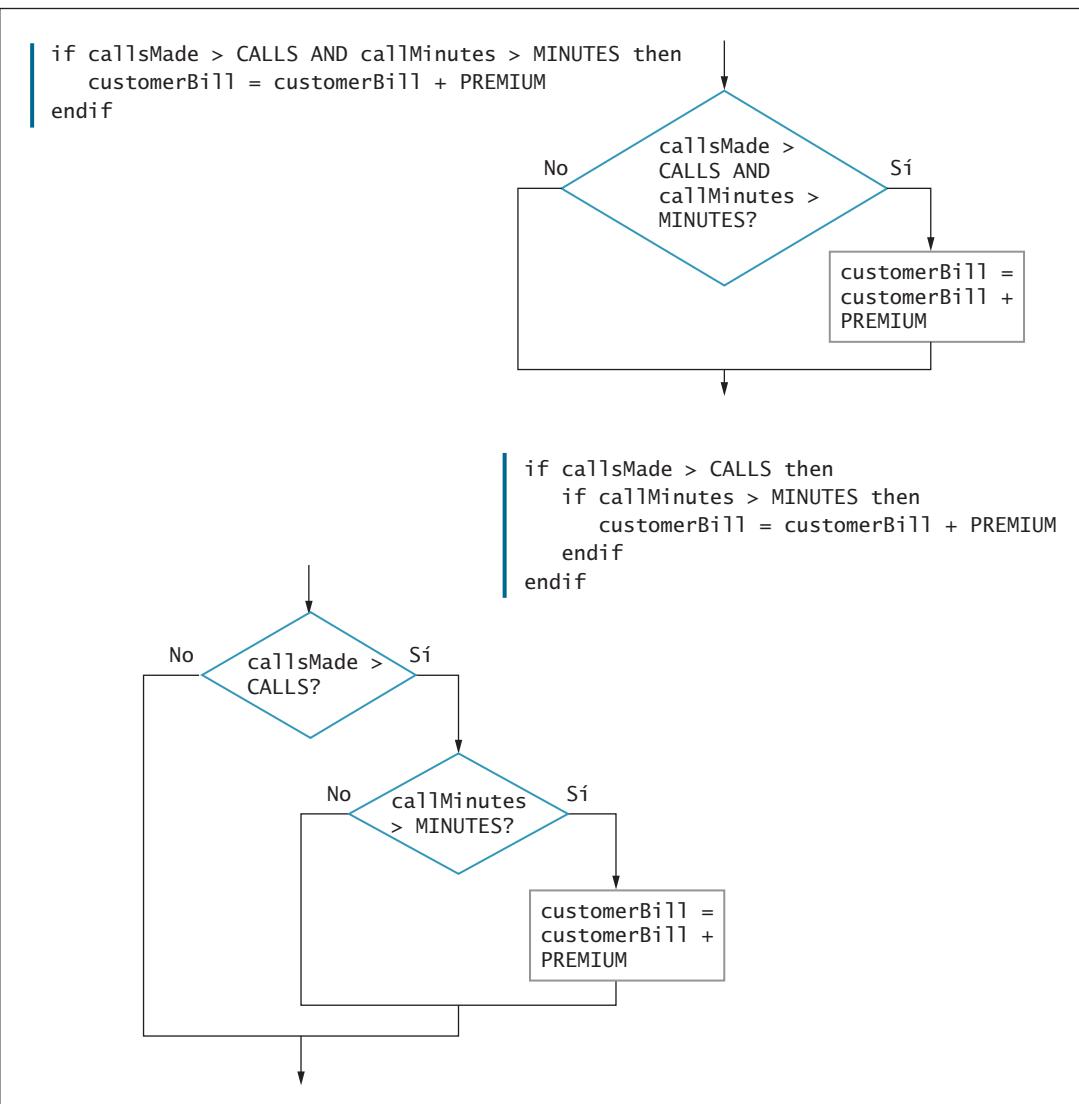


Figura 4-9 Uso de un operador AND y la lógica que hay detrás de él

Ejitar errores comunes en una selección AND

Cuando necesite satisfacer dos o más criterios para iniciar un evento en un programa, debe asegurarse de que la segunda decisión se hace por completo dentro de la primera decisión. Por ejemplo, si el objetivo de un programa es agregar una prima de \$20 a la factura de los clientes de telefonía celular que excedan de 100 llamadas y de 500 minutos en un periodo de facturación, entonces el segmento de programa que se muestra en la figura 4-10 contiene tres tipos diferentes de errores lógicos.

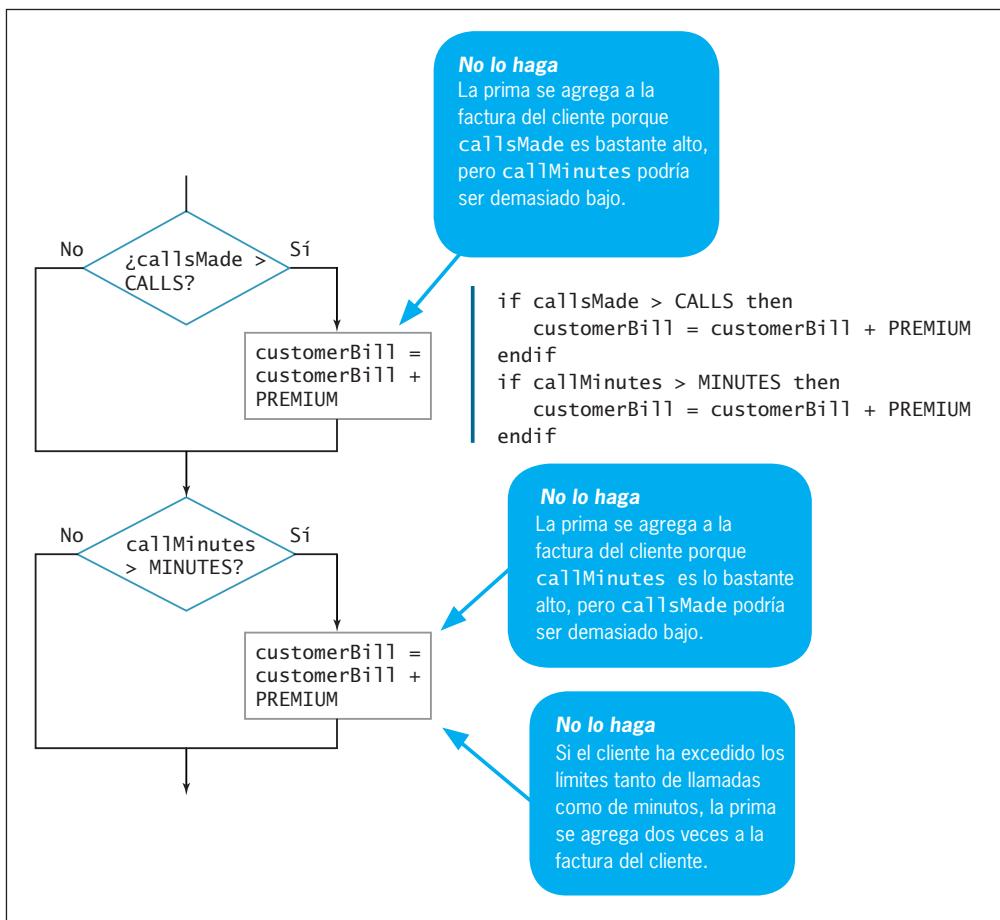


Figura 4-10 Lógica incorrecta para agregar una prima de \$20 a las facturas de clientes de telefonía celular que cumplan dos criterios

La lógica en la figura 4-10 muestra que se agregan \$20 a la factura de un cliente que hace demasiadas llamadas; no necesariamente debería facturársele extra, sus minutos podrían estar debajo del límite para la prima de \$20. Además, un cliente que ha hecho pocas llamadas no se elimina de la segunda pregunta. En cambio, a todos los clientes se les aplica la pregunta de los minutos y a algunos se les asigna la prima aun cuando no hayan pasado el criterio del número de llamadas realizadas. Además, a cualquier cliente que pase ambas pruebas se le agregará la prima dos veces a su factura. Por muchas razones, la lógica que se muestra en la figura 4-10 *no* es correcta para este problema.

Cuando use el operador AND en la mayoría de los lenguajes debe proporcionar una expresión booleana completa en cada lado del operador. En otras palabras, `callMinutes > 100 AND callMinutes < 200` sería una expresión válida para encontrar `callMinutes` entre 100 y 200. Sin embargo, `callMinutes > 100 AND < 200` no sería válida porque lo que sigue al operador AND (`< 200`) no es una expresión booleana completa.

Por claridad, puede rodear cada expresión booleana en una expresión compuesta con su propio conjunto de paréntesis. Use este formato si es más claro para usted; por ejemplo, podría escribir lo siguiente:

```
if (callMinutes > MINUTES) AND (callsMade > CALLS)
    customerBill = customerBill + PREMIUM
endif
```

138

DOS VERDADES Y UNA MENTIRA

Comprensión de la lógica AND

1. Cuando usted anida decisiones debido a que la acción resultante requiere que dos condiciones sean verdaderas, cualquier decisión puede hacerse lógicamente primero y ocurrirán las mismas selecciones.
2. Cuando se requieren dos selecciones para que una acción ocurra, con frecuencia usted puede mejorar el desempeño de su programa eligiendo de manera apropiada cuál selección hacer primero.
3. Para mejorar la eficiencia en una selección anidada en la que dos condiciones deben ser verdaderas para que ocurra alguna acción, primero debería hacer la pregunta que tiene más probabilidades de ser verdadera.

La afirmación falsa es la número 3. Para la eficiencia en una selección anidada, usted debería hacer primero la pregunta que tiene menos probabilidades de ser verdadera.

Comprensión de la lógica OR

En ocasiones usted desea que ocurra una acción cuando una *u* otra de dos condiciones sea verdadera. Esto se llama **decisión OR** porque una condición *o* alguna otra condición deben cumplirse para que un evento se presente. Si alguien pregunta: “¿Estás libre para cenar el viernes o el sábado?”, sólo una de las dos condiciones tiene que ser verdadera para que la respuesta a la pregunta completa sea *sí*; sólo si las respuestas a ambas mitades de la pregunta son falsas el valor de la expresión entera es falso.

Por ejemplo, suponga que desea agregar \$20 a las facturas de los clientes de telefonía celular que hagan más de 100 llamadas o usen más de 500 minutos. La figura 4-11 muestra el módulo `detailLoop()` alterado del programa de facturación que logra este objetivo.

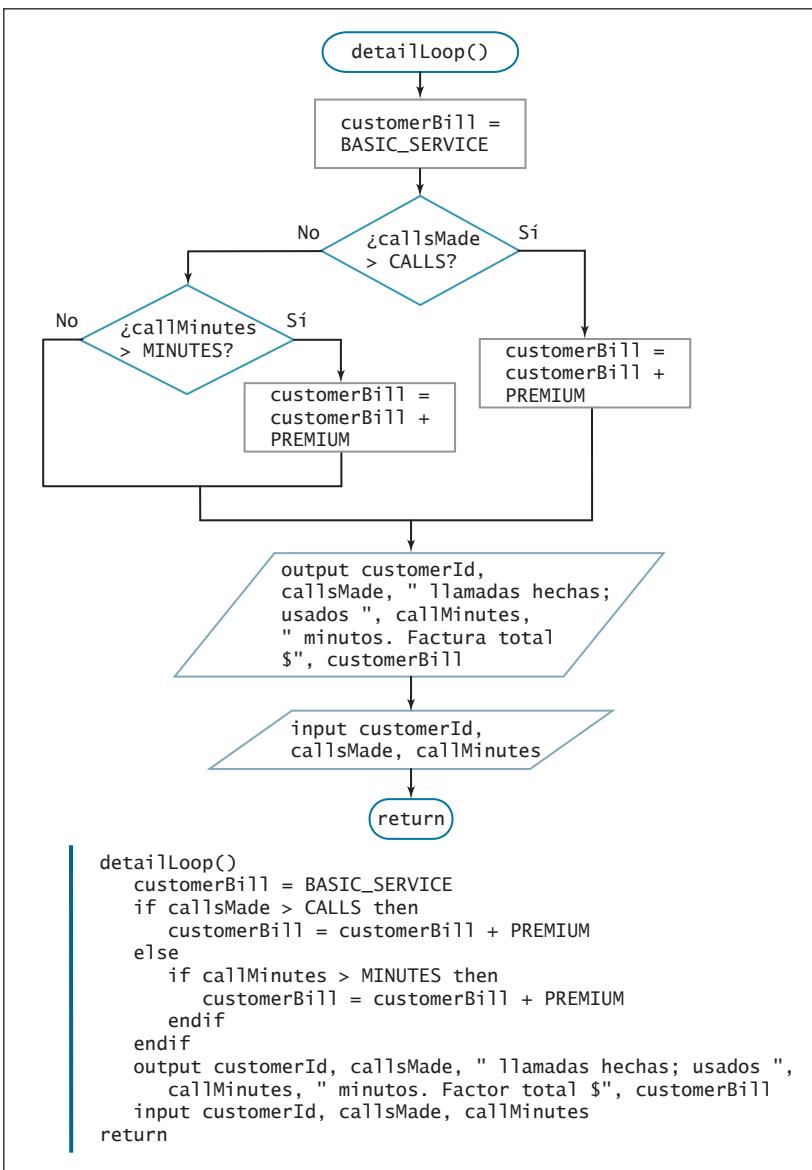


Figura 4-11 Diagrama de flujo y seudocódigo para el programa de facturación de telefonía celular en el que un cliente debe cumplir uno o los dos criterios para que se le facture una prima

El `detailLoop()` en el programa de la figura 4-11 hace la pregunta `¿callsMade > CALLS?`, y si el resultado es verdadero, la cantidad extra se agrega a la factura del cliente. Debido a que hacer muchas llamadas es suficiente para que el cliente incurra en la prima, no hay necesidad de cuestionar más. Sólo si el cliente no ha hecho más de 100 llamadas el programa necesita preguntar si `callMinutes > MINUTES` es verdadera. Si el cliente no hace más de 100 llamadas, pero no obstante usó más de 500 minutos, entonces se agrega la cantidad de la prima a la factura del cliente.

Escritura de decisiones OR para eficiencia

Como con una selección AND, cuando usa una selección OR puede elegir hacer cualquier pregunta primero. Por ejemplo, puede agregar \$20 extra a las facturas que cumplan uno o el otro de dos criterios usando la lógica en cualquier parte de la figura 4-12.

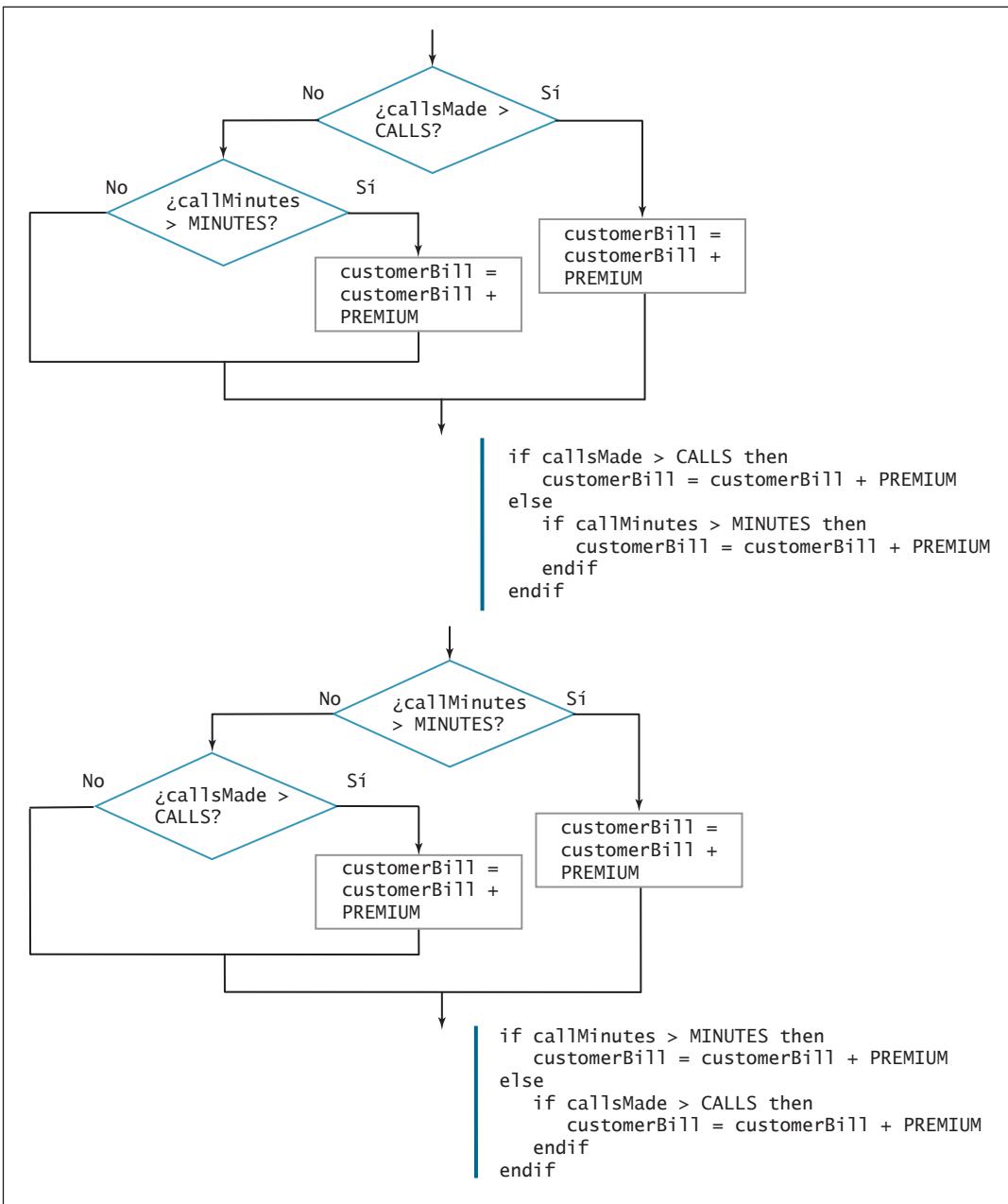


Figura 4-12 Dos formas de asignar una prima a las facturas de los clientes que cumplen uno de dos criterios

Usted pudo adivinar que una de estas selecciones es superior a la otra cuando tiene algún antecedente sobre la probabilidad relativa de cada condición que se prueba. Por ejemplo, digamos que sabe que de 1,000 clientes de telefonía celular, casi 90%, o 900, hacen más de 100 llamadas en un periodo de facturación. También sabe que sólo alrededor de la mitad de los 1,000 clientes, o 500, usan más de 500 minutos de tiempo de llamada.

Cuando usted usa la lógica que se muestra en la primera mitad de la figura 4-12, primero pregunta respecto a las llamadas realizadas. Para 900 clientes la respuesta es verdadera y usted agrega la prima a sus facturas. Sólo alrededor de 100 conjuntos de datos de los clientes continúan a la siguiente pregunta respecto a los minutos de llamada, donde a alrededor de 50% de los 100, o 50, se les factura con la cantidad extra. Al final, ha tomado 1,100 decisiones para agregar en forma correcta las cantidades de la prima para 950 clientes.

Si usa la lógica OR en la segunda mitad de la figura 4-12, pregunta primero sobre los minutos usados (1,000 veces, una vez por cada uno de los 1,000 clientes). El resultado es verdadero para 50%, o 500 clientes, cuya factura se incrementa. Para los otros 500 clientes, pregunta sobre el número de llamadas hechas. Para 90% de los 500, el resultado es verdadero, así que se agregan primas para 450 personas adicionales. Al final, los mismos 950 clientes son facturados con \$20 extra, pero este enfoque requiere ejecutar 1,500 decisiones, 400 decisiones más que cuando se usó la primera lógica de decisión.

La regla general es: *en una decisión OR, primero se hace la pregunta que tiene más probabilidades de ser verdadera*. Este enfoque elimina tantas ejecuciones de la segunda decisión como sea posible y disminuye el tiempo que toma procesar todos los datos. Como con la situación AND, en una situación OR es más eficiente eliminar tantas decisiones extra como sea posible.

Uso del operador OR

Si necesita emprender una acción cuando se cumple una u otra de dos condiciones, puede usar dos estructuras de selección anidadas separadas, como en los ejemplos anteriores. Sin embargo, la mayoría de los lenguajes de programación permiten hacer dos o más preguntas en una sola comparación usando un **operador condicional OR** (o simplemente **operador OR**). Por ejemplo, usted puede hacer la siguiente pregunta:

```
?callsMade > CALLS OR ?callMinutes > MINUTES?
```

Como con el operador AND, la mayoría de los lenguajes de programación requieren una expresión booleana completa en cada lado del operador OR. Cuando usted usa el operador lógico OR, sólo una de las condiciones listadas debe cumplirse para que la acción resultante ocurra. El cuadro 4-3 muestra la tabla de verdad para el operador OR. Como puede ver en el cuadro, la expresión entera $x \text{ OR } y$ es falsa sólo cuando x y y son falsas de manera individual.

¿X?	¿Y?	¿x OR y?
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Verdadero
Falso	Verdadero	Verdadero
Falso	Falso	Falso

Cuadro 4-3 Tabla de verdad para el operador OR

Si el lenguaje de programación que usted usa soporta al operador OR, aún debe observar que la pregunta que coloque primero es la que se hará primero y los casos que pasen la prueba de la primera pregunta no procederán a la segunda. Como con el operador AND, esta característica se llama cortocircuito. La computadora sólo puede hacer una pregunta a la vez; aun cuando escriba el código como se muestra en la parte superior de la figura 4-13, la computadora ejecutará la lógica que se muestra en la parte inferior.

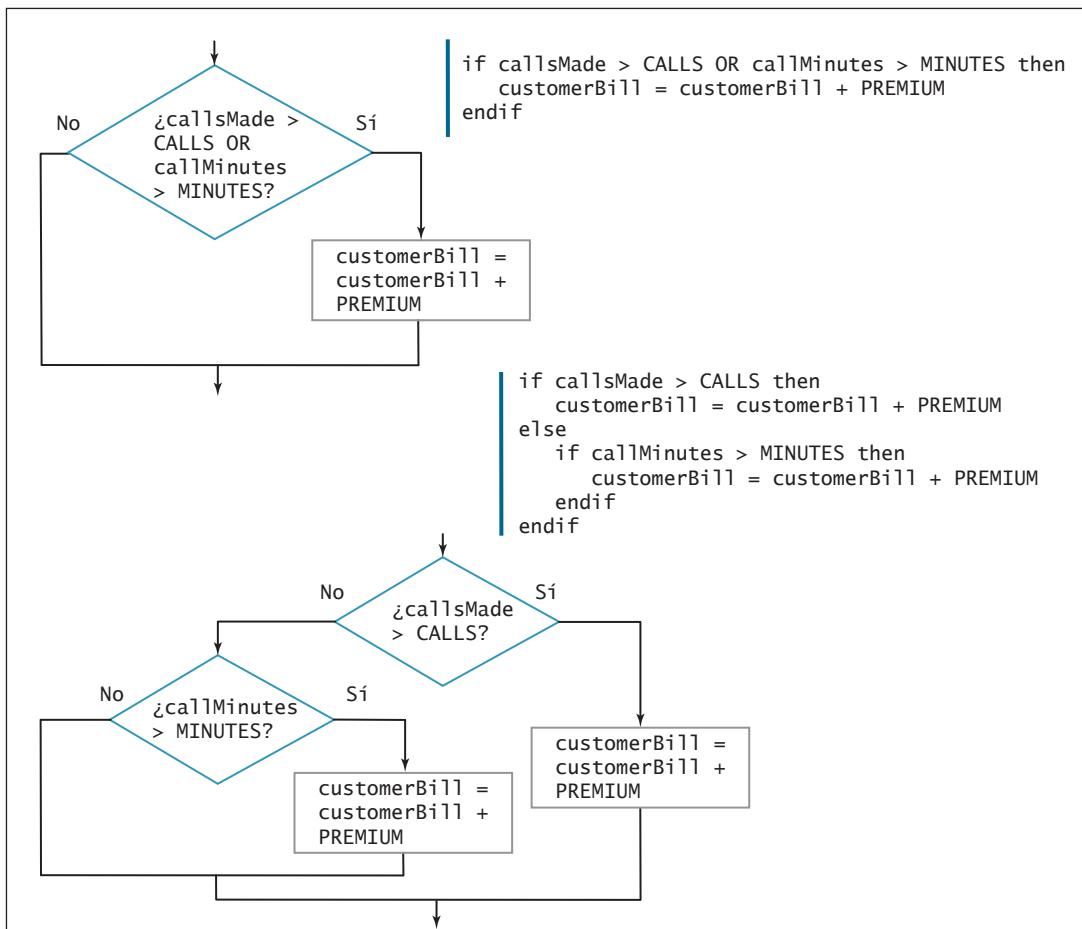


Figura 4-13 Uso de un operador OR y la lógica que hay detrás de él



C#, C++, C y Java usan el símbolo || como el operador lógico OR. En Visual Basic, el operador es Or.

Evitar errores comunes en una selección OR

Quizá usted haya notado que la declaración de asignación `customerBill = customerBill + PREMIUM` aparece dos veces en los procesos de toma de decisión en las figuras 4-12 y 4-13. Cuando se diseña un diagrama de flujo, la tentación es trazar la lógica en forma parecida a la de la figura 4-14. Usted podría afirmar que el diagrama de flujo en la figura 4-14 es correcto porque a los clientes correctos se les facturan los 20 dólares extra. Sin embargo, este diagrama de flujo no está estructurado. La segunda pregunta no es una estructura autónoma con un punto de entrada y uno de salida; en cambio, la línea de flujo sale de la estructura de selección interior para unirse al lado Sí de la estructura de selección exterior.

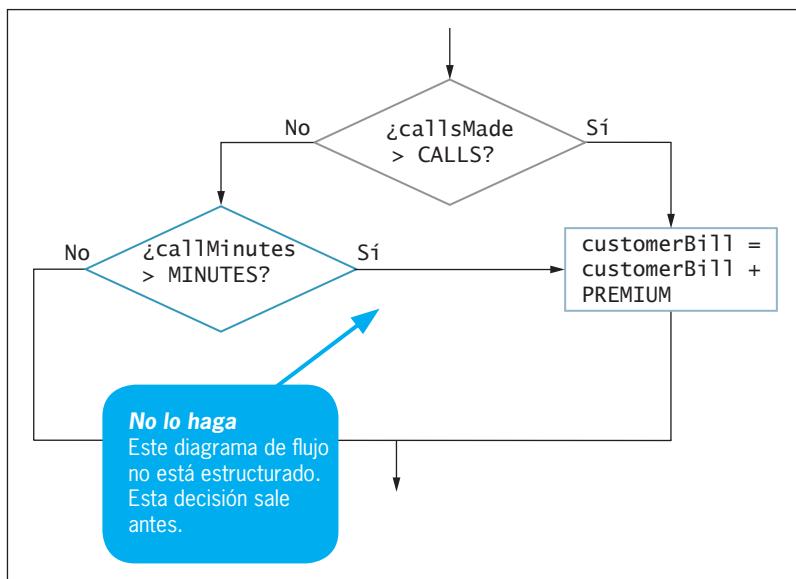


Figura 4.14 Diagrama de flujo no estructurado para determinar la factura de telefonía celular del cliente

La selección OR tiene potencial adicional para que ocurran errores debido a las diferencias en la forma en que las personas y las computadoras usan el lenguaje. Cuando su jefa desea agregar una cantidad extra a las facturas de los clientes que hacen más de 100 llamadas *o* usan más de 500 minutos, es probable que diga: “Agregue 20 dólares a la factura de cualquiera que haga más de 100 llamadas *y* a cualquiera que haya usado más de 500 minutos”. Su solicitud contiene la palabra *y* entre dos tipos de personas, las que han hecho muchas llamadas y las que usaron muchos minutos, poniendo el énfasis en las personas. Sin embargo, cada decisión que usted tome se relaciona con los 20 dólares agregados para un solo cliente que haya cumplido un criterio *o* el otro, *o* ambos. En otras palabras, la condición OR está entre los atributos de cada cliente y no entre clientes diferentes. En lugar de la declaración previa de la gerente, sería más claro si ella dijera: “Agregue 20 dólares a la factura de cualquiera que haya hecho más de 100 llamadas *o* haya usado más de 500 minutos”, pero usted no puede contar con que las personas hablen como las computadoras. Como programador, tiene la labor de aclarar lo que se solicita en realidad. Con frecuencia, una petición casual para A *y* B lógicamente significa una petición para A *o* B.

La forma en que usamos el idioma puede causar otro tipo de error cuando alguien solicita que usted encuentre si un valor cae entre otros dos valores. Por ejemplo, un gerente de cine podría decir: "Otorgue un descuento a los clientes que tengan menos de 13 años de edad y a aquellos que tengan más de 64 años de edad; de lo contrario, cobre el precio completo". Debido a que el gerente ha usado la palabra *y* en la solicitud, usted podría estar tentado a crear la decisión que se muestra en la figura 4-15; sin embargo, esta lógica no proporcionará un precio con descuento a ningún asistente al cine. Debe recordar que cada vez que se toma la decisión en la figura 4-15, es para un solo cliente del cine. Si `patronAge` contiene un valor menor que 13, entonces no es posible que contenga un valor mayor que 64. Del mismo modo, si `patronAge` contiene un valor mayor que 64, no hay forma de que pueda contener un valor menor. Por consiguiente, ningún valor podría almacenarse en `patronAge` para el que ambas partes de la pregunta AND pudiera ser verdadera, y el precio nunca se establecería con descuento para cualquier cliente. La figura 4-16 muestra la lógica correcta.

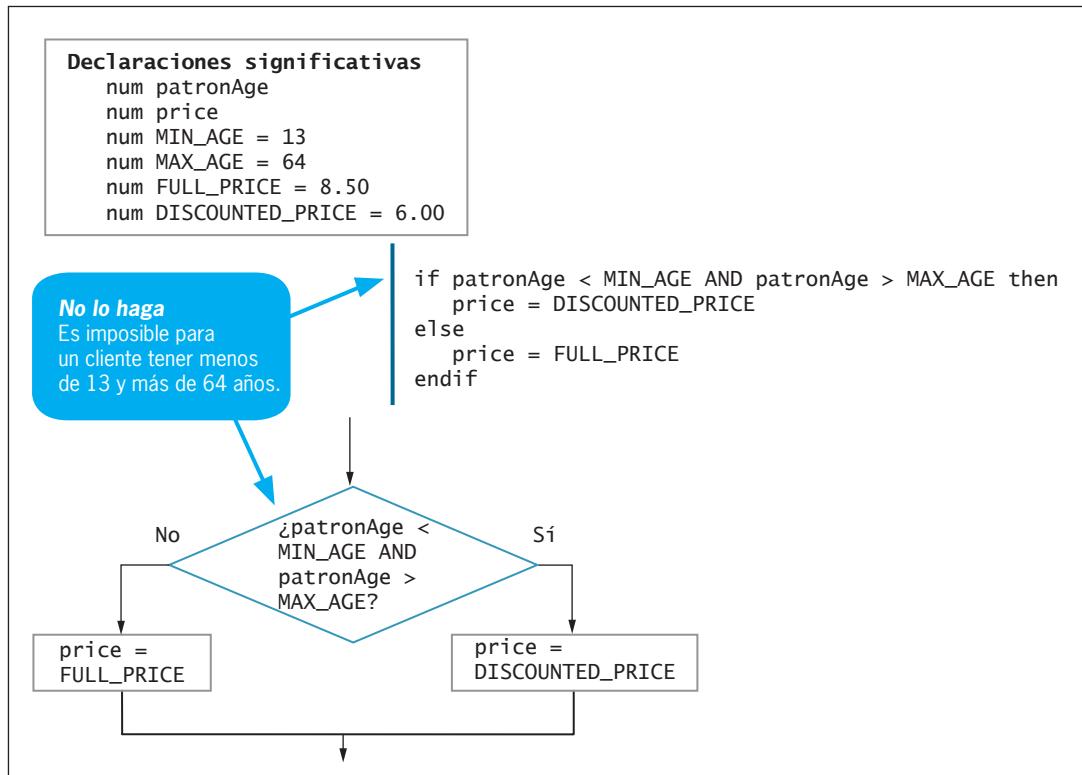


Figura 4-15 Lógica incorrecta que intenta otorgar un descuento para los clientes del cine jóvenes y adultos mayores

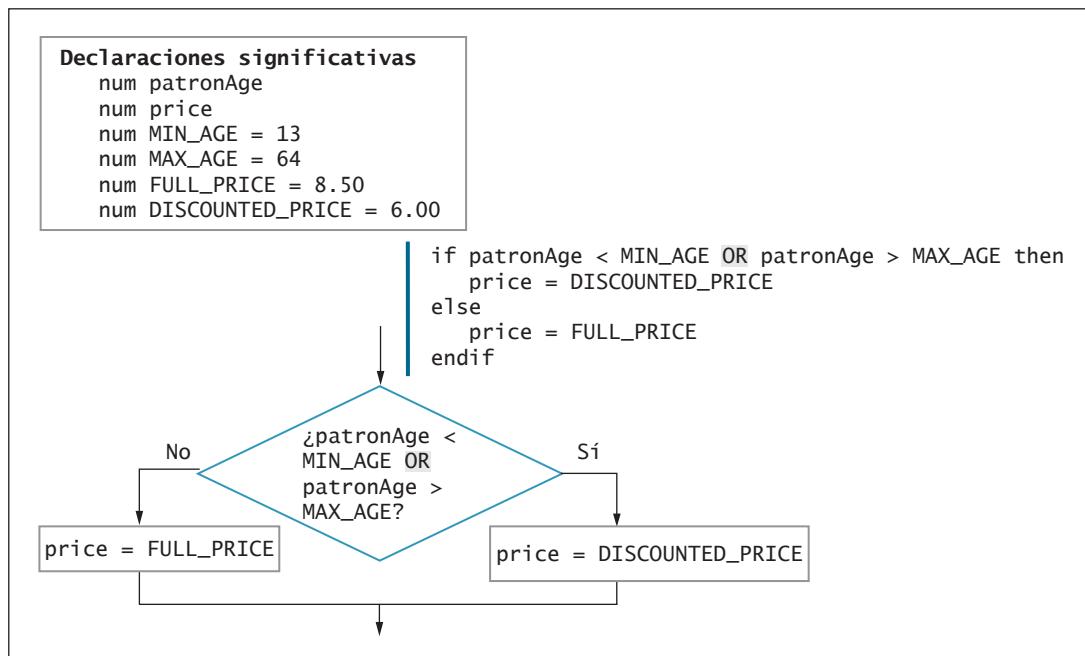


Figura 4-16 Lógica correcta que da un descuento para clientes jóvenes y adultos mayores

Un error similar puede ocurrir en su lógica si el gerente del cine dice algo como: “No otorgue un descuento, es decir, cobre el precio completo, si un cliente tiene más de 12 años o menos de 65 años de edad”. Debido a que la palabra *o* aparece en la petición, usted podría planear su lógica de manera parecida a la de la figura 4-17. Ningún cliente recibiría alguna vez un descuento, porque todos tienen más de 12 o menos de 65. Recuerde, en una decisión OR, sólo una de las condiciones necesita ser verdadera para que la expresión entera sea evaluada como verdadera. Así, por ejemplo, debido a que un cliente que tiene 10 años tiene menos de 65, se cobra el precio completo, y debido a que un cliente que tiene 70 tiene más de 12 también se cobra el precio completo. La figura 4-18 muestra la lógica correcta para esta decisión.

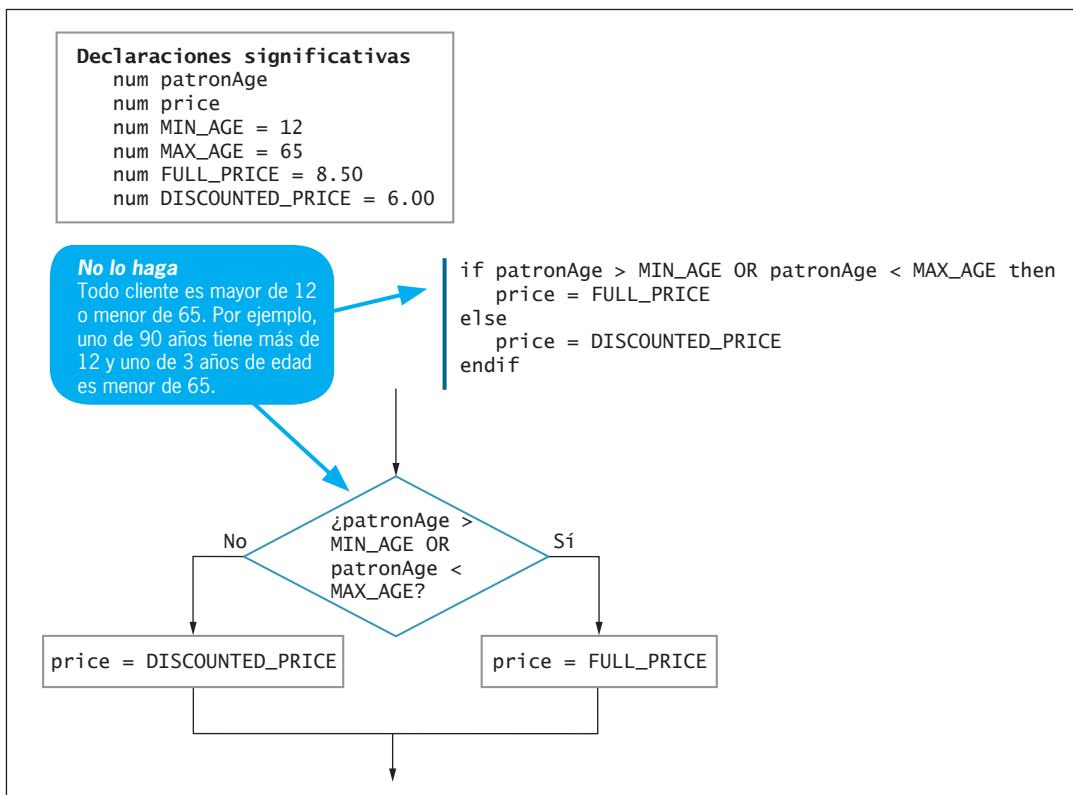


Figura 4-17 Lógica incorrecta que intenta cobrar el precio completo a los clientes cuyas edades rebasen los 12 años y sean menores de 65

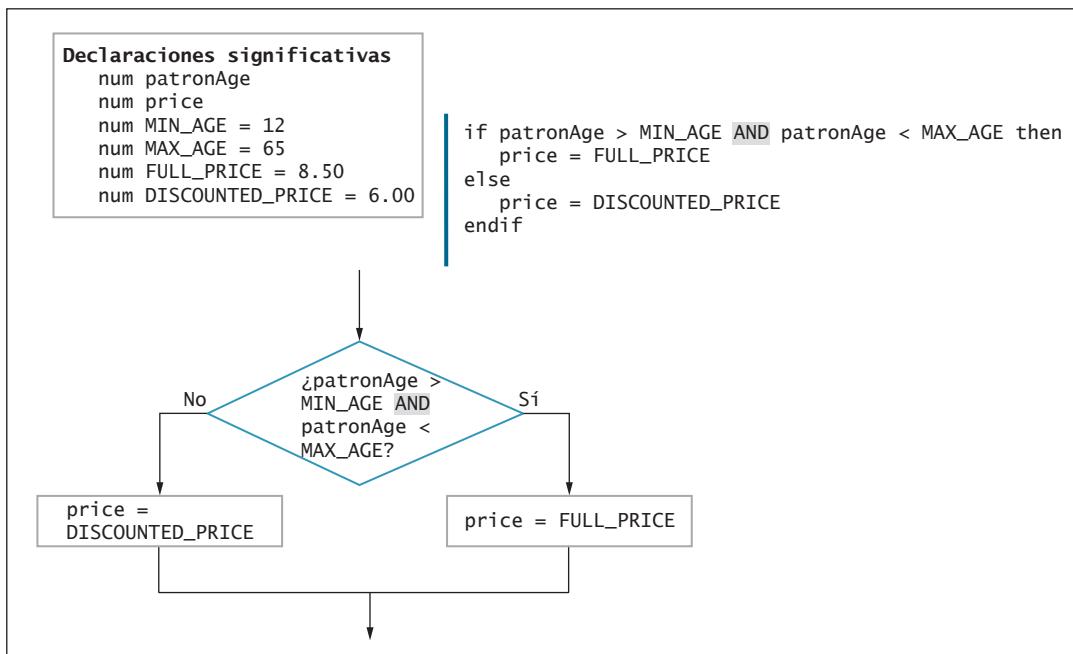


Figura 4-18 Lógica correcta que cobra el precio completo a clientes cuya edad sea mayor de 12 y menor de 65



Además de los operadores AND y OR, la mayoría de los lenguajes soporta un operador NOT. Se usa el **operador lógico NOT** para invertir el significado de una expresión booleana. Por ejemplo, la declaración `if NOT (age < 21) output "OK"` produce OK cuando age es mayor o igual que 21. El operador NOT es unario en lugar de binario; es decir, no se usa entre dos expresiones, sino que se usa frente a una sola expresión. En C++, Java y C#, se usa el signo de admiración como operador NOT. En Visual Basic, el operador es Not.

DOS VERDADES Y UNA MENTIRA

Comprensión de la lógica OR

1. En una selección OR, deben cumplirse dos o más condiciones para que un evento ocurra.
2. Cuando use una selección OR, puede hacer cualquier pregunta primero y aun lograr un programa utilizable.
3. La regla general es: en una decisión OR, primero haga la pregunta que tenga mayor probabilidad de ser verdadera.

La afirmación falsa es la número 1. En una selección OR, solo una de dos condicio- nes debe cumplirse para que un evento ocurra.

Hacer selecciones dentro de rangos

Con frecuencia usted necesita emprender una acción cuando una variable se ubica dentro de un rango de valores. Por ejemplo, suponga que su compañía proporciona varios descuentos a los clientes con base en el número de artículos ordenados, como se muestra en la figura 4-19.

Artículos ordenados	Tasa de descuento (%)
0 a 10	0
11 a 24	10
25 a 50	15
51 o más	20

Figura 4-19 Tasas de descuento basadas en artículos ordenados

Cuando usted escribe el programa que determina una tasa de descuento basada en el número de artículos, podría tomar cientos de decisiones, como `if itemQuantity = 1?`, `if itemQuantity = 2?` y así sucesivamente. Sin embargo, es más conveniente encontrar la tasa de descuento correcta usando una comprobación de rango.

Cuando usted usa una **comprobación de rango** compara una variable con una serie de valores que marcan los límites de los rangos. Para realizar una comprobación de rango, se hacen comparaciones usando el valor mínimo o máximo en cada rango de valores. Por ejemplo, para encontrar cada tasa de descuento listado en la figura 4-19, puede usar una de las siguientes técnicas:

- Hacer comparaciones usando los extremos inferiores de los rangos.
 - Usted puede preguntar: `if itemQuantity < 11`? Si no, `if itemQuantity < 25`? Si no, `if itemQuantity < 51`? (Si es posible que el valor sea negativo, también comprobaría un valor menor que 0 y emprendería la acción apropiada si lo es.)
 - Puede preguntar: `if itemQuantity >= 51`? Si no, `if itemQuantity >= 25`? Si no, `if itemQuantity >= 11`? (Si es posible que el valor sea negativo, también comprobaría un valor mayor o igual que 0 y emprendería la acción apropiada si no lo es.)
- Hacer comparaciones usando los extremos superiores de los rangos.
 - Puede preguntar: `if itemQuantity > 50`? Si no, `if itemQuantity > 24`? Si no, `if itemQuantity > 10`? (Si hay un valor máximo permitido para `itemQuantity`, también comprobaría un valor mayor que ese límite y emprendería la acción apropiada si lo es.)
 - Puede preguntar: `if itemQuantity <= 10`? Si no, `if itemQuantity <= 24`? Si no, `if itemQuantity <= 50`? (Si hay un valor máximo permitido para `itemQuantity`, también comprobaría un valor menor o igual que ese límite y emprendería la acción apropiada si no lo es.)

La figura 4-20 muestra el diagrama de flujo y el seudocódigo que representan la lógica para un programa que determina el descuento correcto para cada cantidad ordenada. En el proceso de toma de decisión, `itemsOrdered` se compara con el extremo superior del grupo con rango menor (`RANGE1`). Si `itemsOrdered` es menor o igual que ese valor, entonces usted conoce el descuento correcto, `DISCOUNT1`; si no, continúa comprobando. Si `itemsOrdered` es menor o igual que el extremo superior del siguiente rango (`RANGE2`), entonces el descuento para el cliente es `DISCOUNT2`; si no, continúa comprobando, y al final el descuento del cliente se establece en `DISCOUNT3` o `DISCOUNT4`. En el seudocódigo en la figura 4-20, note cómo cada `if`, `else` y `endif` asociados se alinean de manera vertical.

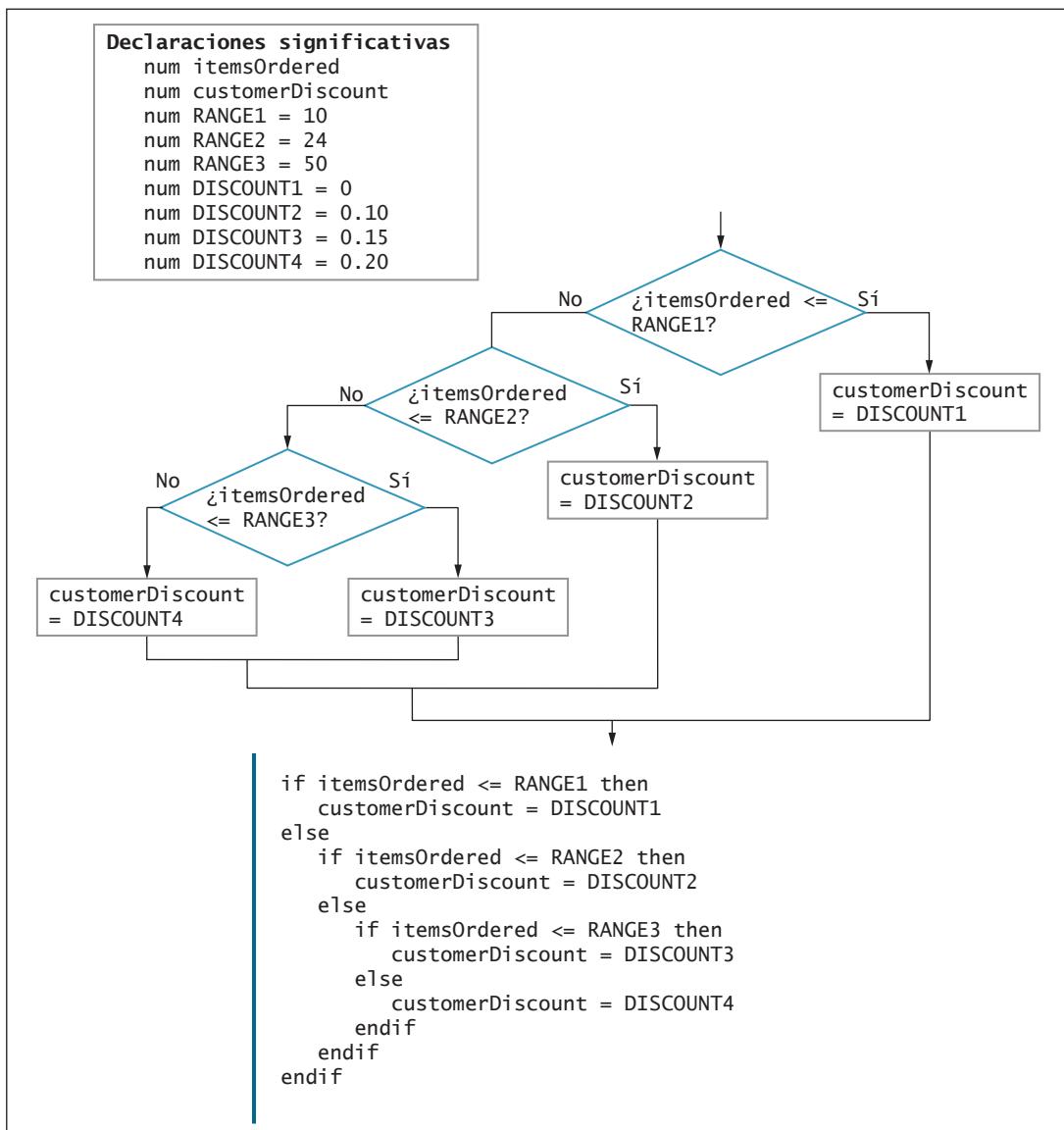


Figura 4-20 Diagrama de flujo y seudocódigo de lógica que selecciona el descuento correcto basado en artículos



En la memoria de la computadora, un signo de porcentaje (%) no se almacena con un valor numérico que represente un porcentaje. En cambio, se almacena el equivalente matemático. Por ejemplo, 15% se almacena como 0.15.

Por ejemplo, considere una orden para 30 artículos. La expresión `itemsOrdered <= RANGE1` se evalúa como falsa, así que se ejecuta la cláusula `else` de la decisión. Ahí, `itemsOrdered <= RANGE2` también se evalúa como falsa, así que se ejecuta la cláusula `else`. La expresión `itemsOrdered <= RANGE3` es verdadera, así que `customerDiscount` se vuelve `DISCOUNT3`, que es 0.15. Recorra la lógica con otros valores para `itemsOrdered` y verifique por sí mismo que cada vez se aplica el descuento correcto.

Evitar errores comunes cuando se usan comprobaciones de rango

Cuando los programadores inexpertos realizan comprobaciones de rango son propensos a incluir la lógica que tiene demasiadas decisiones, lo que conlleva más trabajo del necesario.

La figura 4-21 muestra un segmento de programa que contiene una comprobación de rango en la que el programador hace una pregunta de más (la pregunta sombreada en la figura). Si sabe que `itemsOrdered` no es menor o igual que `RANGE1`, no menor o igual que `RANGE2` y no menor o igual que `RANGE3`, entonces `itemsOrdered` debe ser mayor que `RANGE3`. Preguntar si `itemsOrdered` es mayor que `RANGE3` es un desperdicio de tiempo; ningún pedido de los clientes puede recorrer alguna vez la ruta lógica en la extrema izquierda del diagrama de flujo. Podría decir que dicha ruta es una **ruta sin salida** o **inalcanzable**, y que las declaraciones escritas ahí constituyen un código sin salida o inalcanzable. Aunque un programa que contiene dicha lógica se ejecutará y asignará el descuento correcto a los clientes que ordenen más de 50 artículos, proveer esta ruta es ineficiente.

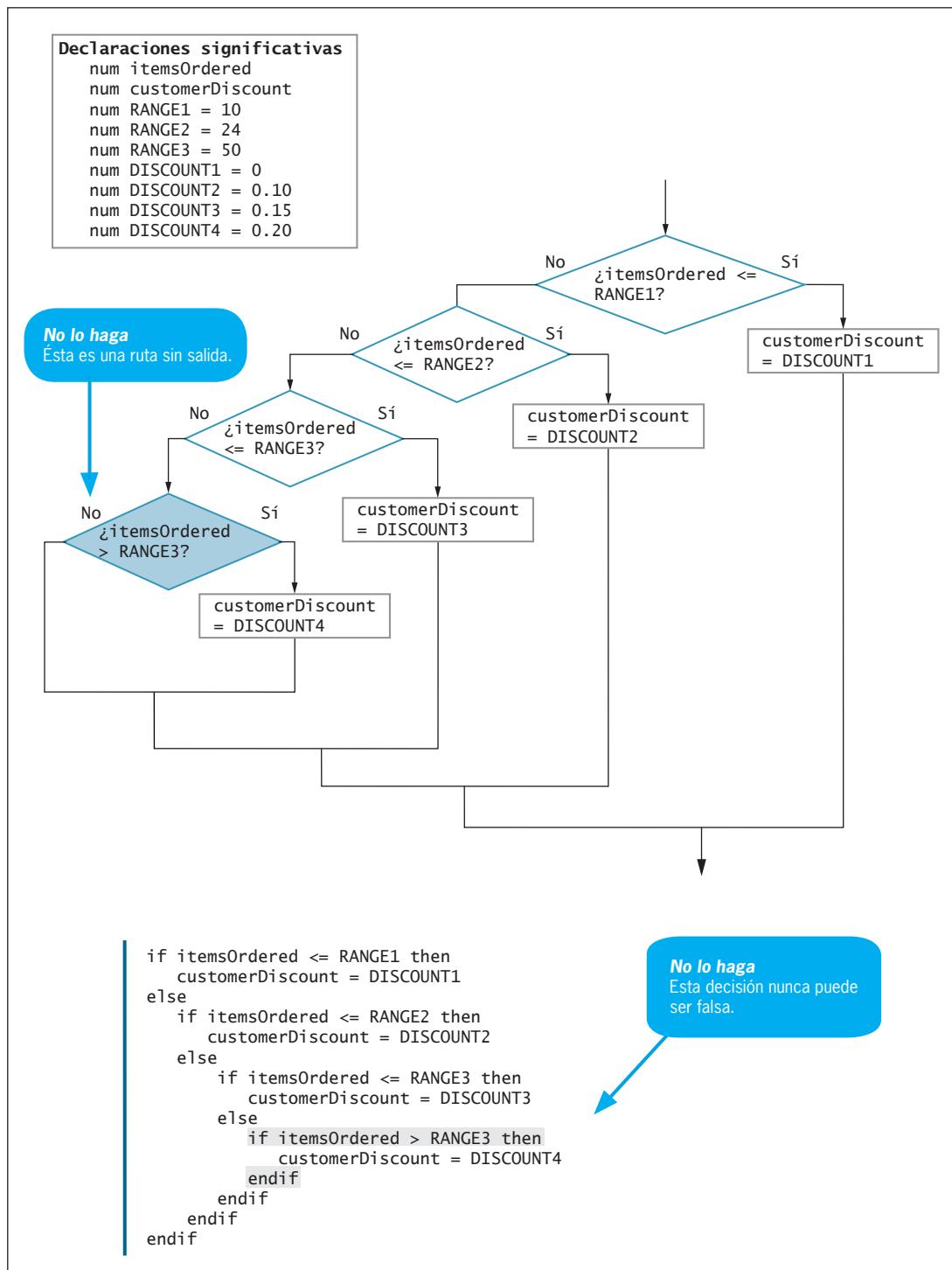


Figura 4-21 Selección de rango ineficiente que incluye una ruta inalcanzable

En la figura 4-21, es más fácil ver la inutilidad de la ruta en el diagrama de flujo que en la representación en seudocódigo de la misma lógica. Sin embargo, cuando usa un `if` sin un `else`, no hace nada cuando la respuesta a la pregunta es falsa.



Cuando usted hace preguntas de seres humanos, en ocasiones ya sabe las respuestas. Por ejemplo, un buen abogado litigante rara vez hace una pregunta en el tribunal si la respuesta será una sorpresa. Sin embargo, con la lógica de la computadora, tales preguntas son un ineficiente desperdicio de tiempo.

152

Otro error que cometen los programadores cuando escriben la lógica para ejecutar una comprobación de rango también implica hacer preguntas innecesarias. Usted nunca debería hacer una pregunta si sólo hay una respuesta o un resultado posible. La figura 4-22 muestra una selección de rango ineficiente que hace dos preguntas innecesarias. En la figura, si `itemsOrdered` es menor o igual que `RANGE1`, `customerDiscount` se establece en `DISCOUNT1`. Si `itemsOrdered` no es menor o igual que `RANGE1`, entonces debe ser mayor que `RANGE1`, así que la siguiente decisión (sombreada en la figura) es innecesaria. La lógica de computadora nunca ejecutará la decisión sombreada a menos que `itemsOrdered` ya sea mayor que `RANGE1`; es decir, a menos que la lógica siga la rama falsa de la primera selección. Si usted usa la lógica en la figura 4-22, desperdicia tiempo haciendo una pregunta que ya se ha contestado antes. La misma lógica se aplica a la segunda decisión sombreada en la figura 4-22. Los programadores principiantes en ocasiones justifican su uso de preguntas innecesarias como “sólo me aseguro por completo”. Dicha precaución es innecesaria cuando se escribe lógica de computadora.

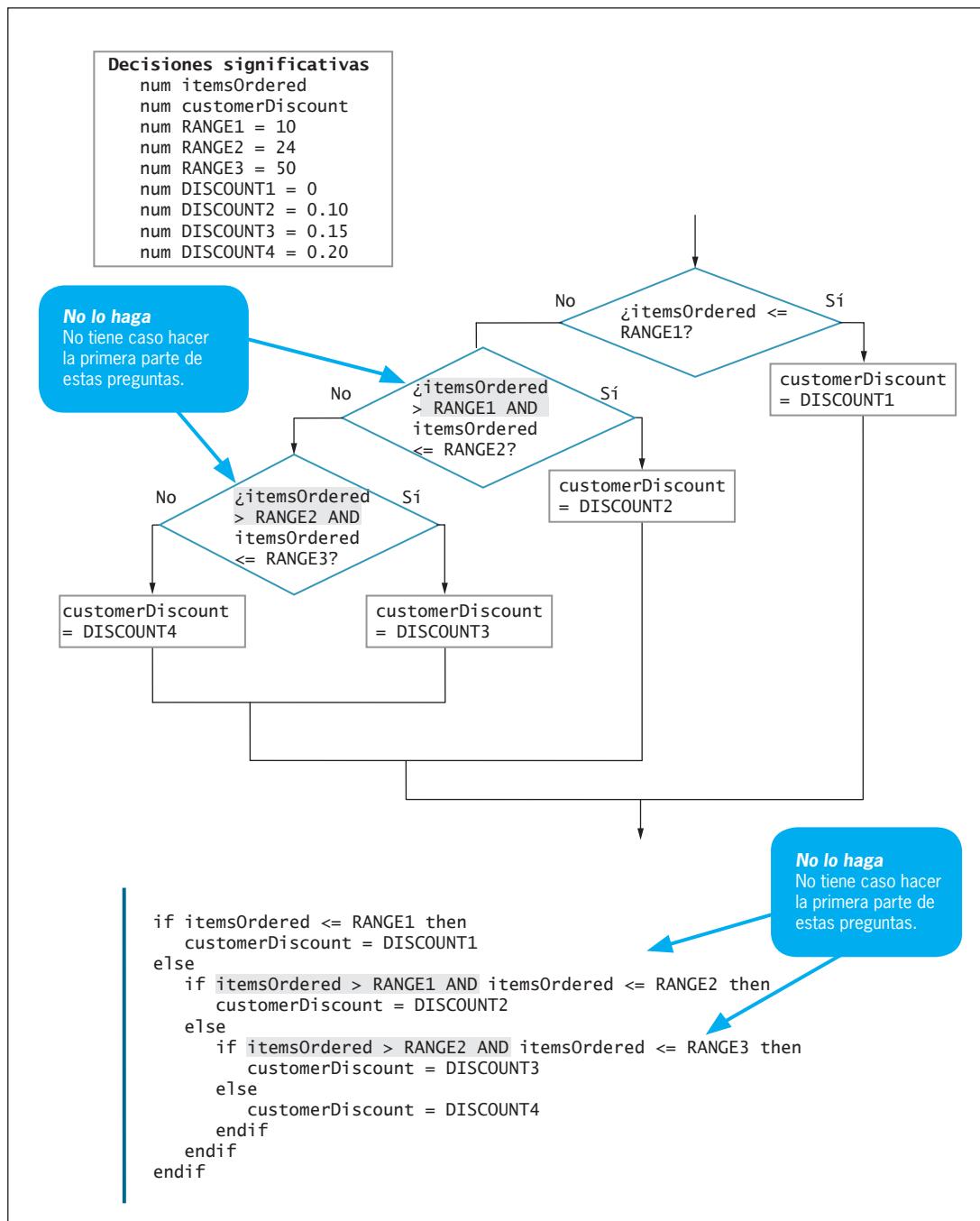


Figura 4-22 Selección de rango ineficiente que incluye preguntas ineficientes

DOS VERDADES Y UNA MENTIRA

Selección dentro de rangos

1. Cuando usted realiza una comprobación de rango, compara una variable con todos los valores en una serie de rangos.
2. Puede realizar una comprobación de rango haciendo comparaciones al usar el valor mínimo en cada rango de valores que utilice.
3. Puede realizar una comprobación de rango haciendo comparaciones al usar el valor máximo en cada rango de valores que utilice.

La afirmación falsa es la número 1. Cuando usa una comprobación de rango, compara una variable con una serie de valores que representan los extremos de los rangos. Dependiendo de su lógica, puede usar el extremo superior o el extremo inferior de cada rango.

Comprensión de la precedencia cuando se combinan operadores AND y OR

La mayoría de los lenguajes de programación permiten combinar en una expresión tantos operadores AND y OR como se necesite. Por ejemplo, suponga que necesita lograr una calificación al menos de 75 en cada uno de tres exámenes para aprobar un curso. Puede declarar una constante MIN_SCORE igual a 75 y probar las múltiples condiciones con una declaración como la siguiente:

```
if score1 >= MIN_SCORE AND score2 >= MIN_SCORE AND score3 >= MIN_SCORE then
    classGrade = "Aprobado"
else
    classGrade = "Reprobado"
endif
```

Por otra parte, si necesita aprobar sólo uno de los tres exámenes para aprobar un curso, entonces la lógica es como sigue:

```
if score1 >= MIN_SCORE OR score2 >= MIN_SCORE OR score3 >= MIN_SCORE then
    classGrade = "Aprobado"
else
    classGrade = "Reprobado"
endif
```

La lógica se vuelve más complicada cuando combina operadores AND y OR dentro de la misma declaración. Cuando lo hace, los operadores AND tienen **precedencia**, lo que significa que los valores booleanos de sus expresiones se evalúan primero.

Por ejemplo, considere un programa que determina si el cliente de un cine puede comprar un boleto con descuento. Suponga que se permiten descuentos para niños y adultos mayores que

asisten a las películas con clasificación G. El siguiente código parece razonable, pero genera resultados incorrectos porque la expresión que contiene el operador AND (véase el sombreado) se evalúa antes que la que contiene el operador OR.

```
if age <= 12 OR age >= 65 AND rating = "G" then
    output "Aplica descuento"
endif
```

Por ejemplo, suponga que un cliente del cine tiene 10 años de edad y la clasificación de la película es R. El cliente no debería recibir un descuento (¡ni siquiera debería permitírselo ver la película!). Sin embargo, dentro de la declaración

`if`, la parte de la expresión que contiene el operador AND, `age >= 65 AND rating = "G"`, se evalúa primero. Para un niño de 10 años y una película clasificada R, la pregunta es falsa (en ambos casos), de modo que la declaración `if` entera se vuelve el equivalente de lo siguiente:

```
if age <= 12 OR aFalseExpression then
    output "Aplica descuento"
endif
```

Debido a que el cliente tiene 10 años, `age <= 12` es verdadera, así que la declaración `if` original se vuelve el equivalente de:

```
if aTrueExpression OR aFalseExpression then
    output "Aplica descuento"
endif
```

La combinación verdadero OR falso se evalúa como verdadera. Por consiguiente, se da salida a la cadena "Aplica descuento" cuando no debería ser.

Muchos lenguajes de programación le permiten usar paréntesis para corregir la lógica y obligar a que la expresión OR sea evaluada primero, como se muestra en el siguiente pseudocódigo:

```
if (age <= 12 OR age >= 65) AND rating = "G" then
    output "Aplica descuento"
endif
```

Con los paréntesis agregados, si `age` del cliente es 12 o menos OR `age` es 65 o más, la expresión se evalúa como:

```
if aTrueExpression AND rating = "G" then
    output "Aplica descuento"
endif
```

En esta declaración, cuando el valor de edad califica a un cliente para un descuento, entonces el valor de clasificación también debe ser aceptable antes de que se aplique el descuento. Ésta era la intención original.

No lo haga

AND se evalúa primero, lo cual no es la intención.

Usted puede usar las siguientes técnicas para evitar la confusión cuando mezcle operadores AND y OR:

- Puede usar paréntesis para anular el orden predeterminado de las operaciones.
- Puede usar paréntesis por claridad aun cuando no cambien lo que el orden de las operaciones sería sin ellos. Por ejemplo, si un cliente estuviera entre 12 y 19 o tuviera una credencial escolar para recibir un descuento de bachillerato, es posible usar la expresión `(age > 12 AND age < 19) OR validId = "Sí"`, aun cuando la evaluación sería la misma sin los paréntesis.
- Puede usar declaraciones `if` anidadas en lugar de operadores AND y OR. Con el diagrama de flujo y el seudocódigo que se muestran en la figura 4-23, es claro cuáles clientes del cine recibirán el descuento. En el diagrama de flujo, se ve que OR está anidado por completo dentro de la rama Sí de la selección `rating = "G"`? Del mismo modo, en el seudocódigo de la figura 4-23, se ve por la alineación que si la clasificación no es G, la lógica procede en forma directa a la última declaración `endif`, evitando por completo cualquier comprobación de `age`.

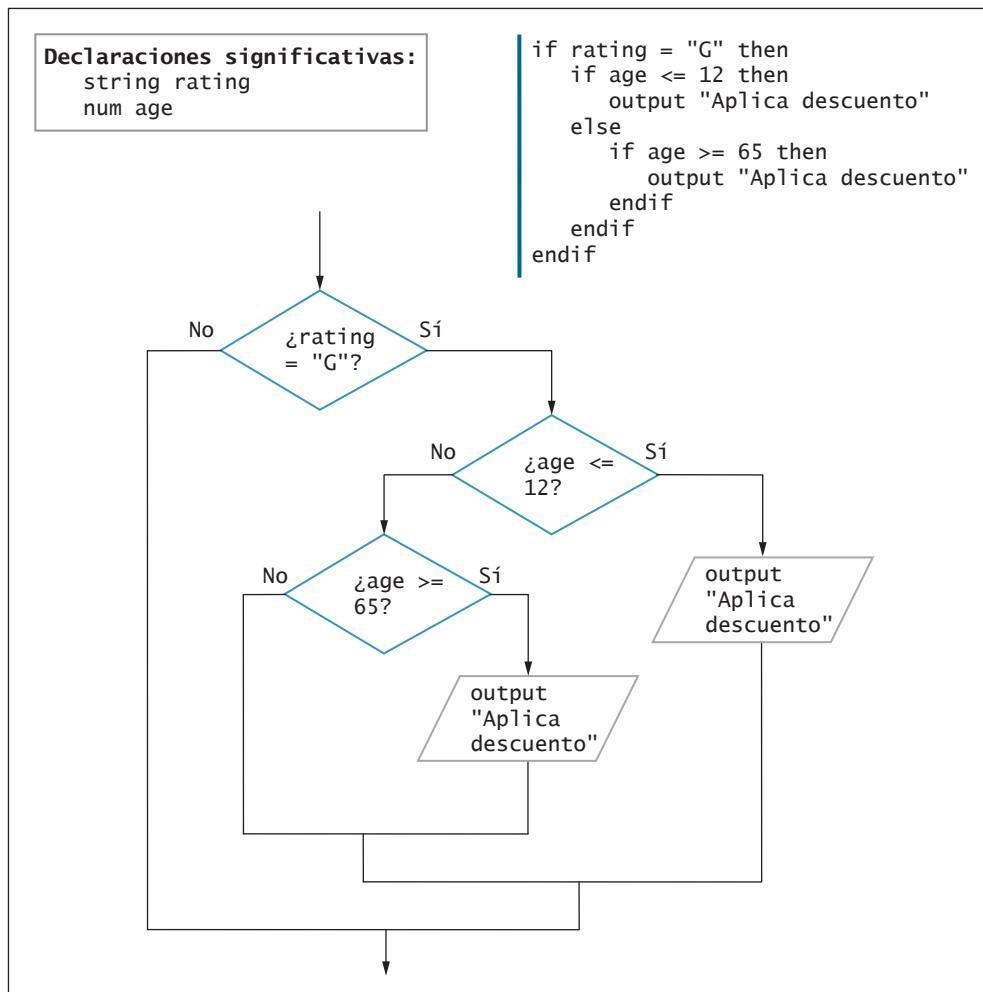


Figura 4-23 Decisiones anidadas que determinan el descuento de los clientes del cine

DOS VERDADES Y UNA MENTIRA

Comprensión de la precedencia cuando se combinan operadores AND y OR

1. La mayor parte de los lenguajes de programación le permiten combinar tantos operadores AND y OR en una expresión como necesite.
2. Cuando combina operadores AND y OR, los operadores OR tienen precedencia, lo que significa que sus valores booleanos se evalúan primero.
3. Siempre puede evitar la confusión de mezclar decisiones AND y OR anidando declaraciones `if` en lugar de usar operadores AND y OR.

La afirmación falsa es la número 2. Cuando combina operadores AND y OR, los operadores AND tienen precedencia, lo que significa que los valores booleanos de sus expresiones se evalúan primero.

Resumen del capítulo

- Las decisiones en los programas de computadora se toman evaluando las expresiones booleanas. Usted puede usar estructuras `if-then-else` o `if-then` para elegir entre dos resultados posibles.
- Usted puede usar operadores de comparación relacionales para comparar dos operandos del mismo tipo. Los operadores de comparación estándar son `=`, `>`, `<`, `>=`, `<=` y `<>`.
- En una decisión AND, dos condiciones deben ser verdaderas para que ocurra una acción resultante. Una decisión AND requiere una decisión anidada o el uso de un operador AND. En una decisión AND, el enfoque más eficiente es hacer la pregunta que tenga menos probabilidad de ser verdadera.
- En una decisión OR, al menos una de dos condiciones debe ser verdadera para que se presente una acción resultante; el enfoque más eficiente es hacer la pregunta que tiene más probabilidades de ser verdadera. La mayoría de los lenguajes de programación permiten hacer dos o más preguntas en una sola comparación usando un operador condicional OR.
- Para realizar una comprobación de rango, haga comparaciones ya sea con el valor mínimo o máximo en cada rango de valores de comparación. Los errores comunes que ocurren cuando los programadores realizan comprobaciones de rango incluyen las preguntas innecesarias y que se han respondido antes.
- Cuando usted combina operadores AND y OR en una expresión, los operadores AND tienen precedencia, lo que significa que sus valores booleanos se evalúan primero.

Términos clave

Una **expresión booleana** es la que representa sólo uno de dos estados, que por lo general se expresan como verdadero o falso.

Una estructura de decisión **if-then** contiene una expresión booleana probada y una acción que sólo ocurre cuando la expresión es verdadera.

Una **cláusula if-then** de una decisión contiene la acción que resulta cuando la expresión booleana en la decisión es verdadera.

La **cláusula else** de una decisión contiene la acción o acciones que se ejecutan sólo cuando la expresión booleana en la decisión es falsa.

Los **operadores de comparación relacionales** son los símbolos que expresan comparaciones booleanas. Ejemplos incluyen =, >, <, >=, <= y <>.

Una **expresión trivial** es aquella que siempre se evalúa con el mismo valor.

Una **condición compuesta** se construye cuando usted necesita hacer múltiples preguntas antes de determinar un resultado.

Una **decisión AND** contiene dos o más decisiones; todas las condiciones deben ser verdaderas para que una acción ocurra.

Una **decisión anidada**, o un **if anidado**, es una decisión dentro de la cláusula **if-then** o **else** de otra decisión.

Una **declaración if en cascada** es una serie de declaraciones **if** anidadas.

Un **operador condicional AND** (o, de manera más sencilla, un **operador AND**) es un símbolo que se usa para combinar decisiones de modo que dos o más condiciones deben ser verdaderas para que una acción ocurra.

Las **tablas de verdad** son diagramas que se usan en matemáticas y lógica para describir la verdad de una expresión entera con base en la verdad de sus partes.

La **evaluación de cortocircuito** es una característica lógica en la que las expresiones en cada parte de una expresión más grande se evalúan sólo en tanto sea necesario para determinar el resultado final.

Una **decisión OR** contiene dos o más decisiones; si al menos una condición se cumple, la acción resultante ocurre.

Un **operador condicional OR** (o, de manera más sencilla, un **operador OR**) es un símbolo que se usa para combinar decisiones cuando cualquier condición puede ser verdadera para que ocurra una acción.

El **operador lógico NOT** es un símbolo que invierte el significado de una expresión booleana.

Cuando se usa una **comprobación de rango**, se compara una variable con una serie de valores que marcan los extremos limitantes de los rangos.

Una **ruta sin salida o inalcanzable** es una ruta lógica que nunca puede recorrerse.

La **precedencia** es la cualidad de una operación que determina el orden en el cual se evalúa.

Preguntas de repaso

1. La declaración de selección `if quantity > 100 then discountRate = RATE` es un ejemplo de _____.
a) una selección de alternativa dual c) un ciclo estructurado
b) una selección de alternativa única d) todo lo anterior
2. La declaración de selección `if dayOfWeek = "Domingo" then price = LOWER_PRICE else price = HIGHER_PRICE` es un ejemplo de _____.
a) una selección de alternativa dual c) una selección unaria
b) una selección de alternativa única d) todo lo anterior
3. Todas las declaraciones de selección deben tener _____.
a) una cláusula `then` c) tanto a como b
b) una cláusula `else` d) ninguno de los anteriores
4. Una expresión como `amount < 10?` es una expresión _____.
a) gregoriana c) maquiavélica
b) eduardiana d) booleana
5. Por lo general, usted sólo compara variables que tienen el mismo _____.
a) tipo c) nombre
b) tamaño d) valor
6. Símbolos como `>` y `<` se conocen como operadores _____.
a) aritméticos c) comparación relacional
b) secuenciales d) precisión de escritura
7. Si pudiera usar sólo tres operadores de comparación relacionales, podría arreglárselas con _____.
a) mayor que, menor que y mayor o igual que
b) igual a, menor que y mayor que
c) menor que, menor o igual que y no igual a
d) igual a, no igual a y menor que
8. Si $a > b$ es falso, ¿entonces cuál de los siguientes siempre es verdadero?
a) $a \leq b$ c) $a = b$
b) $a < b$ d) $a \geq b$

13. Boffo Balloon Company fabrica globos de helio en tres tamaños, 12 colores y con una opción de 40 frases impresas. Como promoción, la compañía ofrece un descuento de 25% en los pedidos de los globos grandes, rojos, impresos con la frase "Feliz Día de los Novios". Para seleccionar de manera más eficiente los pedidos en los que aplica el descuento, usted usaría _____.

- a) declaraciones `if` anidadas con lógica OR
- b) declaraciones `if` anidadas con lógica AND
- c) tres declaraciones `if` sin anidar completamente separadas
- d) No se proporciona suficiente información

14. En el siguiente seudocódigo, ¿qué porcentaje de aumento recibirá un empleado en el Departamento 5?

```
if department < 3 then
    raise = SMALL_RAISE
else
    if department < 5 then
        raise = MEDIUM_RAISE
    else
        raise = BIG_RAISE
    endif
endif
```

- a) SMALL_RAISE
- b) MEDIUM_RAISE
- c) BIG_RAISE
- d) imposible decirlo

15. En el siguiente seudocódigo, ¿qué porcentaje de aumento recibirá un empleado en el Departamento 8?

```
if department < 5 then
    raise = SMALL_RAISE
else
    if department < 14 then
        raise = MEDIUM_RAISE
    else
        if department < 9 then
            raise = BIG_RAISE
        endif
    endif
endif
```

- a) SMALL_RAISE
- b) MEDIUM_RAISE
- c) BIG_RAISE
- d) imposible decirlo

16. En el siguiente seudocódigo, ¿qué porcentaje de aumento recibirá un empleado en el Departamento 10?

```
if department < 2 then
    raise = SMALL_RAISE
else
    if department < 6 then
        raise = MEDIUM_RAISE
    else
        if department < 10 then
            raise = BIG_RAISE
        endif
    endif
endif
```


Ejercicios

1. Suponga que las siguientes variables contienen los valores que se muestran:

```
numberBig = 300          numberMedium = 100        numberSmall = 5  
wordBig = "Dinosaurio"  wordMedium = "Caballo"   wordSmall = "Pato"
```

Para cada una de las siguientes expresiones booleanas, decida si la declaración es verdadera, falsa o ilegal.

- a) `numberBig = numberSmall?`
 - b) `numberBig > numberSmall?`
 - c) `numberMedium < numberSmall?`
 - d) `numberBig = wordBig?`
 - e) `numberBig = "Grande"`
 - f) `wordMedium = "Mediano"`
 - g) `wordBig = "Dinosaurio"`
 - h) `numberMedium <= numberBig / 3?`
 - i) `numberBig >= 200?`
 - j) `numberBig >= numberMedium + numberSmall?`
 - k) `numberBig > numberMedium AND numberBig < numberSmall?`
 - l) `numberBig = 100 OR numberBig > numberSmall?`
 - m) `numberBig < 10 OR numberSmall > 10?`
 - n) `numberBig = 30 AND numberMedium = 100 OR numberSmall = 100?`
2. Mortimer Life Insurance Company desea varias listas de datos de personal de ventas. Diseñe un diagrama de flujo o seudocódigo para lo siguiente:
- a) Un programa que acepte el número de ID de un vendedor y el número de pólizas vendidas en el último mes, y despliegue los datos sólo si el vendedor tiene un alto rendimiento, una persona que vende más de 25 pólizas en el mes.
 - b) Un programa que acepte datos del vendedor en forma continua hasta que se introduzca un valor centinela y despliegue una lista de personas de alto rendimiento.
3. ShoppingBay es un servicio de subasta en línea que requiere varios informes. Diseñe un diagrama de flujo o seudocódigo para lo siguiente:
- a) Un programa que acepte datos de la subasta como sigue: número de ID, descripción del artículo, duración de la subasta en días y oferta mínima requerida. Despliegue datos para una subasta si la oferta mínima requerida es más de 100 dólares.
 - b) Un programa que acepte en forma continua datos de la subasta hasta que se introduzca un valor centinela y despliegue una lista de todos los datos para subastas en las que la oferta mínima requerida sea mayor que 100 dólares.
 - c) Un programa que acepte en forma continua datos de la subasta y despliegue datos para cada subasta en la que la oferta mínima sea \$0.00 y la duración de la subasta sea un día o menos.

- d) Un programa que acepte en forma continua datos de la subasta y despliegue datos para cada subasta en la que la duración esté entre 7 y 30 días inclusive.
- e) Un programa que pida al usuario una oferta máxima requerida, y luego acepte en forma continua datos de la subasta y despliegue datos para cada subasta en la que la oferta mínima sea menor o igual que la cantidad introducida por el usuario.
4. Dash Cell Phone Company cobra a sus clientes una tarifa básica de \$5 por mes por enviar mensajes de texto. Las tarifas adicionales son como sigue:
- Los primeros 60 mensajes por mes, sin importar la longitud del mensaje, se incluyen en la factura básica.
 - Se cobran cinco centavos adicionales por cada mensaje de texto después del 60o. mensaje, hasta 180 mensajes.
 - Se cobran 10 centavos adicionales por cada mensaje de texto después del 180o. mensaje.
 - Los impuestos federales, estatales y locales suman un total de 12% de cada factura.
- Diseñe un diagrama de flujo o seudocódigo para lo siguiente:
- a) Un programa que acepte los siguientes datos sobre la factura de un cliente: código de área donde se encuentra (tres dígitos), número de teléfono (siete dígitos) y número de mensajes de texto enviados. Despliegue todos los datos, incluyendo la factura mensual final tanto como después de agregar los impuestos.
- b) Un programa que acepte en forma continua datos sobre los mensajes de texto hasta que se introduzca un valor centinela y despliegue todos los detalles.
- c) Un programa que acepte en forma continua datos sobre mensajes de texto hasta que se introduzca un valor centinela y sólo despliegue detalles sobre clientes que envíen más de 100 mensajes de texto.
- d) Un programa que acepte en forma continua datos sobre mensajes de texto hasta que se introduzca un valor centinela y sólo despliegue detalles sobre clientes cuya factura total con impuestos sea mayor de \$20.
- e) Un programa que pida al usuario un código de área de tres dígitos de la cual seleccionar facturas. Luego el programa acepta en forma continua datos de mensajes de texto hasta que se introduzca un valor centinela y sólo despliegue datos para mensajes enviados desde el código de área especificado.
5. Drive-Rite Insurance Company proporciona pólizas de seguros para automóviles a los conductores. Diseñe un diagrama de flujo o seudocódigo para lo siguiente:
- a) Un programa que acepte datos de pólizas de seguros, incluyendo el número de las mismas, apellido del cliente, nombre del cliente, edad, fecha de vencimiento de la prima (día, mes y año) y número de accidentes que ha tenido el conductor en los últimos tres años. Si un número de póliza introducido no está entre 1,000 y 9,999 inclusive, establezca el número de póliza en 0. Si el mes no está entre 1 y 12 inclusive, o el día no es correcto para el mes (por ejemplo, no está entre 1 y 31 para enero o 1 y 29 para febrero), establezca el día, mes y año en 0. Despliegue los datos de la póliza después que se hayan hecho cualesquiera revisiones.

- b) Un programa que acepte en forma continua los datos de los tenedores de pólizas hasta que se introduzca un valor centinela y despliegue los datos para cualquier tenedor de póliza mayor de 35 años de edad.
- c) Un programa que acepte datos de los tenedores de pólizas y despliegue los datos para cualquier tenedor de póliza que sea menor de 21 años de edad.
- d) Un programa que acepte datos de los tenedores de pólizas y despliegue los datos para cualquier tenedor de póliza no mayor de 30 años de edad.
- e) Un programa que acepte datos de los tenedores de pólizas y despliegue los datos para cualquier tenedor de póliza cuya prima venza a más tardar el 15 de marzo de cualquier año.
- f) Un programa que acepte datos de los tenedores de pólizas y despliegue los datos para cualquier tenedor de póliza cuya prima se venza hasta e incluyendo al 1 de enero de 2014.
- g) Un programa que acepte datos de los tenedores de pólizas y despliegue los datos para cualquier tenedor de póliza cuya prima se venza el 27 de abril de 2013.
- h) Un programa que acepte datos de los tenedores de pólizas y despliegue los datos para cualquiera que tenga un número de póliza entre 1,000 y 4,000 inclusive, cuya póliza se venza en abril o mayo de cualquier año, y que haya tenido menos de tres accidentes.
6. The Barking Lot es una guardería para perros. Diseñe un diagrama de flujo o pseudocódigo para lo siguiente:
- Un programa que acepte datos para un número de ID del propietario de un perro, y nombre, raza, edad y peso de este último. Despliegue una factura que contenga todos los datos de entrada al igual que la tarifa semanal de la guardería, la cual es de \$55 para perros con menos de 7 kilogramos, \$75 para perros de 7 a 14 kilogramos inclusive, \$105 para perros de 14.1 a 37 kilogramos inclusive, y \$125 para perros con más de 37 kilogramos.
 - Un programa que acepte en forma continua los datos de los perros hasta que se introduzca un valor centinela y despliegue datos de facturación para cada perro.
 - Un programa que acepte en forma continua los datos de los perros hasta que se introduzca un valor centinela y despliegue datos de facturación para los propietarios de perros que deban más de \$100.
7. Mark Daniels es un carpintero que crea letreros personalizados para casas. Desea una aplicación para calcular el precio de cualquier letrero que pida un cliente, con base en los siguientes factores:
- El cargo mínimo para todos los letreros es \$30.
 - Si el letrero se hace de roble, agregue \$15. No se agrega ningún cargo por pino.
 - Las primeras seis letras o números se incluyen en el cargo mínimo; hay un cargo de \$3 por cada carácter adicional.
 - Los caracteres blancos o negros están incluidos en el cargo mínimo; hay un cargo adicional de \$12 para letras laminadas en oro.

Diseñe un diagrama de flujo o seudocódigo para lo siguiente:

- a) Un programa que acepte datos para un número de pedido, nombre del cliente, tipo de madera, número de caracteres y color de los caracteres. Despliegue todos los datos introducidos y el precio final para el letrero.
 - b) Un programa que acepte en forma continua datos de pedidos de letreros y despliegue toda la información relevante para los que son de roble con cinco letras blancas.
 - c) Un programa que acepte en forma continua datos de pedidos de letreros y despliegue toda la información relevante para los que son de pino con letras laminadas en oro y más de 10 caracteres.
8. Black Dot Printing intenta organizar un transporte compartido para ahorrar energía. Cada registro de entrada contiene el nombre y el poblado de residencia de un empleado. Diez por ciento de los empleados de la compañía vive en Wonder Lake; 30% vive en el poblado adyacente de Woodstock. Black Dot desea alentar a los empleados que viven en cualquier poblado para viajar juntos al trabajo. Diseñe un diagrama de flujo o seudocódigo para lo siguiente:
- a) Un programa que acepte los datos de un empleado y lo despliegue con un mensaje que indique si es un candidato para compartir el vehículo.
 - b) Un programa que acepte en forma continua los datos de los empleados hasta que se introduzca un valor centinela y despliegue una lista de todos los empleados que son candidatos para compartir vehículo.
9. Amanda Cho, supervisora en una tienda minorista de ropa, desea reconocer a los vendedores de alto rendimiento. Diseñe un diagrama de flujo o seudocódigo para lo siguiente:
- a) Un programa que acepte en forma continua el nombre y apellido de cada vendedor, el número de turnos que trabajó en un mes, número de transacciones que completó ese mes y el valor en dólares de esas transacciones. Despliegue el nombre de cada vendedor con una puntuación de productividad, misma que se calcula dividiendo primero los dólares entre las transacciones y dividiendo el resultado entre los turnos trabajados. Despliegue tres asteriscos después de la puntuación de productividad si es de 50 o más.
 - b) Un programa que acepte los datos de cada vendedor y despliegue el nombre y una cantidad de bonificación. Los bonos se distribuirán como sigue:
 - Si la puntuación de productividad es 30 o menos, el bono es de \$25.
 - Si la puntuación de productividad es 31 o más y menos que 80, el bono es de \$50.
 - Si la puntuación de productividad es 80 o más y menos que 200, el bono es de \$100.
 - Si la puntuación de productividad es 200 o más, el bono es de \$200.
 - c) Modifique el ejercicio 9b para que refleje el siguiente hecho nuevo, y haga que el programa se ejecute de la manera más eficiente posible.
 - Sesenta por ciento de los empleados tienen una puntuación de productividad mayor que 200.



Encuentre los errores

- Sus archivos descargables para el capítulo 4 incluyen DEBUG04-01.txt, DEBUG04-02.txt y DEBUG04-03.txt. Cada archivo comienza con algunos comentarios que describen el problema. Los comentarios son líneas que comienzan con dos diagonales (//). Después de los comentarios, cada archivo contiene seudocódigo que tiene uno o más errores que usted debe encontrar y corregir. (NOTA: Estos archivos se encuentran disponibles sólo para la versión original en inglés.)



Zona de juegos

- En el capítulo 2, aprendió que muchos lenguajes de programación le permiten generar un número aleatorio entre 1 y un valor límite llamado LIMIT usando una declaración similar a `randomNumber = random(LIMIT)`. Cree la lógica para un juego de adivinanza en el que la aplicación genere un número aleatorio y el jugador trate de adivinarlo. Despliegue un mensaje que indique si la conjectura del jugador fue correcta, demasiado alta o demasiado baja. (Después de terminar el capítulo 5 será capaz de modificar la aplicación de modo que el usuario pueda continuar adivinando hasta que se introduzca la respuesta correcta.)
- Cree una aplicación para un juego de lotería. Genere tres números aleatorios, cada uno entre 0 y 9. Permita que el usuario adivine tres números. Compare cada una de las conjecturas del usuario de los tres números aleatorios y despliegue un mensaje que incluya la conjectura del usuario, los tres dígitos determinados aleatoriamente y la cantidad de dinero que el usuario ha ganado, como se muestra en el cuadro 4-4.

Números coincidentes	Premio (\$)
Cualquier número que coincide	10
Dos coincidentes	100
Tres coincidentes, sin orden	1,000
Tres coincidentes en el orden exacto	1,000,000
Ninguno coincidente	0

Cuadro 4-4 Premios por adivinar los números en un juego de lotería

Asegúrese de que su aplicación puede repetir dígitos. Por ejemplo, si un usuario conjectura 1, 2 y 3, y los dígitos generados en forma aleatoria son 1, 1 y 1, no dé crédito al usuario por tres coincidencias correctas, sólo una.

*Para discusión*

13. Los programas de computadora pueden usarse para tomar decisiones sobre su asegurabilidad al igual que las tarifas que se le cobrarán por sus pólizas de seguro de salud y de vida. Por ejemplo, ciertas condiciones preexistentes pueden elevar sus primas de seguros en forma considerable. ¿Es ético que las compañías de seguros tengan acceso a sus registros médicos y luego tomen decisiones de seguros relacionadas con usted? Explique su respuesta.
14. Las solicitudes de empleo en ocasiones son examinadas por software que toma decisiones sobre la idoneidad del candidato con base en las palabras clave en las solicitudes. ¿Este examen es justo para los candidatos? Explique su respuesta.
15. Las instalaciones médicas con frecuencia tienen más pacientes esperando por trasplantes de órganos que órganos disponibles. Suponga que se le ha pedido que escriba un programa que seleccione cuáles candidatos deberían recibir un órgano disponible. ¿Qué datos en el archivo desearía usar en su programa y qué decisiones tomaría con base en los datos? ¿Qué datos piensa que podrían usar otros que usted no elegiría usar?

5

CAPÍTULO

Creación de ciclos

En este capítulo usted aprenderá sobre:

- ◎ Las ventajas de crear ciclos
- ◎ Usar una variable de control de ciclo
- ◎ Ciclos anidados
- ◎ Evitar errores de ciclo comunes
- ◎ Usar un ciclo `for`
- ◎ Aplicaciones comunes de los ciclos

Comprensión de las ventajas de crear ciclos

Aunque tomar decisiones es lo que hace que las computadoras parezcan inteligentes, la creación de ciclos hace que la programación de computadoras sea eficiente y que valga la pena. Cuando usted usa un ciclo, un conjunto de instrucciones opera en múltiples conjuntos separados de datos. El uso de menos instrucciones resulta en menos tiempo requerido para el diseño y la codificación, menos errores y un tiempo de compilación más breve.

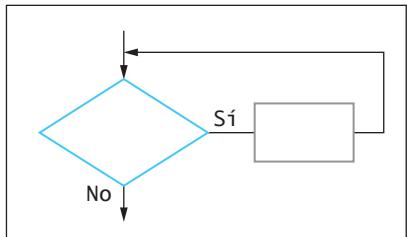


Figura 5-1 La estructura de ciclo

Recuerde la estructura de ciclo sobre la que aprendió en el capítulo 3; se ve como la figura 5-1. En tanto una expresión booleana siga siendo verdadera, el cuerpo de un ciclo `while` se ejecuta.

Usted ya ha aprendido que muchos programas usan un ciclo para controlar las tareas repetitivas. Por ejemplo, la figura 5-2 muestra la estructura básica de muchos programas de negocios. Después de que se completan algunas tareas de administración, el ciclo detallado se repite una vez por cada registro de datos que debe procesarse.

Por ejemplo, la figura 5-2 podría representar la lógica de línea principal de un programa de nómina típico. Los datos del primer empleado se introducirán en el módulo `housekeeping()`, y mientras no se cumpla la condición `eof` el módulo `detailLoop()` ejecutará tareas como la

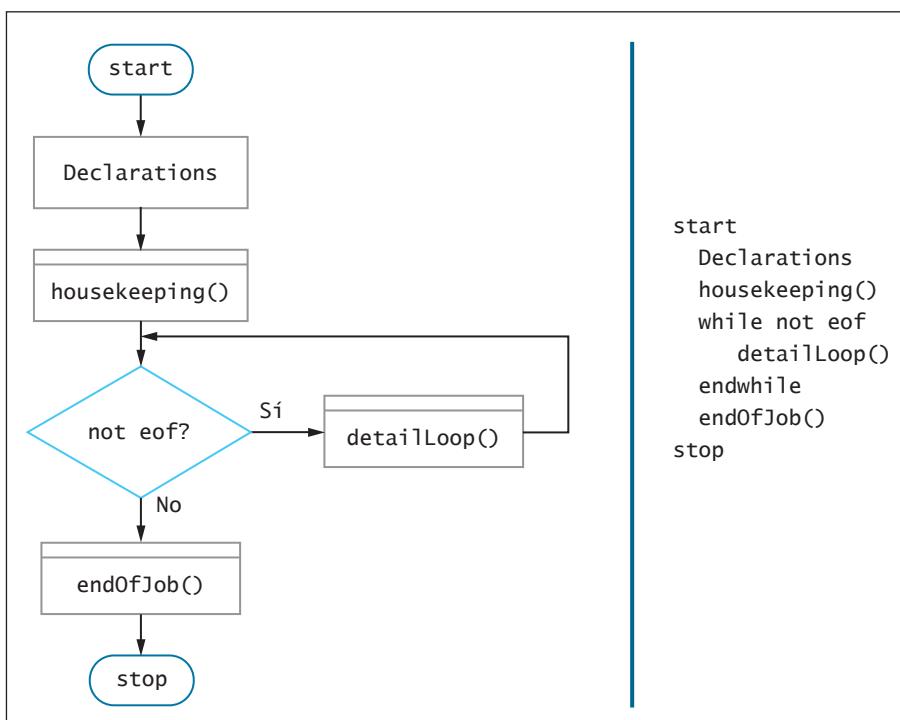


Figura 5-2 La lógica de línea principal común en muchos programas de negocios

determinación del salario regular y extraordinario y la deducción de impuestos, primas de seguro, contribuciones a la beneficencia, cuotas sindicales y otros elementos. Luego, después de que se ha producido el cheque para el pago de un empleado se introducirán los datos del siguiente y el módulo `detailLoop()` se repetirá. La ventaja de hacer que una computadora genere los cheques de nómina es que las instrucciones de cálculo sólo necesitan escribirse una vez y pueden repetirse de manera indefinida.

DOS VERDADES Y UNA MENTIRA

Comprensión de las ventajas de crear ciclos

1. Cuando usted usa un ciclo puede escribir un conjunto de instrucciones que opere en múltiples conjuntos separados de datos.
2. Una ventaja importante de hacer que una computadora ejecute las tareas complicadas es la capacidad para repetirlas.
3. Un ciclo es una estructura que se ramifica en dos rutas lógicas antes de continuar.

mientras algunas condiciones continúan.

La afirmación falsa es la número 3. Un ciclo es una estructura que repite acciones

Uso de una variable de control de ciclo

Usted puede usar un ciclo `while` para ejecutar un cuerpo de declaraciones en forma continua en tanto alguna condición continúe siendo verdadera. El cuerpo de un ciclo podría contener cualquier número de declaraciones, incluyendo llamadas a método, decisiones y otros ciclos. Para hacer que un ciclo `while` termine en forma correcta, debe declarar una **variable de control de ciclo** para manejar el número de repeticiones que ejecuta un ciclo. Deberían ocurrir tres acciones separadas:

- La variable de control de ciclo se inicializa antes de entrar al ciclo.
- La variable de control de ciclo se prueba, y si el resultado es verdadero se entra al cuerpo del ciclo.
- La variable de control de ciclo se altera dentro del cuerpo del ciclo de modo que la expresión `while` en algún momento se evalúa como falsa.

Si usted omite cualquiera de estas acciones o las ejecuta en forma incorrecta corre el riesgo de crear un ciclo infinito. Una vez que su lógica entra al cuerpo de un ciclo estructurado, el cuerpo del ciclo entero debe ejecutarse. Su programa puede dejar un ciclo estructurado sólo en la comparación que prueba la variable de control de ciclo. Por lo común, usted puede controlar las repeticiones de un ciclo en una de dos maneras:

- Usar un contador para crear un ciclo definido controlado por contador.
- Usar un valor centinela para crear un ciclo indefinido.

Uso de un ciclo definido con un contador

172

La figura 5-3 muestra un ciclo que despliega *Hola* cuatro veces. La variable `count` es la variable de control de ciclo. Éste es un **ciclo definido** porque se ejecuta un número predeterminado de veces; en este caso, cuatro. Es un **ciclo contado**, o **ciclo controlado por contador**, porque el programa sigue la pista del número de repeticiones del ciclo al contarlas.

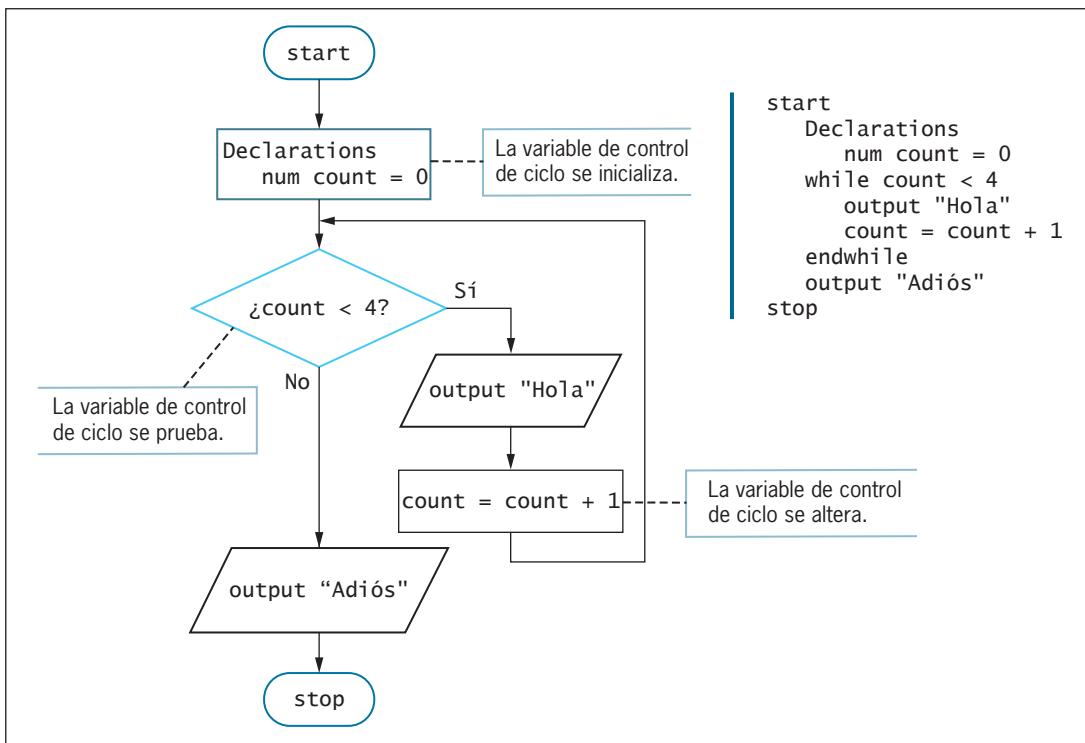


Figura 5-3 Un ciclo while contado que produce *Hola* cuatro veces

El ciclo en la figura 5-3 se ejecuta como sigue:

- La variable de control de ciclo, `count`, se inicializa en 0.
- La expresión `while` compara `count` con 4.
- El valor de `count` es menor que 4, y por tanto el cuerpo del ciclo se ejecuta. El cuerpo del ciclo que se muestra en la figura 5-3 consiste en dos declaraciones que despliegan *Hola* y luego agregan 1 a `count`.
- La siguiente vez que se evalúa `count`, su valor es 1, mismo que todavía es menor que 4, así que el cuerpo del ciclo se ejecuta de nuevo. *Hola* se despliega una segunda vez y `count` se vuelve 2, *Hola* se despliega una tercera vez y `count` se vuelve 3, entonces *Hola* se despliega

una cuarta vez y `count` se vuelve 4. Ahora, cuando se evalúa la expresión `count < 4?`, es `false`, así que el ciclo termina.

Dentro del cuerpo de un ciclo, puede cambiar el valor de la variable de control de ciclo en diversas formas. Por ejemplo:

- Usted podría sencillamente asignar un valor nuevo a la variable de control de ciclo.
- Podría recuperar un valor nuevo desde un dispositivo de entrada.
- Podría **incrementar**, o aumentar, la variable de control de ciclo, como en la lógica en la figura 5-3.
- Podría reducir, o **decrementar**, la variable de control de ciclo. Por ejemplo, el ciclo en la figura 5-3 podría reescribirse de modo que `count` se inicialice en 4 y se reduzca en 1 en cada paso a través del ciclo. Entonces este último debería continuar mientras `count` permanezca mayor que 0.

Los términos *incrementar* y *decrementar* por lo general se refieren a cambios pequeños; con frecuencia el valor que se usa para incrementar o decrementar la variable de control de ciclo es 1. Sin embargo, los ciclos también se controlan al sumar o restar valores diferentes de 1. Por ejemplo, para desplegar las ganancias de la compañía en intervalos de cinco años durante los siguientes 50 años, desearía sumar 5 a una variable de control de ciclo durante cada iteración.



Debido a que con frecuencia usted necesitará incrementar una variable, muchos lenguajes de programación contienen un operador de atajo para incrementar. Aprenderá sobre estos operadores cuando estudie un lenguaje de programación que los utilice.

La lógica de creación de ciclos que se muestra en la figura 5-3 usa un contador. Un **contador** es cualquier variable numérica que usted use para contar el número de veces que ha ocurrido un evento. En la vida cotidiana, las personas por lo general cuentan las cosas comenzando con 1. Muchos programadores prefieren comenzar sus ciclos contados con una variable que contenga 0 por dos razones:

- En muchas aplicaciones de computadora, la numeración comienza con 0 debido a la naturaleza de 0 y 1 de la circuitería de la computadora.
- Cuando aprenda sobre arreglos en el capítulo 6, descubrirá que la manipulación de éstos se presta de manera natural a los ciclos basados en 0.

Uso de un ciclo indefinido con un valor centinela

A menudo, el valor de una variable de control de ciclo no se altera con aritmética, sino por una entrada del usuario. Por ejemplo, quizás usted desea mantener alguna tarea ejecutándose mientras el usuario indica un deseo de continuar. En ese caso, cuando escribe el programa no sabe si el ciclo será ejecutado dos veces, 200 veces o no se ejecutará en absoluto. Éste es un **ciclo indefinido**.

Considere un programa interactivo que despliegue *Hola* de manera repetida en tanto el usuario desee continuar. El ciclo es indefinido porque podría ejecutarse un número de veces diferente cada vez que se ejecuta el programa. El programa aparece en la figura 5-4.

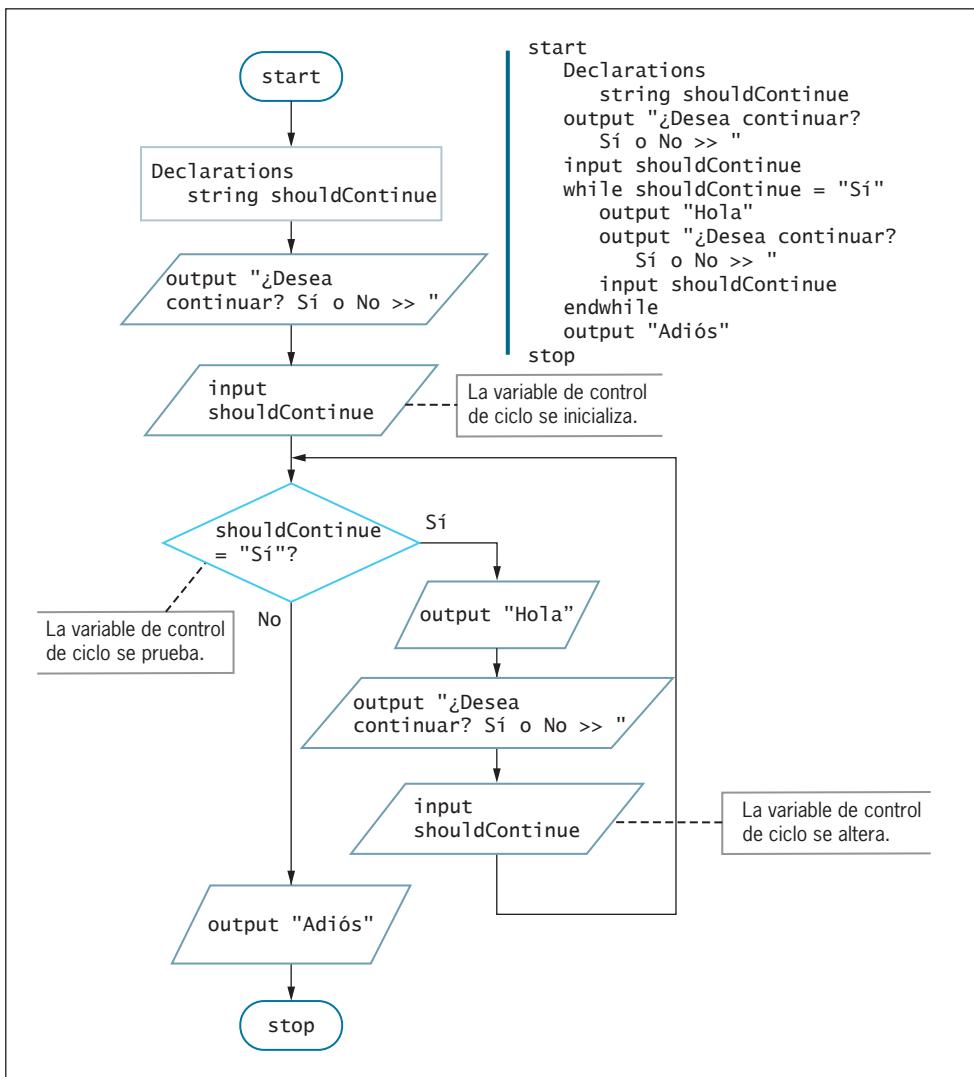


Figura 5-4 Un ciclo while indefinido que despliega *Hola* en tanto el usuario desee continuar

En el programa de la figura 5-4, la variable de control de ciclo es `shouldContinue`. El programa se ejecuta como sigue:

- La primera declaración `input shouldContinue` en la aplicación de la figura 5-4 es una declaración de entrada anticipada. En esta declaración, la primera respuesta del usuario inicializa la variable de control de ciclo.
- La expresión `while` compara la variable de control con el valor centinela *Sí*.
- Si el usuario ha introducido *Sí*, entonces se da salida a *Hola* y se pregunta al usuario si el programa debería continuar. En este paso, el valor de `shouldContinue` podría cambiar.

- En cualquier punto, si el usuario introduce cualquier valor diferente de *Sí*, el ciclo termina. En la mayoría de los lenguajes de programación, las comparaciones son sensibles a las mayúsculas y minúsculas, así que cualquier entrada distinta a *Sí*, incluyendo *sí*, terminará el ciclo.

La figura 5-5 muestra cómo podría verse el programa cuando es ejecutado en la línea de comandos y en un ambiente GUI. Las pantallas muestran programas que ejecutan exactamente las mismas tareas usando diferentes ambientes. En cada ambiente, es posible que el usuario continúe eligiendo ver los mensajes *Hola*, o que elija salir del programa y desplegar *Adiós*.

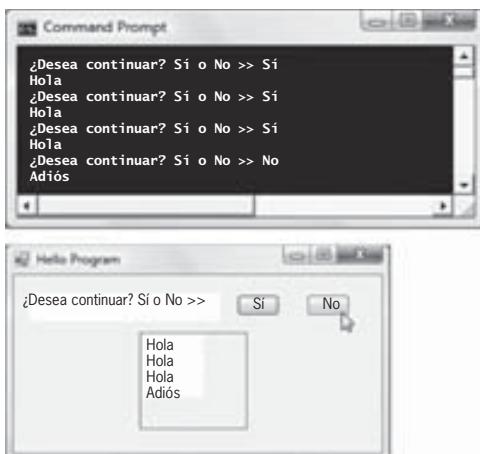


Figura 5-5 Ejecuciones típicas del programa de la figura 5-4 en dos ambientes

Comprensión del ciclo en la lógica de línea principal de un programa

Los segmentos de diagrama de flujo y seudocódigo en la figura 5-4 contienen tres pasos que deberían ocurrir en cualquier ciclo que funcione de manera apropiada:

- Usted debe proporcionar un valor inicial para la variable que controlará el ciclo.
- Debe probar la variable de control de ciclo para determinar si se ejecuta el cuerpo del mismo.
- Dentro del ciclo, debe alterar la variable de control de ciclo.

En el capítulo 2 aprendió que la lógica de línea principal de muchos programas de negocios sigue un esbozo estándar que consiste en tareas de administración, un ciclo que se repite y tareas de finalización. Los tres pasos cruciales que ocurren en cualquier ciclo también se presentan en la lógica de línea principal estándar. La figura 5-6 muestra el diagrama de flujo para la lógica de línea principal del programa de nómina que vio en la figura 2-8; se resaltan los tres pasos que controlan el ciclo. Aquí, los tres pasos (inicialización, prueba y alteración de la variable de control de ciclo) están en módulos diferentes. Sin embargo, todos ocurren en los lugares correctos y muestran que la lógica de línea principal usa un ciclo estándar y correcto.

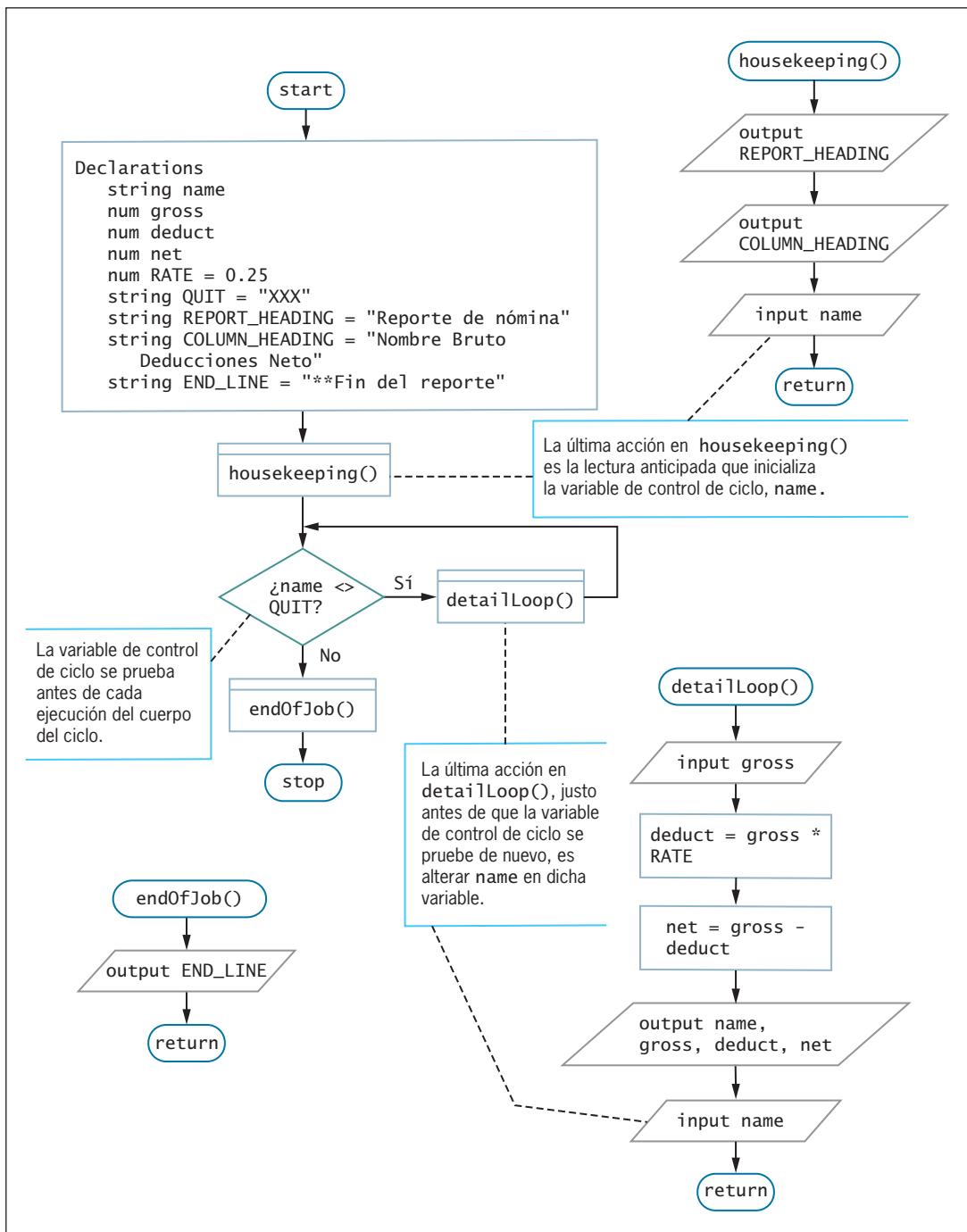


Figura 5-6 Un programa de nómina que muestra cómo se usa la variable de control de ciclo

DOS VERDADES Y UNA MENTIRA

Uso de una variable de control de ciclo

1. Para hacer que un ciclo `while` se ejecute en forma correcta, debe establecerse una variable de control de ciclo en 0 antes de entrar al ciclo.
2. Para hacer que un ciclo `while` se ejecute en forma correcta, debe probarse una variable de control de ciclo antes de entrar al cuerpo del ciclo.
3. Para hacer que un ciclo `while` se ejecute en forma correcta, el cuerpo del ciclo debe realizar alguna acción que altere el valor de la variable de control de ciclo.

zarse, pero no necesariamente en 0.

La afirmación falsa es la número 1. Una variable de control de ciclo debe inicializarse en 0.

Ciclos anidados

La lógica de programa se vuelve más complicada cuando es preciso usar unos ciclos dentro de otros, o **ciclos anidados**. Cuando uno aparece dentro de otro, el que contiene al otro se llama **ciclo exterior**, y el que está contenido se llama **ciclo interior**. Se necesita crear ciclos anidados cuando los valores de dos o más variables se repiten para producir cada combinación de valores. Por lo general, cuando se crean ciclos anidados, cada uno tiene su propia variable de control de ciclo.

Por ejemplo, suponga que desea escribir un programa que produce hojas de respuestas a cuestionarios como las que se muestran en la figura 5-7. Cada hoja de respuestas tiene un encabezado único seguido por cinco partes con tres preguntas en cada parte y usted desea una línea para responder cada pregunta. Podría escribir un programa que use 63 declaraciones de salida separadas para producir tres hojas (cada hoja contiene 21 líneas impresas), pero es más eficiente usar ciclos anidados.

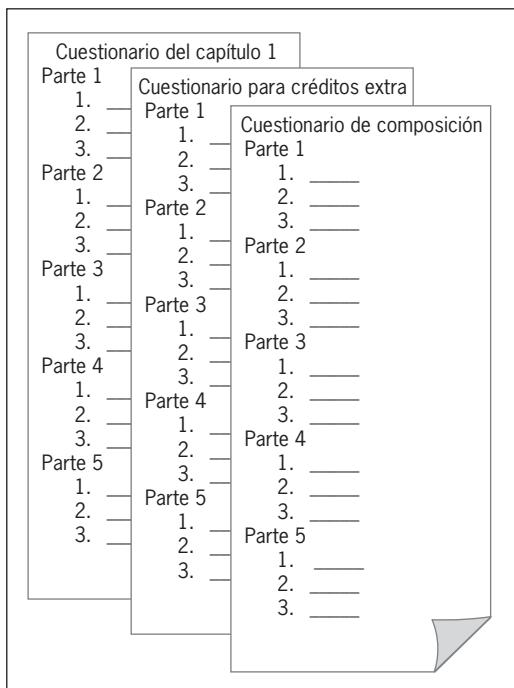


Figura 5-7 Hojas de respuestas de un cuestionario

La figura 5-8 muestra la lógica para el programa que produce hojas de respuestas. Se declaran tres variables de control de ciclo para el programa:

- `quizName` controla el módulo `detailLoop()` que se llama desde la lógica de línea principal.
- `partCounter` controla el ciclo exterior dentro del módulo `detailLoop()`; hace un seguimiento de las partes de la hoja de respuestas.
- `questionCounter` controla el ciclo interior en el módulo `detailLoop()`, hace un seguimiento de las preguntas y las líneas de respuesta dentro de cada sección de parte en cada hoja de respuestas.

También se declaran cinco constantes nombradas. Tres de estas constantes (`QUIT`, `PARTS` y `QUESTIONS`) contienen los valores centinela para cada uno de los tres ciclos en el programa. Las otras dos contienen el texto al que se dará salida (la palabra *Parte* que precede a cada número de parte y la combinación punto-espacio-línea que forma una línea de respuesta para cada pregunta).

Cuando empieza el programa, el módulo `housekeeping()` se ejecuta y el usuario introduce el nombre al que se dará salida en la parte superior del primer cuestionario. Si el usuario introduce el valor `QUIT`, el programa termina de inmediato, pero si el usuario introduce alguna otra cosa, como *Cuestionario de composición*, entonces se ejecuta el módulo `detailLoop()`.

En `detailLoop()` el nombre del cuestionario sale en la parte superior de la hoja de respuestas. Entonces se inicializa `partCounter` en 1. La variable `partCounter` es la variable de control de

ciclo para el ciclo exterior en este módulo. El ciclo exterior continúa mientras partCounter es menor o igual que PARTS. La última declaración en el ciclo exterior agrega 1 a partCounter. En otras palabras, el ciclo exterior se ejecutará cuando partCounter sea 1, 2, 3, 4 y 5.

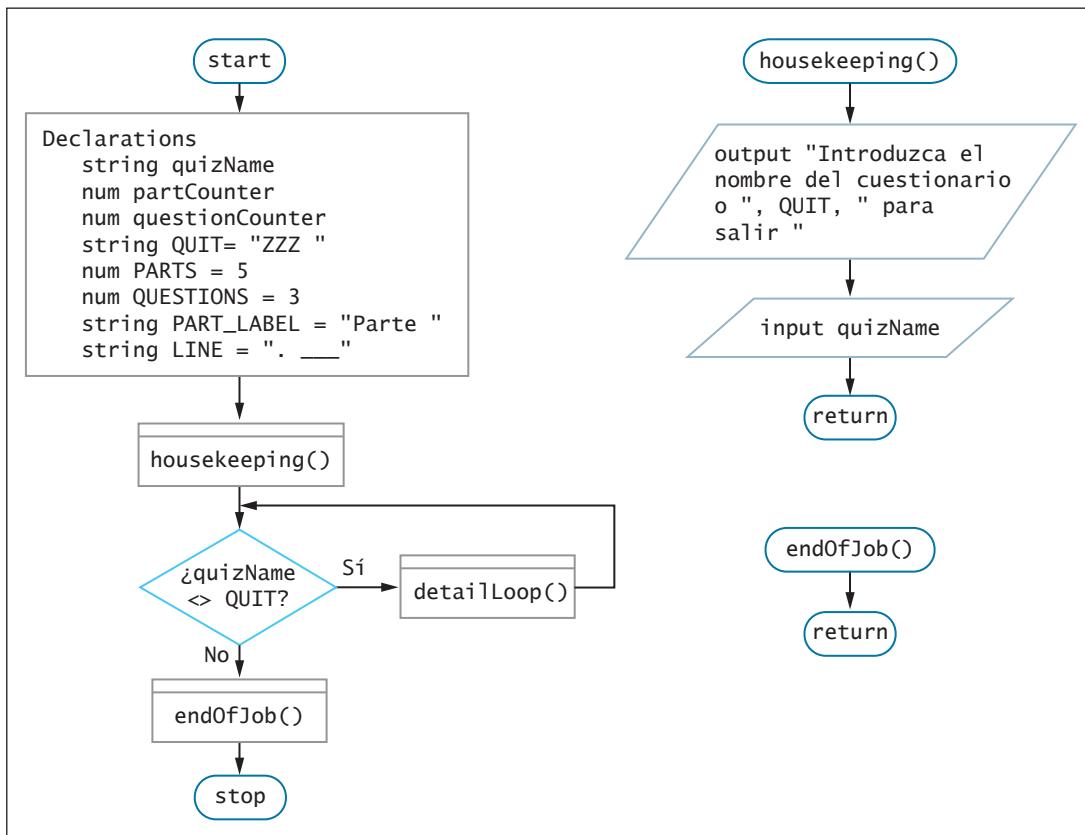
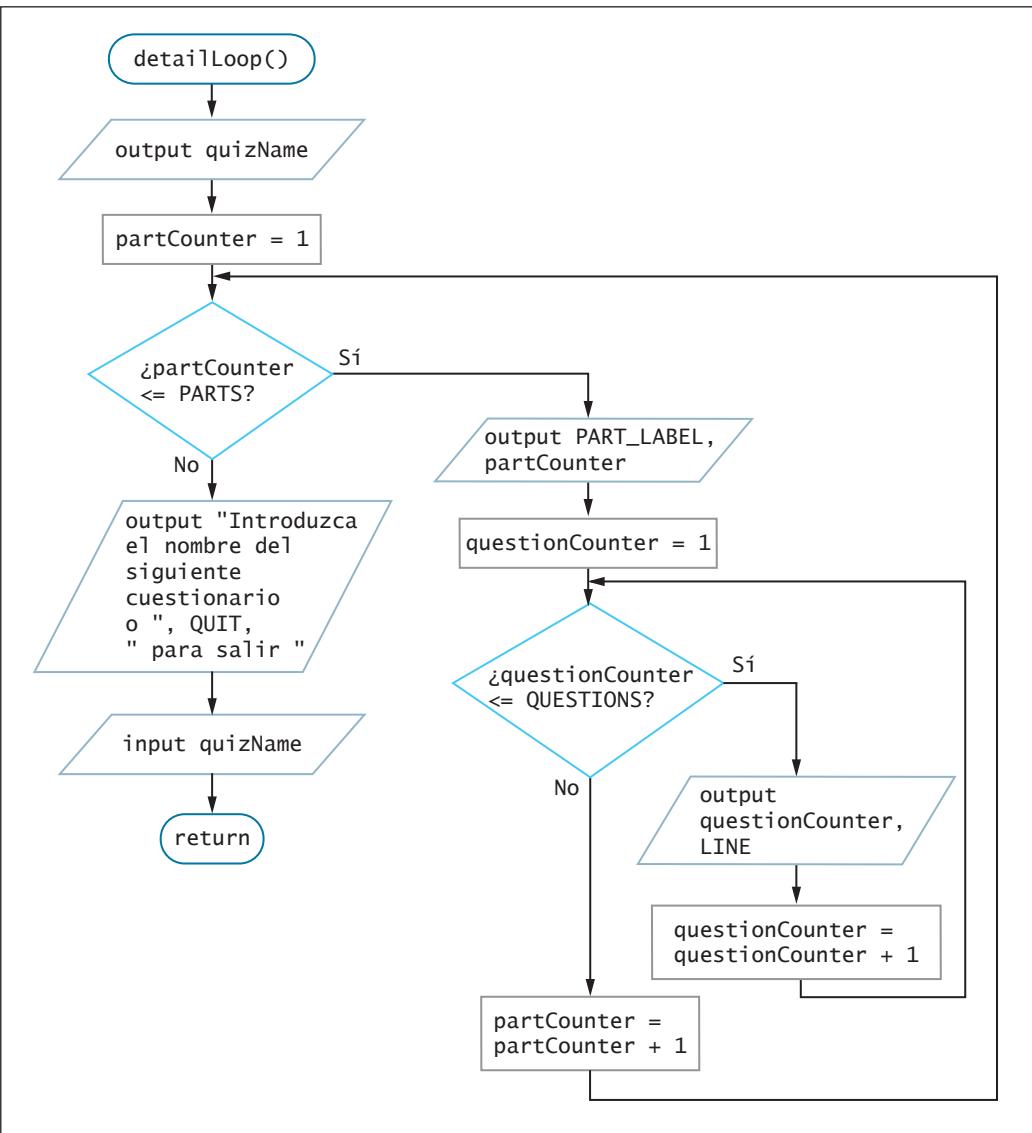


Figura 5-8 Diagrama de flujo y seudocódigo para el programa AnswerSheet (continúa)

(continuación)

180

**Figura 5-8** Diagrama de flujo y seudocódigo para el programa AnswerSheet (continúa)

(continuación)

```

start
Declarations
    string quizName
    num partCounter
    num questionCounter
    string QUIT = "ZZZ "
    num PARTS = 5
    num QUESTIONS = 3
    string PART_LABEL = "Parte "
    string LINE = ". _____"
housekeeping()
while quizName <> QUIT
    detailLoop()
endwhile
endOfJob()
stop

housekeeping()
    output "Introduzca el nombre del cuestionario o ",
    QUIT, " para salir " input quizName
return

detailLoop()
    output quizName
    partCounter = 1
    while partCounter <= PARTS
        output PART_LABEL, partCounter
        questionCounter = 1
        while questionCounter <= QUESTIONS
            output questionCounter, LINE
            questionCounter = questionCounter + 1
        endwhile
        partCounter = partCounter + 1
    endwhile
    output "Introduzca el siguiente nombre de cuestionario o ", QUIT,
    " para salir "
    input quizName
return

endOfJob()
return

```

Figura 5-8 Diagrama de flujo y seudocódigo para el programa AnswerSheet

En la figura 5-8, alguna salida (el indicador del usuario) se enviará a un dispositivo de salida, como un monitor. Otra salida (la hoja de respuestas) se enviaría a otro dispositivo, digamos, una impresora. Las declaraciones necesarias para enviar la salida hacia los dispositivos separados difieren entre los lenguajes. En el capítulo 7 se proporcionan más detalles.



El módulo `endOfJob()` se incluye en el programa de la figura 5-8 aun cuando no contiene declaraciones, de modo que la lógica de línea principal contiene todas las partes que usted ha aprendido. Un módulo vacío que actúa como un marcador se llama **stub**.

En el ciclo exterior en el módulo `detailLoop()` en la figura 5-8, la palabra *Parte* y el valor actual de `partCounter` son la salida. Luego se ejecutan los siguientes pasos:

- Se inicializa la variable de control de ciclo para el que es interior estableciendo `questionCounter` en 1.
- Se evalúa la variable de control de ciclo `questionCounter` comparándola con `QUESTIONS` y mientras `questionCounter` no exceda a `QUESTIONS` se ejecuta el cuerpo del ciclo: se da salida al valor de `questionCounter`, seguido por un punto y una línea para respuesta.
- Al final del cuerpo del ciclo se altera la variable de control de ciclo agregando 1 a `questionCounter` y se hace de nuevo la comparación `questionCounter`.

En otras palabras, cuando `partCounter` es 1, se da salida al encabezado de la parte y a las líneas de respuesta para las preguntas 1, 2 y 3. Entonces `partCounter` se vuelve 2, se da salida al encabezado de la parte y se crean líneas de respuesta para otro conjunto de preguntas 1, 2 y 3. Luego `partCounter` se vuelve 3, 4 y 5 por turno y se crean tres líneas de respuesta para cada parte.

En el programa de la figura 5-8 es importante que `questionCounter` se restablezca en 1 dentro del ciclo exterior, justo antes de entrar al ciclo interior. Si este paso se omitiera, la parte 1 contendría las preguntas 1, 2 y 3, pero las partes subsiguientes estarían vacías.

DOS VERDADES Y UNA MENTIRA

Ciclos anidados

1. Cuando un ciclo está anidado dentro de otro, el que contiene al otro se llama ciclo exterior.
2. Usted necesita crear ciclos anidados cuando los valores de dos o más variables se repiten para producir cada combinación de valores.
3. El número de veces que se ejecuta un ciclo siempre depende de una constante.

La afirmación falsa es la número 3. El número de veces que se ejecuta un ciclo puede depender de una constante o de un valor que varía.

Evitar errores comunes en los ciclos

Los programadores cometan los siguientes errores comunes:

- Descuidar la inicialización de la variable de control de ciclo
- Descuidar la alteración de la variable de control de ciclo
- Usar la comparación errónea con la variable de control de ciclo
- Incluir dentro del ciclo declaraciones que pertenecen al exterior del mismo

Las siguientes secciones explican estos errores comunes con más detalle.

Error: descuidar la inicialización de la variable de control de ciclo

Dejar de inicializar una variable de control de ciclo es un error. Por ejemplo, considere el programa en la figura 5-9. Indica al usuario que introduzca un nombre y mientras el valor de name continúa sin ser el valor centinela ZZZ, da salida a un saludo que usa el nombre y pide el siguiente nombre. Este programa funciona en forma correcta.

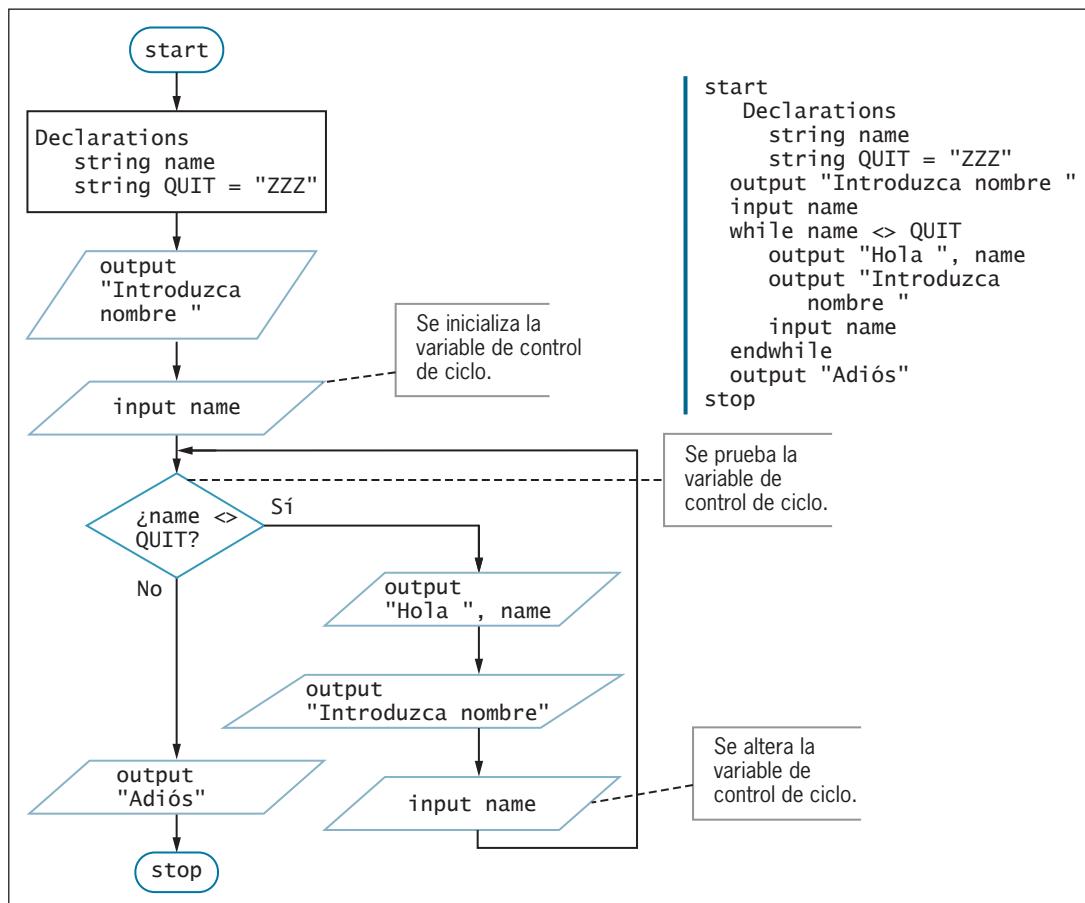


Figura 5-9 Lógica correcta para el programa de saludo

La figura 5-10 muestra un programa incorrecto en el que no se asigna un valor inicial a la variable de control de ciclo. Si la variable `name` no se establece como un valor inicial, entonces cuando se pruebe la condición `eof` no hay forma de predecir si será verdadera. Si el usuario no introduce un valor para `name`, el valor basura que se tiene originalmente para esa variable podría ser `ZZZ` o no podría serlo. Así, ocurre uno de dos escenarios:

- Con mayor probabilidad, el valor no inicializado de `name` no es `ZZZ`, de modo que la primera salida de saludo incluirá basura; por ejemplo, *Hola 12BGr5*.
- Por una remota posibilidad, el valor no inicializado de `name` es `ZZZ`, así que el programa termina inmediatamente antes que el usuario pueda introducir cualquier nombre.

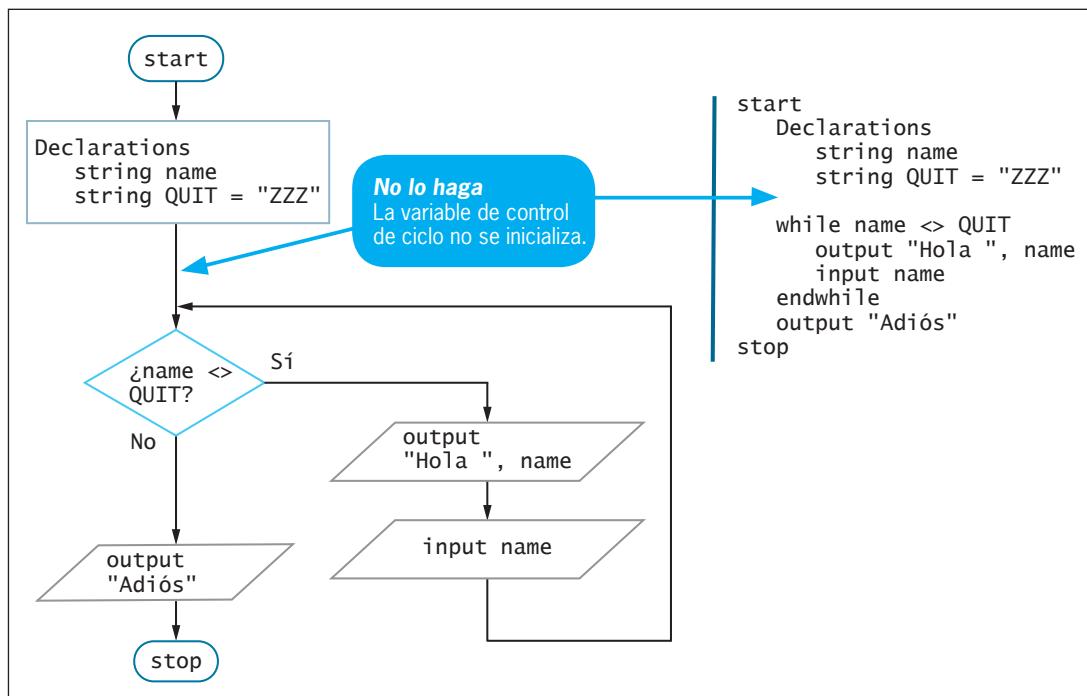


Figura 5-10 Lógica incorrecta para el programa de saludo debido a que falta la inicialización de la variable de control de ciclo

Error: descuidar la alteración de la variable de control de ciclo

Ocurrirán diferentes clases de errores si falla al alterar una variable de control de ciclo dentro de este último. Por ejemplo, en el programa de la figura 5-9 que acepta y despliega nombres, usted crea un error si no acepta nombres dentro del ciclo. La figura 5-11 muestra la lógica incorrecta resultante.

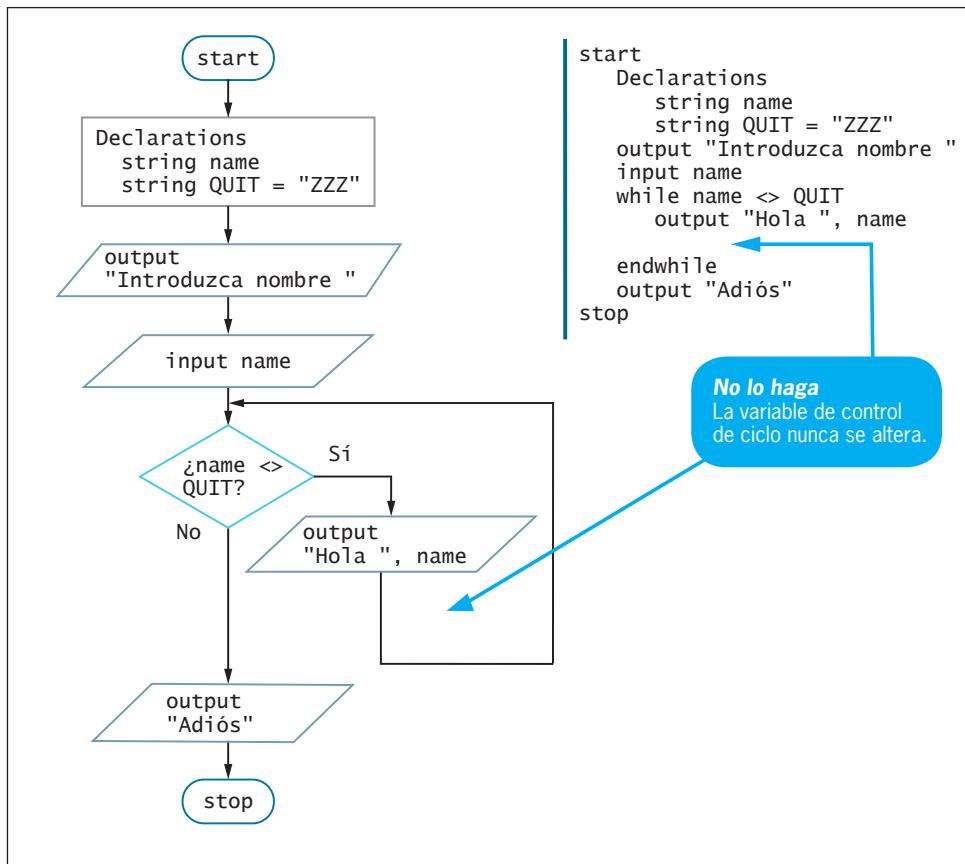


Figura 5-11 Lógica incorrecta para el programa de saludo debido a que no se altera la variable de control de ciclo

Si usted elimina la instrucción `input name` del final del ciclo en el programa, ningún nombre se introduce alguna vez después del primero. Por ejemplo, suponga que cuando inicia el programa, el usuario introduce *Fred*. El nombre será comparado con el valor centinela, y entrará al ciclo. Después que se dé salida a un saludo para *Fred*, ningún nombre nuevo se introduce, así que cuando la lógica regresa a la pregunta que controla al ciclo, `name` todavía no será `ZZZ`, y el saludo para *Fred* continuará desplegándose infinitamente. Usted nunca deseará crear un ciclo que no pueda terminar.

Error: usar la comparación errónea con la variable de control de ciclo

Los programadores deben tener cuidado de usar la comparación correcta en la declaración que controla un ciclo. Una comparación es correcta sólo cuando se usan los operandos y el operador apropiados. Por ejemplo, aunque sólo difiere un golpe de tecla entre el programa de saludo original en la figura 5-9 y el de la figura 5-12, el programa original genera correctamente saludos con nombre y el segundo no.

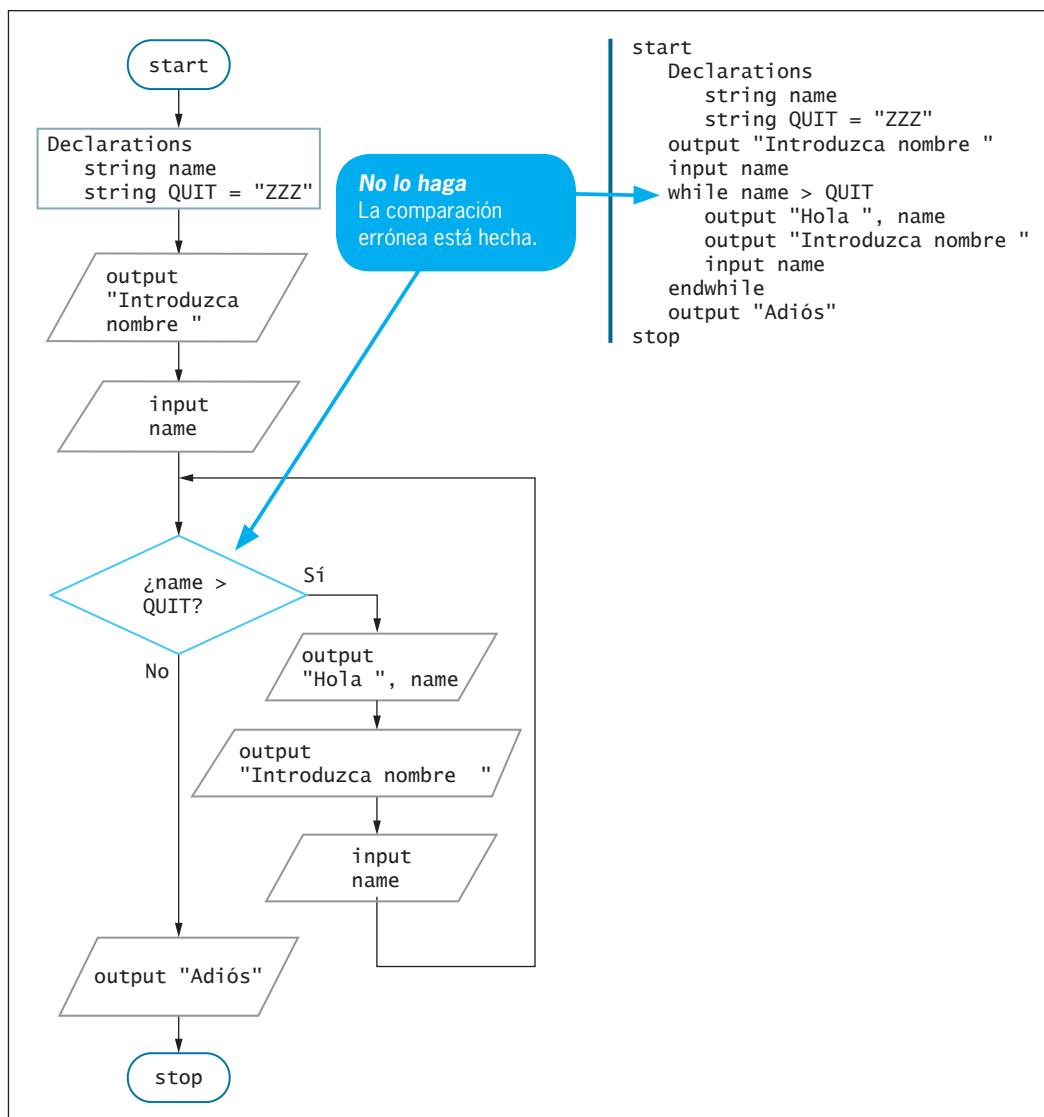


Figura 5-12 Lógica incorrecta para el programa de saludo debido a que se hace la prueba errónea con la variable de control de ciclo

En la figura 5-12 se hace una comparación mayor que ($>$) en lugar de una comparación no igual a ($<>$). Suponga que cuando se ejecuta el programa, el usuario introduce *Fred* como el primer nombre. En la mayoría de los lenguajes de programación, cuando se hace la comparación entre *Fred* y *ZZZ*, los valores se comparan alfabéticamente. *Fred* no es mayor que *ZZZ*, así que nunca se entra al ciclo y el programa termina.

Usar la comparación errónea puede tener efectos graves; por ejemplo, en un ciclo contado, si usted usa \leq en lugar de $<$ para comparar un contador con un valor centinela, el programa realizará una ejecución del ciclo una vez más. Si el ciclo sólo despliega saludos, el error quizás no sea grave, pero si éste ocurre en una aplicación de una compañía de préstamos, a cada cliente podría cargársele un interés mensual adicional. Si el error se presenta en una aplicación de una línea aérea, podría aceptar un exceso de reservaciones para un vuelo. Si el error se presenta en una aplicación expendedora de fármacos en una farmacia, cada paciente podría recibir una unidad extra (y posiblemente dañina) de medicamentos.

Error: incluir dentro del ciclo declaraciones que pertenecen al exterior del mismo

Suponga que escribe un programa para el gerente de una tienda que desea otorgar un descuento de 30% en todos los artículos que vende. El gerente desea 100 etiquetas adhesivas con los precios nuevos para cada artículo. El usuario introduce un precio, se calcula el nuevo precio descontado, se imprimen 100 etiquetas adhesivas y se introduce el siguiente precio. La figura 5-13 muestra un programa que ejecuta el trabajo en forma ineficiente debido a que el mismo valor, *newPrice*, se calcula 100 veces separadas para cada *price* que se introduce.

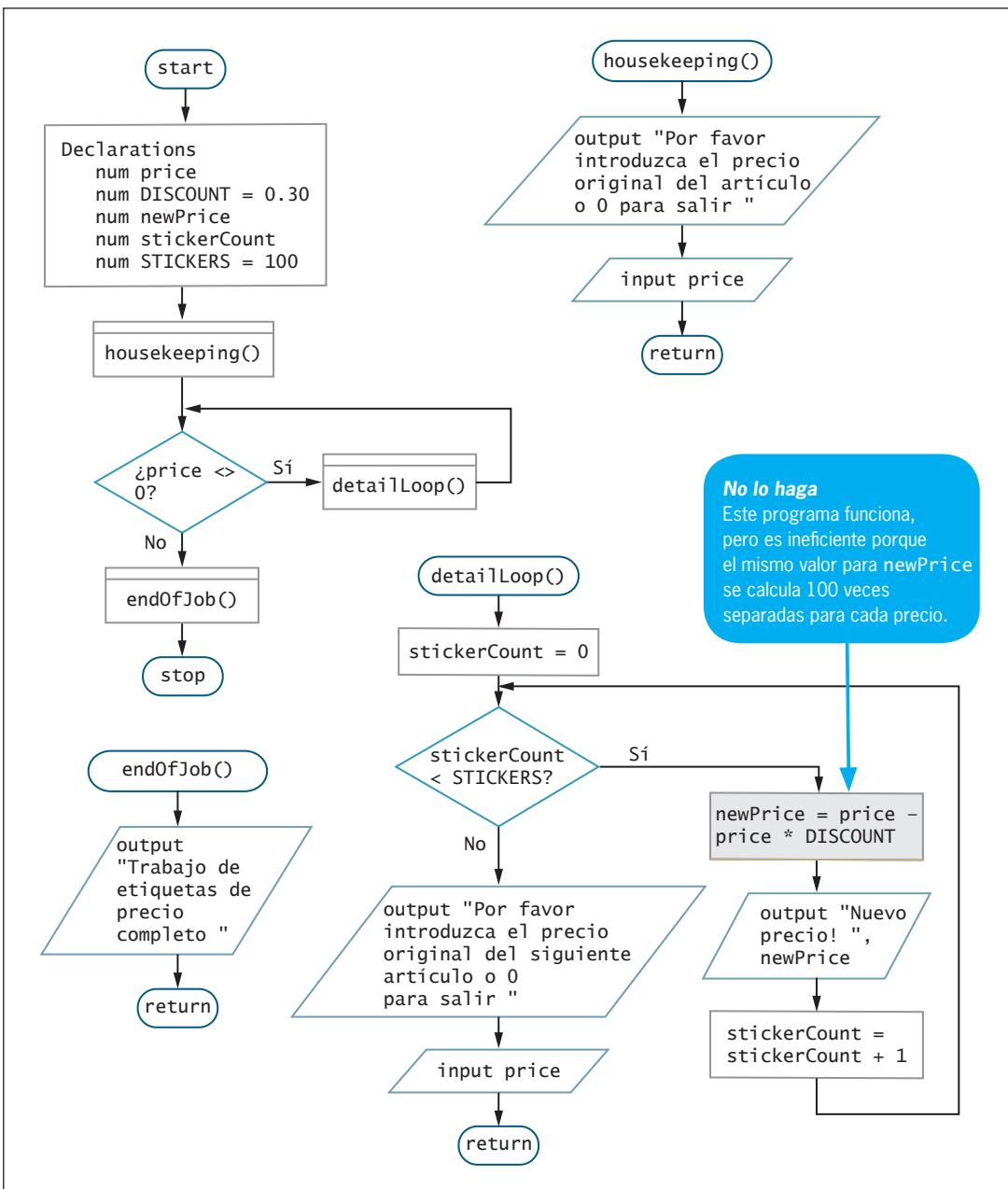


Figura 5-13 Forma ineficiente de producir 100 etiquetas adhesivas de los precios con descuento para los artículos con diferentes precios (continúa)

(continuación)

```

start
Declarations
    num price
    num DISCOUNT = 0.30
    num newPrice
    num stickerCount
    num STICKERS = 100
housekeeping()
while price <> 0
    detailLoop()
endwhile
endOfJob()
stop

housekeeping()
    output "Por favor introduzca el precio original del artículo
        o 0 para salir "
    input price
return

detailLoop()
    stickerCount = 0
    while stickerCount < STICKERS
        newPrice = price - price * DISCOUNT ←
            output "¡Nuevo precio! ", newPrice
            stickerCount = stickerCount + 1
    endwhile
    output "Por favor introduzca el precio original
        del siguiente artículo o 0 para salir "
    input price
return

endOfJob()
    output "Trabajo de etiquetas de precio completo"
return

```

No lo haga

Este programa funciona, pero es ineficiente porque el mismo valor para `newPrice` se calcula 100 veces separadas para cada precio.

Figura 5-13 Forma ineficiente de producir 100 etiquetas adhesivas de los precios con descuento para los artículos con diferentes precios

La figura 5-14 muestra el mismo programa, en el que el valor `newPrice` al que se da salida en la etiqueta se calcula sólo una vez por cada precio nuevo; el cálculo se ha movido a una mejor ubicación. El programa en las figuras 5-13 y 5-14 hace lo mismo, pero el segundo lo hace de manera más eficiente. Conforme se vuelva más competente en la programación, usted reconocerá muchas oportunidades para realizar las mismas tareas en formas alternas más elegantes y eficientes.



Cuando se describe a las personas o eventos como elegantes, significa que poseen una gracia refinada. Del mismo modo, los programadores usan el término *elegante* para describir programas que están bien diseñados y son fáciles de entender y mantener.

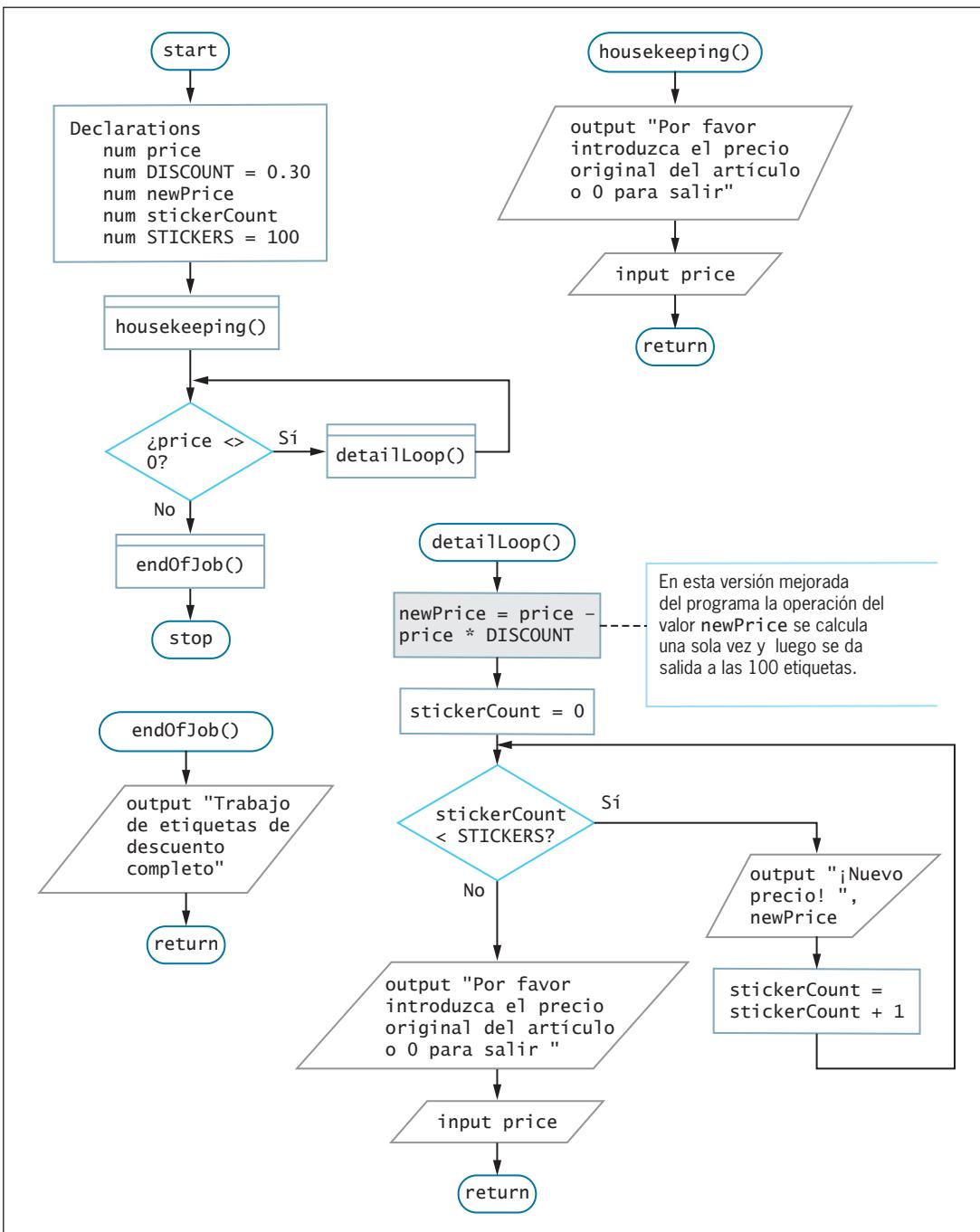


Figura 5-14 Programa mejorado de elaboración de etiquetas de descuento (continúa)

(continuación)

```

start
    Declarations
        num price
        num DISCOUNT = 0.30
        num newPrice
        num stickerCount
        num STICKERS = 100
    housekeeping()
    while price <> 0
        detailLoop()
    endwhile
    endOfJob()
stop

housekeeping()
    output "Por favor introduzca el precio original del
    artículo o 0 para salir "
return

detailLoop()
    newPrice = price - price * DISCOUNT ----
    stickerCount = 0
    while stickerCount < STICKERS
        output "¡Nuevo precio! ", newPrice
        stickerCount = stickerCount + 1
    endwhile
    output "Por favor introduzca el precio original del siguiente
    artículo o 0 para salir "
    input price
return

endOfJob()
    output "Trabajo de etiquetas de descuento completo"
return

```

En esta versión mejorada del programa, la operación del valor newPrice se calcula una sola vez, y luego se les da salida a las 100 etiquetas.

Figura 5-14 Programa mejorado de elaboración de etiquetas de descuento

DOS VERDADES Y UNA MENTIRA

Evitar errores comunes en los ciclos

1. En un ciclo, descuidar la inicialización de la variable de control de ciclo es un error.
2. En un ciclo, descuidar la alteración de la variable de control de ciclo es un error.
3. En un ciclo, es un error comparar la variable de control de ciclo usando $>=$ o $<=$.

usando $>=$ o $<=$.

La afirmación falsa es la número 3. Muchos ciclos se crean en forma correcta

Uso de un ciclo for

Todo lenguaje de programación de alto nivel contiene una declaración `while` que usted puede usar para codificar cualquier ciclo, incluyendo los indefinidos y definidos. Además de la declaración `while`, la mayoría de los lenguajes de computadora soportan una declaración `for`. Por lo general se usa la **declaración for**, o **ciclo for**, con los ciclos definidos (aquellos que se repiten un número específico de veces) cuando usted sabe con exactitud cuántas veces se repetirá el ciclo. La declaración `for` le proporciona tres acciones en una declaración compacta. En una declaración `for`, una variable de control de ciclo:

- Se inicializa
- Se evalúa
- Se altera

La declaración `for` adopta una forma similar a la siguiente:

```
for loopControlVariable = initialValue to finalValue step stepValue
    do something
endfor
```

La cantidad por la que una variable de control del ciclo `for` cambia con frecuencia se llama **valor de paso**. El valor de paso puede ser positivo o negativo; es decir, puede incrementarse o decrementarse.

Por ejemplo, para desplegar *Hola* cuatro veces, usted puede escribir cualquiera de las series de declaraciones en la figura 5-15.

```
count = 0
while count <= 3
    output "Hola"
    count = count + 1
endwhile
```

```
for count = 0 to 3 step 1
    output "Hola"
endfor
```

Figura 5-15 Declaraciones `while` y `for` comparables, cada una de ellas produce *Hola* cuatro veces

Cada uno de los segmentos de código en la figura 5-15 lleva a cabo las mismas tareas:

- La variable `count` se inicializa en 0.
- La variable `count` se compara con el valor límite 3; mientras `count` es menor o igual que 3, el cuerpo del ciclo se ejecuta.
- Como la última declaración en la ejecución del ciclo, el valor de `count` se incrementa en 1. Después del incremento, se hace de nuevo la comparación con el valor límite.

El ciclo `for` simplemente expresa la misma lógica en una forma más compacta que la declaración `while`. Nunca se requiere una declaración `for` para cualquier ciclo; siempre puede usarse en su lugar una `while`. Sin embargo, cuando la ejecución de un ciclo se basa en una variable de control de ciclo progresiva desde un valor inicial conocido hasta uno final conocido en pasos iguales, el ciclo `for` proporciona una abreviatura conveniente. Es fácil de leer para otras

personas, y debido a que la inicialización, prueba y alteración de la variable de control de ciclo se ejecutan todas en una ubicación, es menos probable que deje fuera uno de estos elementos cruciales.

Aunque los ciclos **for** se usan por lo común para controlar la ejecución de un bloque de declaraciones un número fijo de veces, el programador no necesita conocer el valor inicial, final o de paso para el ciclo cuando se escribe el programa. Por ejemplo, cualquiera de los valores podría ser introducido por el usuario o el resultado de un cálculo.

El ciclo **for** es útil en particular cuando se procesan arreglos. Aprenderá sobre arreglos en el capítulo 6.



En Java, C++ y C#, un ciclo **for** que despliega 21 valores (0 a 20) podría verse como el siguiente:

```
for(count = 0; count <= 20; count++)  
{  
    output count  
}
```

Las tres acciones (inicialización, comparación y alteración de la variable de control de ciclo) están separadas por punto y coma dentro de un conjunto de paréntesis que siguen a la palabra clave **for**. La expresión **count++** agrega 1 a **count**. El bloque de declaraciones que depende del ciclo se encuentra entre un par de llaves.

Tanto **while** como **for** son ejemplos de ciclos preprueba. En un **ciclo preprueba**, la variable de control de ciclo se prueba antes de cada iteración. Esto significa que el cuerpo del ciclo podría nunca ejecutarse debido a que la pregunta que controla el ciclo quizás sea falsa la primera vez que se hace. La mayor parte de los lenguajes permiten usar una variación de la estructura conocida como **ciclo posprueba**, que prueba la variable de control de ciclo después de cada iteración. En un ciclo posprueba, el cuerpo del ciclo se ejecuta al menos una vez porque la variable de control de ciclo se prueba hasta después de una iteración. El apéndice F contiene información sobre los ciclos posprueba.



Algunos libros y programas de diagramas de flujo usan un símbolo que parece un hexágono para representar un ciclo **for** en un diagrama de flujo. Sin embargo, no se necesitan símbolos especiales para expresar la lógica de un ciclo **for**. Un ciclo **for** tan solo es un atajo del código, así que este libro usa símbolos estándar de diagrama de flujo para representar la inicialización, la prueba y la alteración de la variable de control de ciclo.

DOS VERDADES Y UNA MENTIRA

Uso de un ciclo for

1. La declaración `for` le proporciona tres acciones en una declaración compacta: inicialización, evaluación y alteración de una variable de control del ciclo.
2. El cuerpo de una declaración `for` siempre se ejecuta al menos una vez.
3. En la mayor parte de los lenguajes de programación, puede proporcionar un ciclo `for` con cualquier valor de paso.

La afirmación falsa es la número 2. El cuerpo de una declaración `for` podría no ejecutarse dependiendo del valor inicial en la variable de control de ciclo.

Aplicaciones comunes de los ciclos

Aunque cada programa de computadora es diferente muchas técnicas son comunes a diversas aplicaciones. Los ciclos, por ejemplo, se usan con frecuencia para acumular totales y validar datos.

Uso de un ciclo para acumular totales

Los informes de negocios con frecuencia incluyen totales. El supervisor que solicita una lista de empleados en el plan dental de la compañía en general está tan interesado en el número de empleados participantes como en quiénes son. Cuando usted recibe su factura telefónica cada mes, es común que revise el total lo mismo que los cargos por las llamadas individuales.

Suponga que un corredor de bienes raíces desea ver una lista de todas las propiedades vendidas en el último mes al igual que el valor total de todas las propiedades. Un programa podría aceptar datos de ventas que incluyan el domicilio de cada propiedad vendida y su precio. Un empleado podría introducir los registros de datos conforme se hace cada venta y almacenarlos en un archivo hasta el final del mes; luego pueden usarse en un informe mensual. La figura 5-16 muestra un ejemplo de un informe así.

INFORME DE VENTAS DE FIN DE MES	
Domicilio	Precio
287 Acorn St	150,000
12 Maple Ave	310,000
8723 Marie Ln	65,500
222 Acorn St	127,000
29 Bahama Way	450,000
Total	1,102,500

Figura 5-16 Informe de ventas de bienes raíces de fin de mes

Para crear el informe de ventas debe darse salida al domicilio y precio de cada propiedad vendida y sumar su valor en un acumulador. Un **acumulador** es una variable que se usa para recopilar o acumular valores, y es muy similar a un contador que se utiliza para contar las iteraciones de un ciclo. Sin embargo, a un contador por lo general sólo se le suma uno, mientras que a un acumulador se le suma algún otro valor. Si el corredor de bienes raíces desea saber cuántos listados tiene la compañía, usted los *cuenta*; cuando el corredor desea saber el valor total de los bienes raíces, usted los *acumula*.

Para acumular el total de los precios de bienes raíces, usted declara una variable numérica como `accumPrice` y la inicializa en 0. Conforme obtiene datos para cada transacción de bienes raíces, da salida y agrega su valor al acumulador `accumPrice`, como se muestra en la parte sombreada en la figura 5-17.

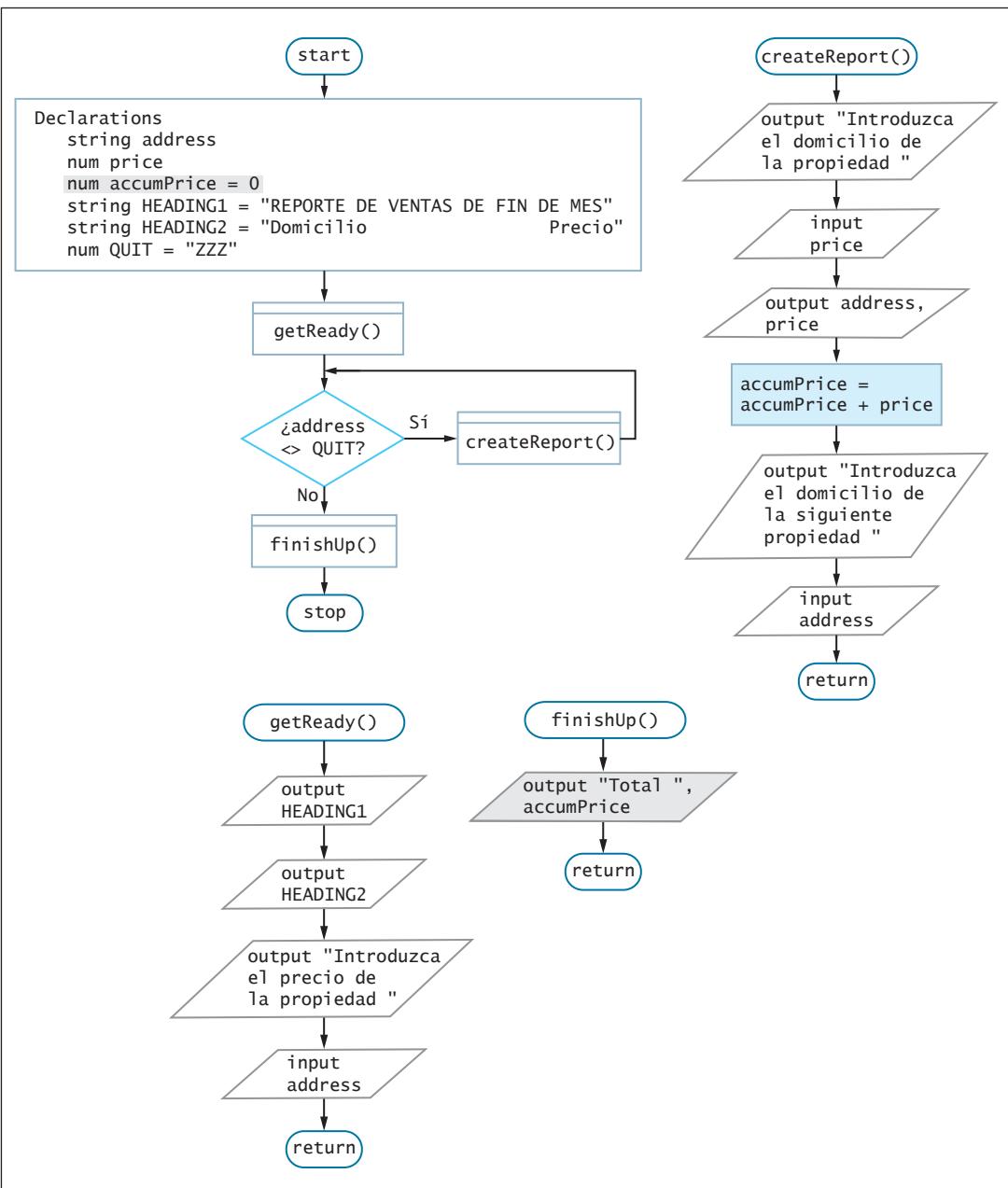


Figura 5-17 Diagrama de flujo y seudocódigo para el programa de informe de ventas de bienes raíces (continúa)

(continuación)

```

start
Declarations
    string address
    num price
    num accumPrice = 0
    string HEADING1 = "REPORTE DE VENTAS DE FIN DE MES"
    string HEADING2 = "Domicilio          Precio"
    num QUIT = "ZZZ"
getReady()
while address <> QUIT
    createReport()
endwhile
finishUp()
stop

getReady()
    output HEADING1
    output HEADING2
    output "Introduzca el domicilio de la propiedad "
    input address
return

createReport()
    output "Introduzca el precio de la propiedad "
    input price
    output address, price
    accumPrice = accumPrice + price
    output "Introduzca el domicilio de la siguiente propiedad "
    input address
return

finishUp()
    output "Total ", accumPrice
return

```

Figura 5-17 Diagrama de flujo y seudocódigo para el programa de informe de ventas de bienes raíces



Algunos lenguajes de programación asignan 0 a una variable que usted no inicializa en forma explícita, pero muchos no lo hacen; cuando trata de agregar un valor a una variable no inicializada emiten un mensaje de error o permiten iniciar de manera incorrecta con un acumulador que contiene basura. El curso de acción más seguro y claro es asignar el valor 0 a los acumuladores antes de usarlos.

Después de que el programa en la figura 5-17 obtiene y despliega la última transacción de bienes raíces, el usuario introduce el valor centinela y la ejecución del ciclo termina. En este punto, el acumulador contendrá el gran total de todos los valores de bienes raíces. El programa despliega la palabra *Total* y el valor acumulado *accumPrice*. Luego el programa termina.

La figura 5-17 resalta las tres acciones que por lo general usted debe realizar con un acumulador:

- Se inicializan en 0.
- Son alterados, por lo general una vez por cada conjunto de datos procesado.
- Al final del procesamiento, los acumuladores dan salida.

198

Después de dar salida al valor de `accumPrice`, los programadores principiantes con frecuencia desean reiniciarlo en 0. Su argumento es que están “limpiando después de ellos”. Aunque usted puede dar este paso sin dañar la ejecución del programa no sirve para ningún propósito útil. No puede establecer `accumPrice` en 0 antes de tenerlo listo para el siguiente programa, o incluso para la siguiente vez que ejecute el mismo programa. Las variables existen sólo durante una ejecución, e incluso si sucede que una aplicación futura contiene una variable llamada `accumPrice`, la variable no necesariamente ocupará la misma ubicación de memoria que ésta. Aun si usted ejecuta la misma aplicación por segunda vez las variables podrían ocupar ubicaciones de memoria física diferentes de aquellas que ocuparon durante la primera ejecución. Al principio de cualquier método, es responsabilidad del programador inicializar todas las variables que deben empezar con un valor específico. No hay ningún beneficio en cambiar el valor de una variable cuando nunca se usará de nuevo durante la ejecución actual.

Algunos informes de negocios son **informes sumarios**; sólo contienen totales sin datos para los registros individuales. En el ejemplo de la figura 5-17, suponga que al corredor no le importan los detalles de las ventas individuales, sino sólo el total de todas las transacciones. Usted podría crear un informe sumario omitiendo el paso que da salida a `address` y `price` desde el módulo `createReport()`. Entonces podría simplemente dar salida a `accumPrice` al final del programa.

Uso de un ciclo para validar datos

Cuando usted pide a un usuario que introduzca datos en un programa de computadora no tiene la seguridad de que éstos serán precisos. Las entradas incorrectas de los usuarios son con mucho la fuente de errores más común. Los programas que usted escribe mejorarán si emplea la **programación defensiva**, lo que significa tratar de prepararse para todos los errores posibles antes de que ocurran. Los ciclos con frecuencia se usan para **validar datos**; es decir, asegurarse de que sean significativos y útiles. Por ejemplo, la validación aseguraría que un valor es el tipo de dato correcto o que se ubica dentro de un rango aceptable.

Suponga que parte de un programa que usted escribe le pide a un usuario que introduzca un número que represente su mes de nacimiento. Si el usuario teclea un número menor que 1 o mayor que 12, usted debe emprender alguna clase de acción. Por ejemplo:

- Desplegar un mensaje de error y detener el programa.
- Elegir la asignación de un valor predeterminado para el mes (por ejemplo, 1) antes de proceder.
- Pedir de nuevo al usuario una entrada válida.

Si elige este último curso de acción podría adoptar al menos dos enfoques: usar una selección y, si el mes es inválido, pedir al usuario que reintroduzca un número, como se muestra en la figura 5-18.

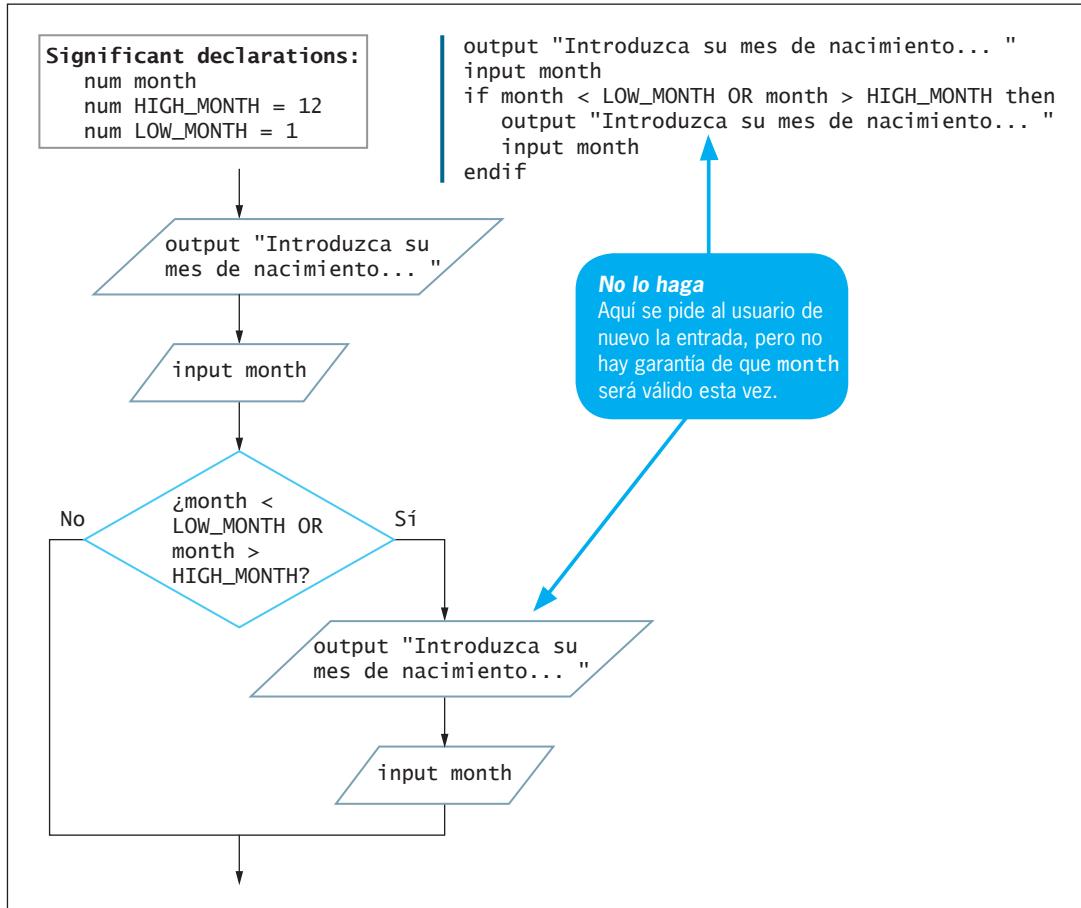


Figura 5-18 Pedir de nuevo una entrada al usuario una vez después que se ha introducido un mes inválido

El problema con la lógica en la figura 5-18 es que el usuario podría no introducir todavía datos válidos en el segundo intento. Por supuesto, sería posible que usted añadiera una tercera decisión, pero aún así no controlaría lo que el usuario introduce.

La mejor solución es usar un ciclo para indicar continuamente al usuario que introduzca un mes hasta que lo haga en forma correcta. La figura 5-19 muestra este enfoque.

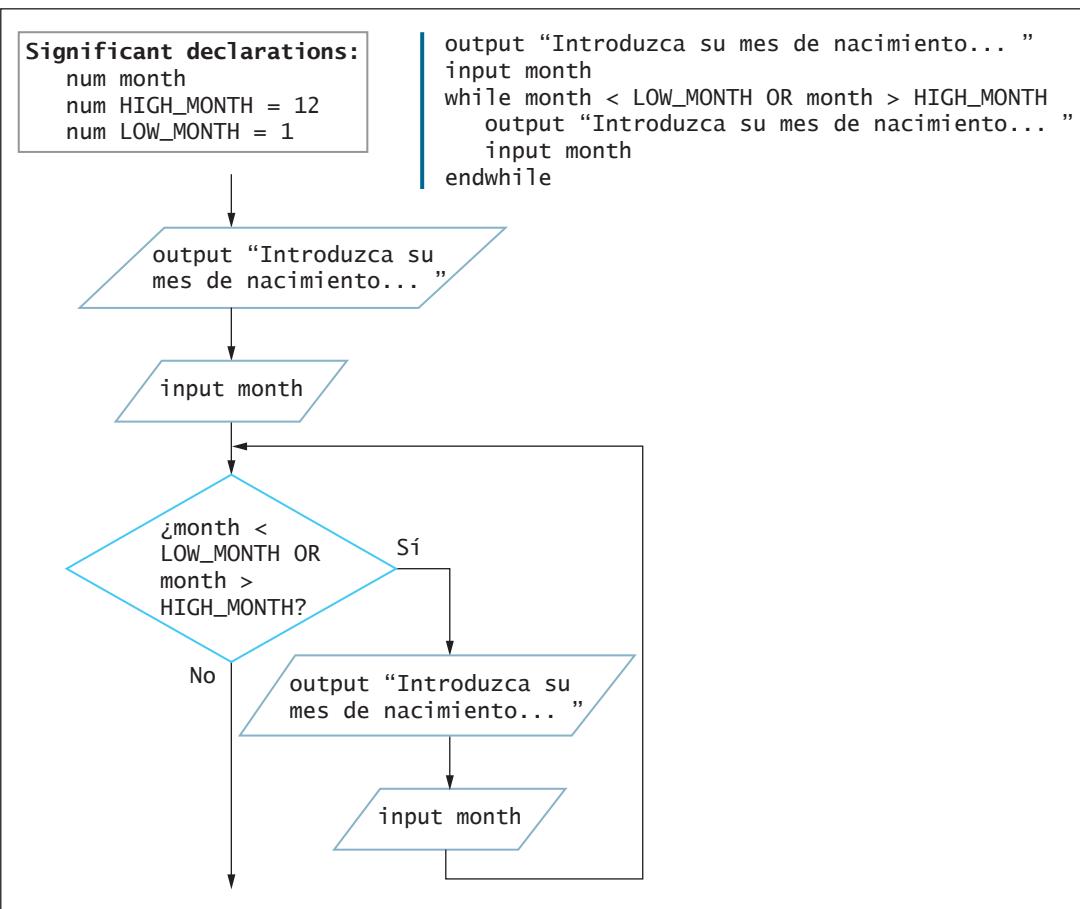


Figura 5-19 Al usuario se le indica continuamente después de que se introduce un mes inválido



La mayoría de los lenguajes proporcionan una forma incorporada para verificar si un valor introducido es numérico. Cuando usted depende de la entrada del usuario, con frecuencia acepta cada pieza de datos de entrada como una cadena y luego intenta convertirlos en un número. El procedimiento para realizar verificaciones numéricas varía ligeramente en los diferentes lenguajes de programación.

Por supuesto, la validación de datos no previene todos los errores; sólo porque un elemento de datos es válido no significa que sea correcto. Por ejemplo, un programa puede determinar que 5 es un mes de nacimiento válido, pero no que su cumpleaños caiga en realidad en el mes 5. Los programadores emplean las siglas **GIGO**, del inglés *garbage in, garbage out* (*entra basura, sale basura*): significa que si su entrada es incorrecta, su salida no tiene ningún valor.

Limitación de un ciclo que pide entradas de nuevo

La petición reiterada de entradas a un usuario es una buena forma para asegurar que los datos sean válidos, pero puede ser frustrante para éste si la situación continúa de manera indefinida. Por ejemplo, suponga que el usuario debe introducir un mes de nacimiento válido, pero ha

usado otra aplicación en la que enero era el mes 0 y sigue introduciendo 0 sin importar cuántas veces usted repita el indicador. Una adición útil al programa serían los valores limitantes como parte del indicador. En otras palabras, en lugar de la declaración de salida "Introduzca su mes de nacimiento...", la siguiente declaración sería más útil:

```
output "Introduzca su mes de nacimiento entre ", LOW_MONTH, " y " HIGH_MONTH,  
"..."
```

Aun así, el usuario quizá no entienda el indicador o no lo lea con cuidado, de modo que usted tal vez desee emplear la táctica que se usó en la figura 5-20, en la que el programa mantiene un conteo del número de repeticiones del indicador. En este ejemplo, una constante llamada ATTEMPTS se establece en 3. Mientras un conteo de los intentos del usuario de introducir datos correctos permanezca por debajo de este límite y el usuario introduzca datos inválidos se le sigue repitiendo el indicador; si excede el número limitado de intentos permitidos, el ciclo termina. La siguiente acción depende de la aplicación. Si count es igual a ATTEMPTS después de que termina el ciclo de entrada de datos, usted desearía forzar los datos inválidos a un valor predeterminado. **Forzar** un elemento de datos significa que usted anula los que son incorrectos estableciendo la variable en un valor específico. Por ejemplo, podría decidir que si un valor de mes no cae entre 1 y 12, forzaría el mes a 0 o 99, lo cual indica a los usuarios que no existe un valor válido. En una aplicación diferente, usted quizá elija tan solo terminar el programa. En un programa interactivo basado en la web, podría elegir que un representante de servicio al cliente inicie una sesión de chat con el usuario para ofrecerle ayuda.



Los programas que frustran a los usuarios pueden dar como resultado la pérdida de ingresos. Por ejemplo, si es difícil navegar en el sitio web de una compañía, los usuarios podrían darse por vencidos y no hacer negocios con ella.

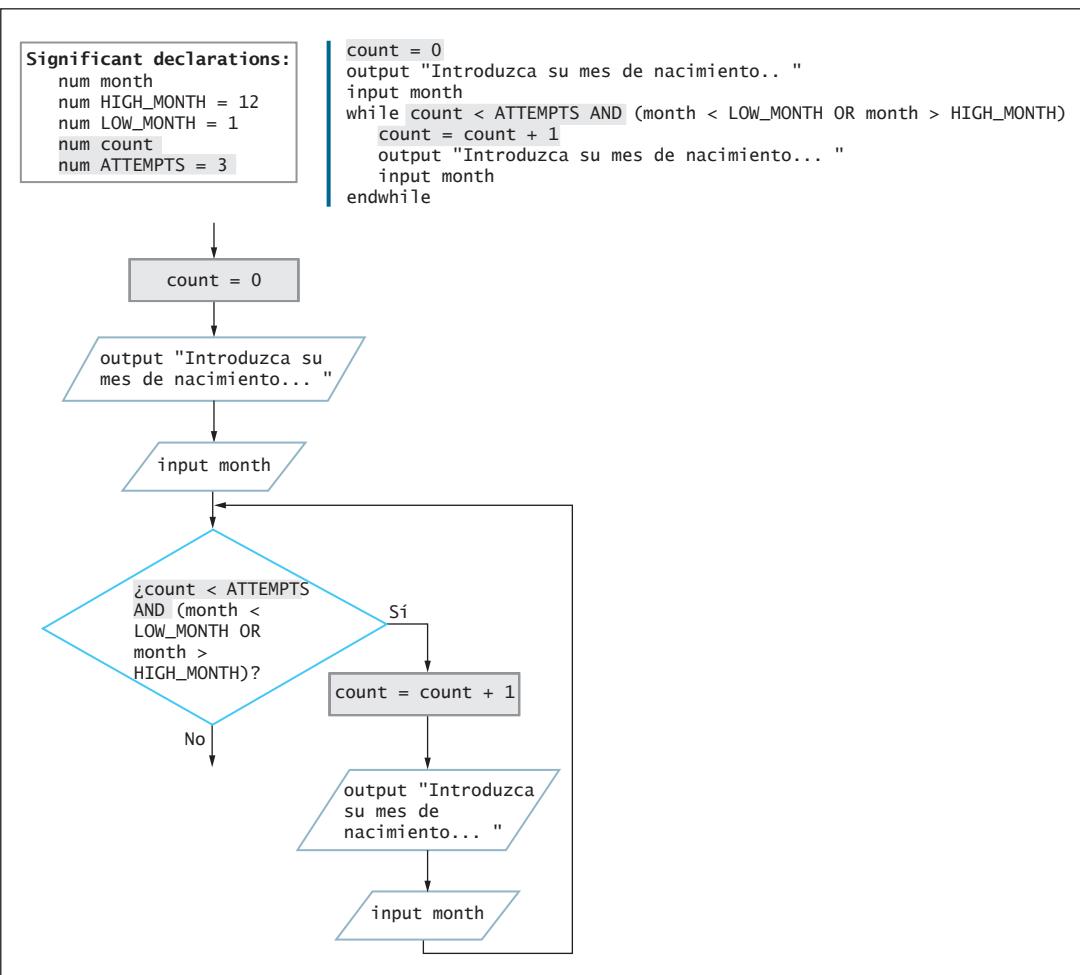


Figura 5-20 Limitación de la repetición de indicadores para el usuario

Validación de un tipo de datos

Los datos que usted usa en los programas de computadora son variados. Es razonable pensar que la validación de los datos requiere una diversidad de métodos. Por ejemplo, algunos lenguajes de programación permiten verificar los elementos de datos para asegurar que son del tipo correcto. Aunque esta técnica varía de un lenguaje a otro, con frecuencia usted puede hacer una declaración como la que se muestra en la figura 5-21. En este segmento de programa, `isNumeric()` representa una llamada a un módulo; se usa para comprobar si `salary` del empleado introducido se ubica dentro de la categoría de datos numéricos. Usted verifica para asegurarse de que un valor es numérico por muchas razones: una importante es que sólo los valores numéricos pueden usarse en forma correcta en las declaraciones aritméticas. Un módulo como `isNumeric()` se proporciona con más frecuencia con el traductor del lenguaje que usa para escribir sus programas. Dicho método opera como una caja negra; en otras palabras, usted puede usar los resultados del método sin entender sus declaraciones internas.

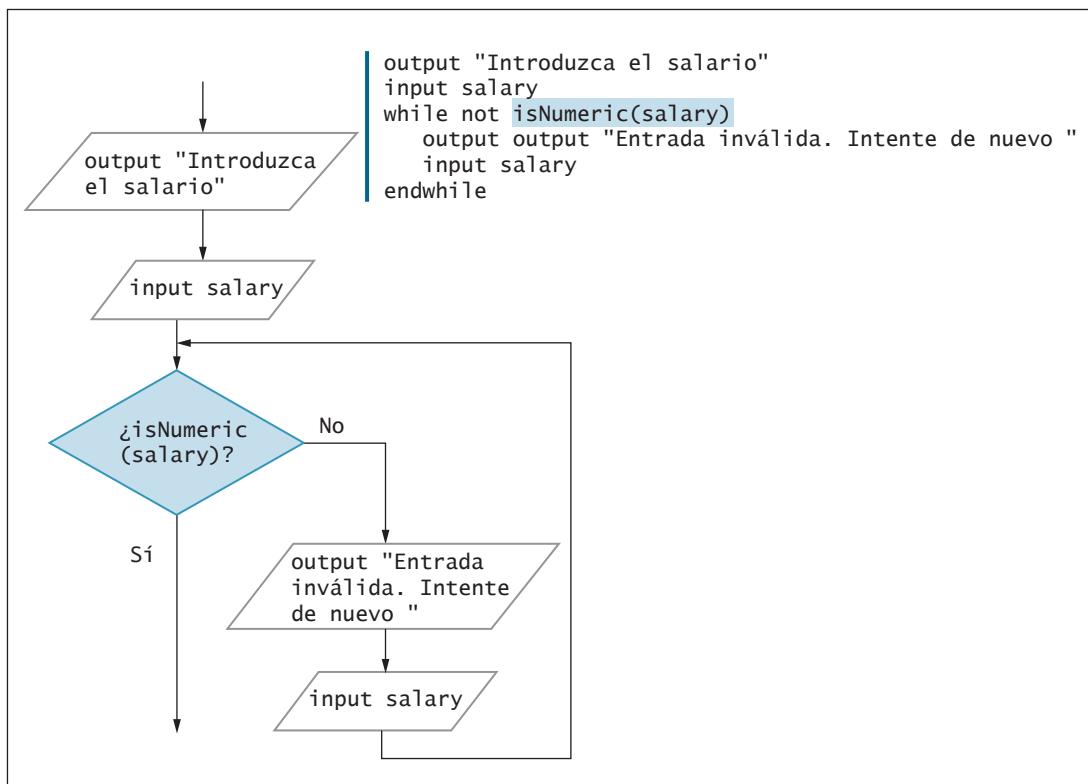


Figura 5-21 Verificación de los datos para ver si son del tipo correcto

Además de permitirle comprobar si un valor es numérico, algunos lenguajes contienen métodos como `isChar()`, que verifica si un valor es un tipo de datos carácter; `isWhitespace()`, que se asegura de que un valor sea un carácter que no se imprime, como un espacio o un tabulador; e `isUpper()`, que verifica si un valor es una letra mayúscula.

En muchos lenguajes, usted acepta todos los datos del usuario como una cadena de caracteres y luego usa métodos incorporados para convertirlos al tipo de datos correcto para su aplicación. Cuando los métodos de conversión tienen éxito, usted tiene datos útiles; cuando no es así se debe a que el usuario ha introducido el tipo de datos erróneo, usted puede emprender una acción apropiada, como emitir un mensaje de error, volver a pedir la entrada al usuario o forzar los datos a un valor predeterminado.

Validación de la sensatez y consistencia de los datos

Los elementos de datos pueden ser del tipo correcto y estar dentro del rango, pero aun así ser incorrectos. Usted ha experimentado este problema si alguien ha escrito mal su nombre o lo ha sobrefacturado: los datos podrían haber sido del tipo correcto; por ejemplo, se usaron letras alfabéticas en su nombre, pero el nombre en sí era incorrecto. No se puede comprobar

la sensatez de muchos elementos de datos; por ejemplo, los nombres Catherine, Katherine y Kathryn son igual de razonables, pero sólo una ortografía es correcta para una mujer particular.

Sin embargo, es posible verificar la sensatez de muchos elementos de datos; si usted hizo una compra el 3 de mayo de 2013, entonces no es posible que el pago se venza antes de esa fecha. Quizá dentro de su organización, no puede ganar más de 20 dólares por hora si trabaja en el Departamento 12. Si su código postal es 90201, su estado de residencia no puede ser Nueva York. Si su efectivo en caja en su tienda era de 3,000 dólares cuando cerró el martes, la cantidad no debería ser diferente cuando la tienda abra el miércoles. Si el título de una clienta es *señorita*, el género de la clienta debería ser *F*. Cada uno de estos ejemplos implica comparar la sensatez o consistencia de dos elementos de datos. Debería considerar hacer tantas de estas comparaciones como sea posible cuando escriba sus propios programas.

Con frecuencia, probar la sensatez y consistencia implica usar archivos de datos adicionales. Por ejemplo, para comprobar que un usuario ha entrado a un municipio válido de residencia para un estado, usted podría usar un archivo que contenga los nombres de todos los municipios dentro de cada estado en su país y verificar el municipio del usuario contra los que contiene el archivo.

Los buenos programas defensivos tratan de prever todas las inconsistencias y errores posibles; entre más precisos sean sus datos, será más útil la información que producirá como salida de sus programas.



Cuando se convierta en un programador profesional deseará que sus programas funcionen en forma correcta como una fuente de orgullo profesional. En un nivel más básico, no deseará que le llamen para trabajar a las 3:00 a.m. cuando la ejecución del turno nocturno de su programa falle debido a errores que usted creó.

DOS VERDADES Y UNA MENTIRA

Aplicaciones comunes de los ciclos

1. Un acumulador es una variable que se usa para recopilar o acumular valores.
2. Por lo común un acumulador se inicializa en 0.
3. Por lo común un acumulador se reinicia en 0 después de su salida.

La afirmación falsa es la número 3. Por lo común no hay necesidad de reiniciar un acumulador después de su salida.

Resumen del capítulo

- Un ciclo contiene un conjunto de instrucciones que opera en múltiples conjuntos de datos separados.
- En cualquier ciclo deben ocurrir tres pasos: es preciso inicializar una variable de control de ciclo, compararla con algún valor que controle si el ciclo continúa o se detiene, y alterar la variable que controla el ciclo.
- Cuando debe usar unos ciclos dentro de otros, use ciclos anidados. Cuando los anida debe mantener dos variables de control de ciclo individuales y alterar cada una en el momento apropiado.
- Los errores comunes que cometan los programadores cuando escriben ciclos incluyen el descuido en la inicialización, la alteración y la comparación errónea de la variable de control de ciclo e incluir declaraciones dentro del ciclo que pertenecen a su exterior.
- La mayoría de los lenguajes de computadora soportan una declaración **for** o un ciclo **for** que puede usar con ciclos definidos cuando sabe cuántas veces se repetirá uno de ellos. La declaración **for** usa una variable de control de ciclo que se inicializa, evalúa y altera de manera automática.
- Los ciclos se usan en muchas aplicaciones; por ejemplo, para acumular totales en los informes de negocios. Los ciclos también se usan para asegurar que las entradas de datos del usuario son válidas pidiendo a este último una entrada de manera continua.

Términos clave

Una **variable de control de ciclo** determina si un ciclo continuará.

Un **ciclo definido** tiene un número de repeticiones que es un valor predeterminado.

Un **ciclo contado**, o **ciclo controlado por contador**, tiene repeticiones que son manejadas por un contador.

Incrementar una variable es agregarle un valor constante, con frecuencia 1.

Decrementar una variable es disminuirla por un valor constante, con frecuencia 1.

Un **contador** es cualquier variable numérica que se use para contar las veces que ha ocurrido un evento.

En un **ciclo indefinido** usted no puede determinar el número de ejecuciones.

Los **ciclos anidados** ocurren cuando existe una estructura de ciclo dentro de otra.

Un **ciclo externo** contiene otro cuando ambos están anidados.

Un **ciclo interno** está contenido dentro de otro cuando ambos están anidados.

Un **stub** es un método sin declaraciones que se usa como marcador.

Una **declaración for**, o **ciclo for**, puede usarse para codificar ciclos definidos; contiene una variable de control de ciclo que se inicializa, evalúa y altera en forma automática.

Un **valor de paso** es un número que se usa para incrementar una variable de control de ciclo en cada paso a través de un ciclo.

Un **ciclo prueba** prueba su variable de control de ciclo antes de cada iteración, lo que significa que el cuerpo del ciclo podría no ejecutarse nunca.

Un **ciclo posprueba** prueba su variable de control de ciclo después de cada iteración, lo que significa que el cuerpo del ciclo se ejecuta al menos una vez.

Un **acumulador** es una variable que se usa para recopilar o acumular valores.

Un **informe sumario** sólo lista los totales, sin registros de los detalles individuales.

La **programación defensiva** es una técnica con la que usted trata de prepararse para todos los errores posibles antes de que ocurran.

Validar datos es asegurarse de que los elementos de datos son significativos y útiles; por ejemplo, de que los valores son el tipo de datos correcto o que se ubican dentro de un rango aceptable.

GIGO (entra basura, sale basura) significa que si su entrada es incorrecta, su salida carece de valor.

Forzar un elemento de datos significa que se anulan los que son incorrectos al establecerlos en un valor específico.

Preguntas de repaso

- La estructura que le permite escribir un conjunto de instrucciones que opere en múltiples conjuntos de datos separados es _____.
 - secuencia
 - selección
 - ciclo
 - caso
 - El ciclo que aparece con frecuencia en la lógica de línea principal en un programa _____.
 - siempre depende de si una variable es igual a 0
 - funciona correctamente con base en la misma lógica que otros ciclos
 - es un ciclo no estructurado
 - es un ejemplo de un ciclo infinito
 - ¿Cuál de los siguientes *no* es un paso que debe ocurrir con cada ciclo que funcione en forma correcta?
 - Inicializar una variable de control del ciclo antes de que inicie el ciclo.
 - Establecer el valor de control de ciclo igual a un centinela durante cada iteración.
 - Comparar el valor de control de ciclo con un centinela durante cada iteración.
 - Alterar la variable de control de ciclo durante cada iteración.
 - Las declaraciones ejecutadas dentro de un ciclo se conocen en forma colectiva como _____.
 - cuerpo del ciclo
 - controles del ciclo
 - secuencias
 - centinelas

5. Un contador da seguimiento a _____.
 - a) el número de veces que ha ocurrido un evento
 - b) el número de ciclos de máquina requeridos por un segmento de programa
 - c) el número de estructuras de ciclo dentro de un programa
 - d) el número de veces que se ha revisado un software
6. Agregar 1 a una variable también se llama _____.
 - a) compendiar
 - b) reiniciar
 - c) decrementar
 - d) incrementar
7. ¿Cuál de los siguientes es un ciclo definido?
 - a) uno que se ejecuta en tanto un usuario continúa introduciendo datos válidos
 - b) uno que se ejecuta 1,000 veces
 - c) los dos anteriores
 - d) ninguno de los anteriores
8. ¿Cuál de los siguientes es un ciclo indefinido?
 - a) uno que se ejecuta exactamente 10 veces
 - b) uno que sigue a un indicador que pregunta a un usuario cuántas repeticiones hacer y usa el valor para controlar el ciclo
 - c) los dos anteriores
 - d) ninguno de los anteriores
9. Cuando decrementa una variable, usted _____.
 - a) la establece en 0
 - b) la reduce en un décimo
 - c) le resta un valor
 - d) la elimina del programa
10. Cuando dos ciclos están anidados, el que es contenido por el otro es el ciclo _____.
 - a) cautivo
 - b) no estructurado
 - c) interno
 - d) externo
11. Cuando los ciclos están anidados, _____.
 - a) por lo común comparten una variable de control de ciclo
 - b) uno debe terminar antes de que comience el otro
 - c) ambos deben ser del mismo tipo, definido o indefinido
 - d) ninguno de los anteriores

20. Una variable podría contener un valor incorrecto aun cuando _____.
 a) es el tipo de datos correcto
 b) está dentro de un rango requerido
 c) es una constante codificada por el programador
 d) todos los anteriores

Ejercicios

1. ¿Cuál es la salida para cada uno de los segmentos de seudocódigo de la figura 5-22?

a.

```

a = 1
b = 2
c = 5
while a < c
    a = a + 1
    b = b + c
endwhile
output a, b, c
  
```

b.

```

d = 4
e = 6
f = 7
while d > f
    d = d + 1
    e = e - 1
endwhile
output d, e, f
  
```

c.

```

g = 4
h = 6
while g < h
    g = g + 1
endwhile
output g, h
  
```

d.

```

j = 2
k = 5
n = 9
while j < k
    m = 6
    while m < n
        output "Adiós"
        m = m + 1
    endwhile
    j = j + 1
endwhile
  
```

e.

```

j = 2
k = 5
m = 6
n = 9
while j < k
    while m < n
        output "Hola"
        m = m + 1
    endwhile
    j = j + 1
endwhile
  
```

f.

```

p = 2
q = 4
while p < q
    output "Adiós"
    r = 1
    while r < q
        output "Hola"
        r = r + 1
    endwhile
    p = p + 1
endwhile
  
```

Figura 5-22 Segmentos de seudocódigo para el ejercicio 1

- Diseñe la lógica para un programa que dé como salida todos los números del 1 al 20.
- Diseñe la lógica para un programa que dé como salida todos los números del 1 al 20 junto con su valor al doble y al triple.
- Diseñe la lógica para un programa que dé como salida todos los números pares del 2 al 100.
- Diseñe la lógica para un programa que dé como salida números en orden invertido del 25 al 0.
- Diseñe la lógica para un programa que permita a un usuario introducir un número. Despliegue la suma de todos los números desde 1 hasta el número introducido.

7. a) Diseñe una aplicación para Homestead Furniture Store que obtenga los datos de las transacciones de ventas, incluidos un número de cuenta, nombre del cliente y precio de compra. Dé salida al número de cuenta y nombre y luego al pago del cliente cada mes durante los siguientes 12 meses. Suponga que no hay cargos por financiamiento, que el cliente no hace compras nuevas y que liquida el saldo con pagos mensuales iguales.
b) Modifique la aplicación de Homestead Furniture Store de modo que se ejecute de manera continua para cualquier cantidad de clientes hasta que se suministre un valor centinela para el número de cuenta.
8. a) Diseñe una aplicación para Domicile Designs que obtenga los datos de las transacciones de ventas, incluidos un número de cuenta, nombre del cliente y precio de compra. La tienda carga 1.25% de interés sobre el saldo deudor cada mes. Dé salida al número de cuenta y nombre, luego al saldo proyectado del cliente cada mes durante los siguientes 12 meses. Suponga que cuando el saldo llegue a 25 dólares o menos, el cliente puede liquidar la cuenta. Al principio de cada mes, se agrega 1.25% de interés al saldo, y luego el cliente hace un pago igual a 7% del saldo actual. Suponga que el cliente no hace compras nuevas.
b) Modifique la aplicación de Domicile Designs de modo que se ejecute en forma continua para cualquier cantidad de clientes hasta que se suministre un valor centinela para el número de cuenta.
9. Yabe Online Auctions requiere que sus vendedores publiquen los artículos para venta por un periodo de seis semanas durante el que el precio de cualquier artículo no vendido baja 12% cada semana. Por ejemplo, uno que cuesta 10 dólares durante la primera semana cuesta 12% menos, u 8.80 dólares, durante la segunda semana. Durante la tercera, el mismo artículo cuesta 12% menos que 8.80, o 7.74 dólares. Diseñe una aplicación que permita a un usuario introducir precios hasta que se introduzca un valor centinela apropiado. La salida del programa es el precio de cada artículo durante cada semana, de la uno a la seis.
10. El señor Roper es propietario de 20 edificios de departamentos. Cada edificio contiene 15 unidades que renta por 800 dólares por mes cada uno. Diseñe la aplicación que daría salida a 12 talonarios de pago para cada uno de los 15 departamentos en cada uno de los 20 edificios. Cada talonario debe contener el número de edificio (1 a 20), el número de departamento (1 a 15), el mes (1 a 12) y la cantidad de renta que se debe.
11. Diseñe una calculadora de planeación del retiro para Skulling Financial Services. Permita que un usuario introduzca el número de años de trabajo que restan en su profesión y la cantidad anual de dinero que puede ahorrar. Suponga que el usuario gana 3% de interés simple sobre sus ahorros cada año. La salida del programa es un calendario que lista cada número de año en retiro empezando con el año cero y los ahorros del usuario al inicio de ese año. Suponga que el usuario gasta 50,000 dólares por año en el retiro y luego gana 3% de interés sobre el saldo restante. Termine la lista después de 40 años, o cuando el saldo del usuario sea 0 o menos, lo que ocurra primero.
12. Ellison Private Elementary School tiene tres salones de clases en cada uno de nueve grados, jardín de niños (grado 0) hasta el grado 8, y permite a los padres pagar la colegiatura a lo largo del año escolar de nueve meses. Diseñe la aplicación que dé salida al pago de nueve talonarios de colegiatura para cada uno de los 27 salones de clases. Cada

talonario debe contener el número de grado (0 a 8), el número de salón de clases (1 a 3), el mes (1 a 9) y la cantidad de colegiatura que se debe. La colegiatura para el jardín de niños es de 80 dólares por mes. La colegiatura para los otros grados es de 60 dólares por mes por el nivel de grado.

13. a) Diseñe un programa para la Hollywood Movie Rating Guide, que pueda instalarse en un quiosco en los cines; cada cliente del cine introduce un valor de 0 a 4 indicando el número de estrellas con las que califica a la película de la semana que se presenta en la guía. Si un usuario introduce un valor que no queda en el rango correcto, vuelva a indicarle varias veces hasta que introduzca un valor correcto. El programa se ejecuta en forma continua hasta que el gerente del cine introduce un número negativo para salir. Al final del programa, despliegue la clasificación de estrellas promedio para la película.
- b) Modifique el programa de la clasificación de películas de modo que un usuario tenga tres intentos para introducir una clasificación válida. Después de tres entradas incorrectas, el programa emite un mensaje apropiado y continúa con un usuario nuevo.
14. La cafetería Noir Coffee Shop desea contar con alguna investigación de mercado sobre sus clientes. Cuando un cliente hace un pedido, un empleado le pide su código postal y su edad; el empleado introduce los datos al igual que el número de artículos que ordenó el cliente. El programa opera en forma continua hasta que el empleado introduce un 0 para el código postal al final del día. Cuando el empleado introduce un código postal inválido (más de cinco dígitos) o una edad inválida (definida como menos de 10 o más de 110) el programa le pide de nuevo una entrada en forma continua. Cuando el empleado introduce menos de 1 o más de 12 artículos, el programa vuelve a pedirle una entrada dos veces más. Si introduce un valor alto en el tercer intento el programa lo acepta, pero si se trata de un valor menor que 1 en el tercer intento, se despliega un mensaje de error y el pedido no se cuenta. Al final del programa, despliega un conteo del número de artículos ordenados por los clientes del mismo código postal que la cafetería (54984) y un conteo de otros códigos postales. También despliega la edad promedio del cliente al igual que cuenta el número de artículos ordenados por clientes menores de 30 y por clientes de 30 y más.



Encuentre los errores

15. Sus archivos descargables para el capítulo 5 incluyen DEBUG05-01.txt, DEBUG05-02.txt y DEBUG05-03.txt. Cada archivo comienza con algunos comentarios que describen el problema. Los comentarios son líneas que comienzan con dos diagonales (//). Después de los comentarios, cada archivo contiene seudocódigo que tiene uno o más errores que usted debe encontrar y corregir. (NOTA: Estos archivos se encuentran disponibles sólo para la versión original en inglés.)



Zona de juegos

16. En el capítulo 2, aprendió que en muchos lenguajes de programación usted puede generar un número aleatorio entre 1 y un valor límite llamado LIMIT usando una

declaración similar a `randomNumber = random(LIMIT)`. En el capítulo 4, creó la lógica para un juego de adivinanza en el que la aplicación genera un número aleatorio y el jugador intenta adivinarlo. Ahora, cree el juego de adivinanza en sí. Después de cada adivinanza, despliegue un mensaje indicando si la del jugador fue correcta, demasiado alta o demasiado baja. Cuando el jugador al fin adivina el número correcto, despliegue un conteo del número de intentos que se requirieron.

17. Cree la lógica para un juego que simule el lanzamiento de dos dados generando dos números entre 1 y 6 inclusive. El jugador elige un número entre 2 y 12 (los totales más bajo y más alto posibles para dos dados). Entonces el jugador “lanza” dos dados hasta tres veces. Si sale el número elegido por él, entonces gana y el juego termina. Si el número no sale en los tres lanzamientos, la computadora gana.
18. Cree la lógica para el juego de dados Pig, en el que un jugador puede competir con la computadora. El objeto del juego es ser el primero en obtener 100 puntos. El usuario y la computadora toman turnos para “lanzar” un par de dados siguiendo estas reglas:
 - En un turno, cada jugador lanza dos dados. Si no aparece 1, los valores de los dados se suman a un total acumulado para el turno y el jugador puede elegir si lanza de nuevo o pasa el turno al otro. Cuando un jugador pasa el total acumulado en el turno se suma a su total del juego.
 - Si aparece 1 en uno de los dados, el total del turno del jugador se convierte en 0; en otras palabras, no se suma nada más al total del juego del jugador para ese turno, y le toca el turno al otro.
 - Si aparece 1 en ambos dados, no sólo se acaba el turno del jugador, sino que el total acumulado entero del jugador se reinicia a 0.
 - Cuando la computadora no lanza un 1 y puede elegir si lanza de nuevo, genera un número aleatorio de 1 a 2. Entonces la computadora decidirá continuar cuando el valor es 1 y decidirá salir y pasar el turno al jugador cuando el valor no es 1.



Para discusión

19. Suponga que escribe un programa que usted sospecha que está en un ciclo infinito debido a que se mantiene corriendo por varios minutos sin salida y sin terminar. ¿Qué le agregaría a su programa para que le ayude a descubrir el origen del problema?
20. Suponga que sabe que todos los empleados en su organización tienen un número de identificación de siete dígitos que se usa para entrar al sistema de cómputo. Un ciclo sería útil para adivinar todas las combinaciones de siete dígitos en una identificación. ¿Hay alguna circunstancia en la que usted debería intentar adivinar el número de identificación de otro empleado?
21. Si todos los empleados en una organización tuvieran un número de identificación de siete dígitos, adivinar todas las combinaciones posibles sería una tarea de programación relativamente fácil. ¿Cómo podría alterar el formato de las identificaciones de empleados para hacerlas más difíciles de adivinar?

6

CAPÍTULO

Arreglos

En este capítulo usted aprenderá sobre:

- ◎ Almacenamiento de datos en arreglos
- ◎ Cómo un arreglo puede reemplazar las decisiones anidadas
- ◎ Usar constantes con arreglos
- ◎ Buscar un arreglo para una correspondencia exacta
- ◎ Usar arreglos paralelos
- ◎ Buscar un arreglo para una correspondencia de rango
- ◎ Permanecer dentro de los límites del arreglo
- ◎ Usar un ciclo **for** para procesar arreglos

Almacenamiento de datos en arreglos

Un **arreglo** es una serie o lista de valores en la memoria de la computadora. Por lo general, todos los valores en un arreglo tienen algo en común; por ejemplo, podrían representar una lista de los números de identificación de los empleados o de los precios para los artículos vendidos en una tienda.

Siempre que usted requiera múltiples ubicaciones de almacenamiento para los objetos, puede usar el equivalente de un arreglo de programación de la vida real. Si guarda papeles importantes en carpetas de archivos y etiqueta cada carpeta con una letra consecutiva del alfabeto, entonces usa el equivalente de un arreglo; si conserva sus recibos en una pila de cajas de zapatos y etiqueta cada caja con el nombre de un mes, también lo hace. Del mismo modo, cuando planea los cursos para el siguiente semestre en su escuela revisando una lista de ofertas de cursos, usa un arreglo.



Los arreglos que se han expuesto en este capítulo son unidimensionales y son similares a las listas.

Cada uno de estos arreglos de la vida real le ayuda a organizar los objetos o la información. Usted *podría* almacenar todos sus documentos o recibos en una enorme caja de cartón o encontrar los cursos si están impresos al azar en un libro grande; sin embargo, usar un sistema organizado de almacenamiento y despliegue hace su vida más fácil en cada caso. El uso de un arreglo de programación logrará los mismos resultados para sus datos.

De qué modo los arreglos ocupan la memoria de la computadora

Cuando usted declara un arreglo se refiere a una estructura que contiene múltiples elementos de datos; cada uno de éstos es un **elemento** del arreglo. Cada elemento tiene el mismo tipo de datos y ocupa un área en la memoria junto a los otros, o contiguo a ellos. Usted puede indicar el número de elementos que contendrá un arreglo (el **tamaño del arreglo**) cuando lo declara junto con sus otras variables y constantes. Por ejemplo, podría declarar un arreglo numérico de tres elementos no inicializado cuyo nombre es `someVals` como sigue:

```
num someVals[3]
```

Cada elemento del arreglo se distingue de los otros con un **subíndice** único, conocido como **índice**, que es un número que indica la posición de un elemento particular dentro de un arreglo. Todos los elementos del arreglo tienen el mismo nombre de grupo, pero cada uno también tiene un subíndice único que indica qué tan lejos está del primer elemento. Por consiguiente, todos los subíndices del arreglo son siempre una secuencia de enteros.

Por ejemplo, un arreglo de cinco elementos usa los subíndices 0 a 4 y uno de 10 elementos usa los subíndices 0 a 9. En todos los lenguajes, los valores del subíndice deben ser números enteros secuenciales. En la mayor parte de los lenguajes modernos, como Visual Basic, Java, C++ y C#, se tiene acceso al primer elemento del arreglo usando el subíndice 0 y en este libro sigue esta convención.

Para usar un elemento de arreglo se coloca su subíndice dentro de paréntesis o corchetes (dependiendo del lenguaje de programación) después del nombre del grupo. En este libro se

usan corchetes para contener los subíndices del arreglo de modo que usted no confunda los nombres de arreglo con los nombres de método. Muchos lenguajes de programación más recientes como C++, Java y C# también usan la notación con corchetes.

Después de que usted declara un arreglo puede asignar valores a algunos de los elementos o a todos en forma individual. La asignación de valores al arreglo en ocasiones se conoce como **poblar el arreglo**. El siguiente código muestra una declaración de arreglo de tres elementos seguida por tres declaraciones separadas que pueblan el arreglo:

Declarations

```
num someVals[3]
someVals[0] = 25
someVals[1] = 36
someVals[2] = 47
```

La figura 6-1 muestra un arreglo nombrado `someVals` que contiene tres elementos, de modo que éstos son `someVals[0]`, `someVals[1]` y `someVals[2]`; se les han asignado los valores 25, 36 y 47, respectivamente. El elemento `someVals[0]` está cero números alejado del comienzo del arreglo; `someVals[1]` está a un número del inicio del arreglo y `someVals[2]` está a dos números de distancia.

Si es apropiado, usted puede declarar e inicializar los elementos de arreglo en una declaración; la mayoría de los lenguajes de programación usan una similar a la siguiente para declarar un arreglo de tres elementos y asignarle valores:

```
num someVals[3] = 25, 36, 47
```

Cuando use una lista de valores para inicializar un arreglo, el primer valor que liste se asignará al primer elemento del mismo (elemento 0) y los valores subsiguientes se asignan en orden a los elementos restantes. Muchos lenguajes de programación le permiten inicializar un arreglo con menos valores iniciales que los elementos declarados, pero ninguno le permite inicializarlo usando más valores iniciales que posiciones disponibles. Cuando se asignen los valores iniciales para un arreglo en este libro, a cada elemento se le proporcionará un valor.

Después de que se ha declarado un arreglo y se han asignado valores apropiados a los elementos específicos, usted puede usar un elemento individual en la misma forma en que usaría cualquier otro elemento de datos del mismo tipo. Por ejemplo, puede introducir valores para los elementos del arreglo y dar salida a los valores, y si los elementos son numéricos puede efectuar aritmética con ellos.

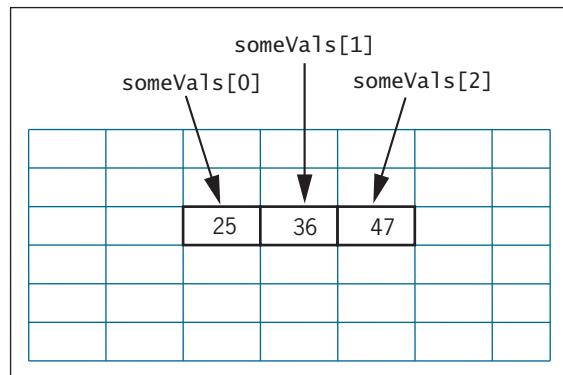


Figura 6-1 Apariencia de un arreglo de tres elementos en la memoria de la computadora

DOS VERDADES Y UNA MENTIRA

Almacenamiento de datos en arreglos

1. En un arreglo, cada elemento tiene el mismo tipo de datos.
2. Se tiene acceso a cada elemento del arreglo usando un subíndice, que puede ser un número o una cadena.
3. Los elementos del arreglo siempre ocupan ubicaciones de memoria adyacentes.

La afirmación falsa es la número 2. El subíndice de un arreglo debe ser un número. Puede ser una constante nombrada, una constante literal o una variable.

Cómo un arreglo puede reemplazar decisiones anidadas

Considere una aplicación solicitada por el departamento de recursos humanos de una compañía para elaborar estadísticas sobre los dependientes declarados por los empleados. El departamento desea un informe que liste el número de empleados que han declarado 0, 1, 2, 3, 4 o 5 dependientes (suponga que sabe que ningún empleado tiene más de cinco). Por ejemplo, la figura 6-2 muestra un informe típico.

Sin usar un arreglo, usted podría escribir la aplicación que produce los conteos para las seis categorías de dependientes (0 a 5) usando una serie de decisiones. La figura 6-3 muestra el pseudocódigo y el diagrama de flujo para la parte de la toma de decisiones de una aplicación así. Aunque esta lógica funciona, su extensión y complejidad son innecesarias una vez que entiende cómo usar un arreglo.

Dependientes	Cuenta
0	43
1	35
2	24
3	11
4	5
5	7

Figura 6-2 Informe de dependientes típico

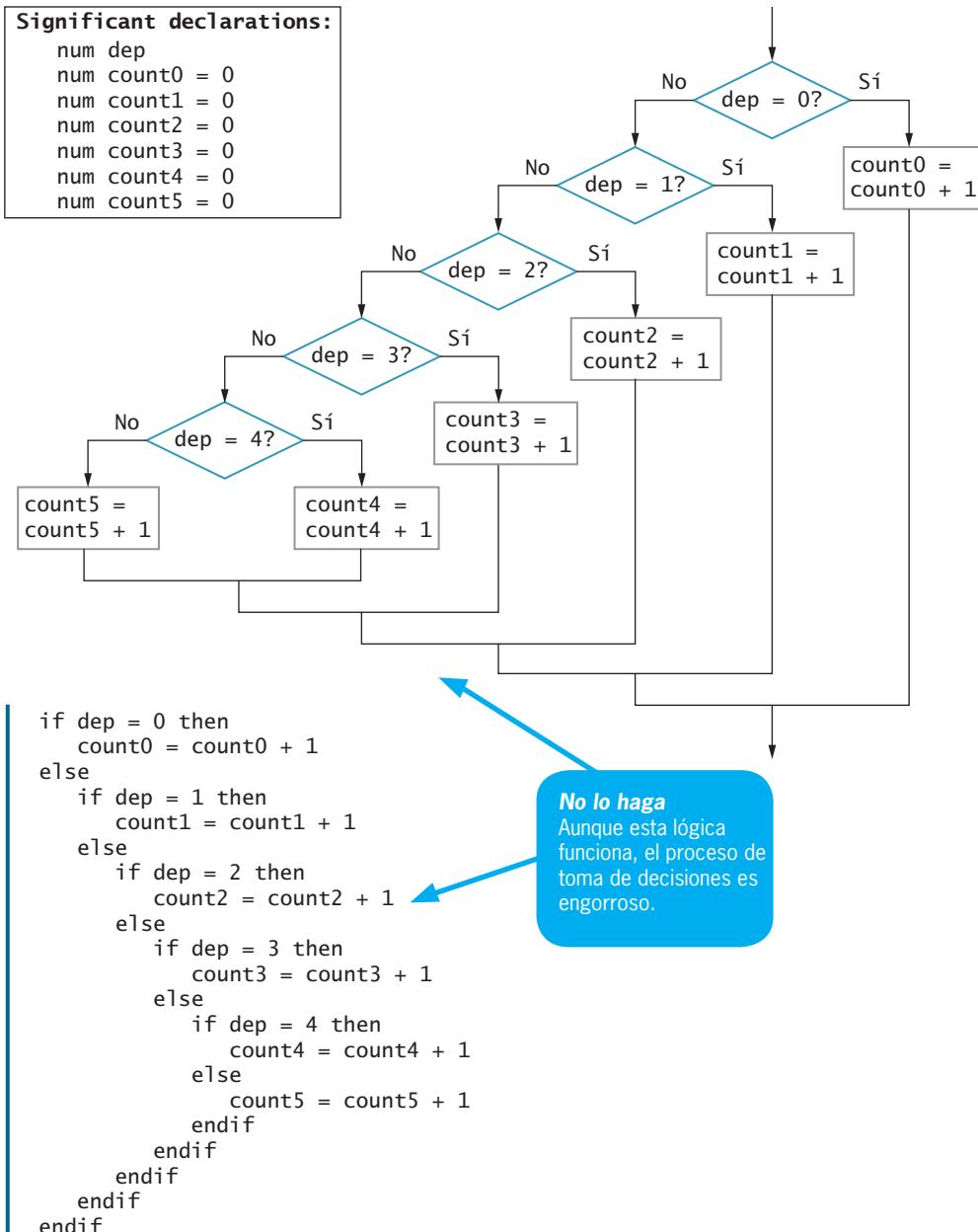


Figura 6-3 Diagrama de flujo y seudocódigo del proceso de toma de decisiones usando una serie de decisiones: modo difícil



El proceso de toma de decisiones en la figura 6-3 logra su propósito y la lógica es correcta, pero el proceso es engorroso y ciertamente no se recomienda. Siga la lógica aquí de modo que entienda cómo funciona la aplicación; en las páginas siguientes, verá cómo hacerla más elegante.

218

En la figura 6-3, la variable `dep` se compara con 0. Si es 0, se agrega 1 a `count0`; si no, entonces `dep` se compara con 1. Con base en el resultado, se agrega 1 a `count1` o `dep` se compara con 2, y así sucesivamente. Cada vez que la aplicación ejecuta este proceso de toma de decisiones al final se agrega 1 a una de las seis variables que actúan como contador. La lógica dependiente del conteo en la figura 6-3 funciona, pero aun con sólo seis categorías de dependientes, el proceso de toma de decisiones es poco manejable. ¿Qué tal si el número de dependientes fuera cualquier valor de 0 a 10, o de 0 a 20? Con estos escenarios la lógica básica del programa permanecería igual; sin embargo, usted necesitaría declarar muchas variables adicionales para llevar las cuentas y requeriría muchas decisiones adicionales.

El uso de un arreglo proporciona un enfoque alternativo a este problema de programación y reduce en gran medida el número de declaraciones que usted necesita. Cuando declara un arreglo, proporciona un nombre de grupo para un número de variables asociadas en la memoria. Por ejemplo, los seis acumuladores dependientes de la cuenta pueden redefinirse como un solo arreglo llamado `count`. Los elementos individuales se vuelven `count[0]`, `count[1]`, `count[2]`, `count[3]`, `count[4]` y `count[5]`, como se muestra en el proceso de toma de decisiones que se revisa en la figura 6-4.

La declaración sombreada en la figura 6-4 muestra que cuando `dep` es 0, se agrega 1 a `count[0]`. Puede ver declaraciones similares para el resto de los elementos `count`; cuando `dep` es 1, se agrega 1 a `count[1]`, cuando `dep` es 2, se agrega 1 a `count[2]`, y así sucesivamente. Cuando el valor de `dep` es 5, esto significa que no fue 1, 2, 3, o 4, así que se agrega 1 a `count[5]`. En otras palabras, se agrega 1 a uno de los elementos del arreglo `count` en lugar de a una variable individual llamada `count0`, `count1`, `count2`, `count3`, `count4` o `count5`. ¿Esta versión es una gran mejora sobre el original de la figura 6-3? Por supuesto que no. Todavía no ha obtenido ventaja de los beneficios de usar el arreglo en esta aplicación.

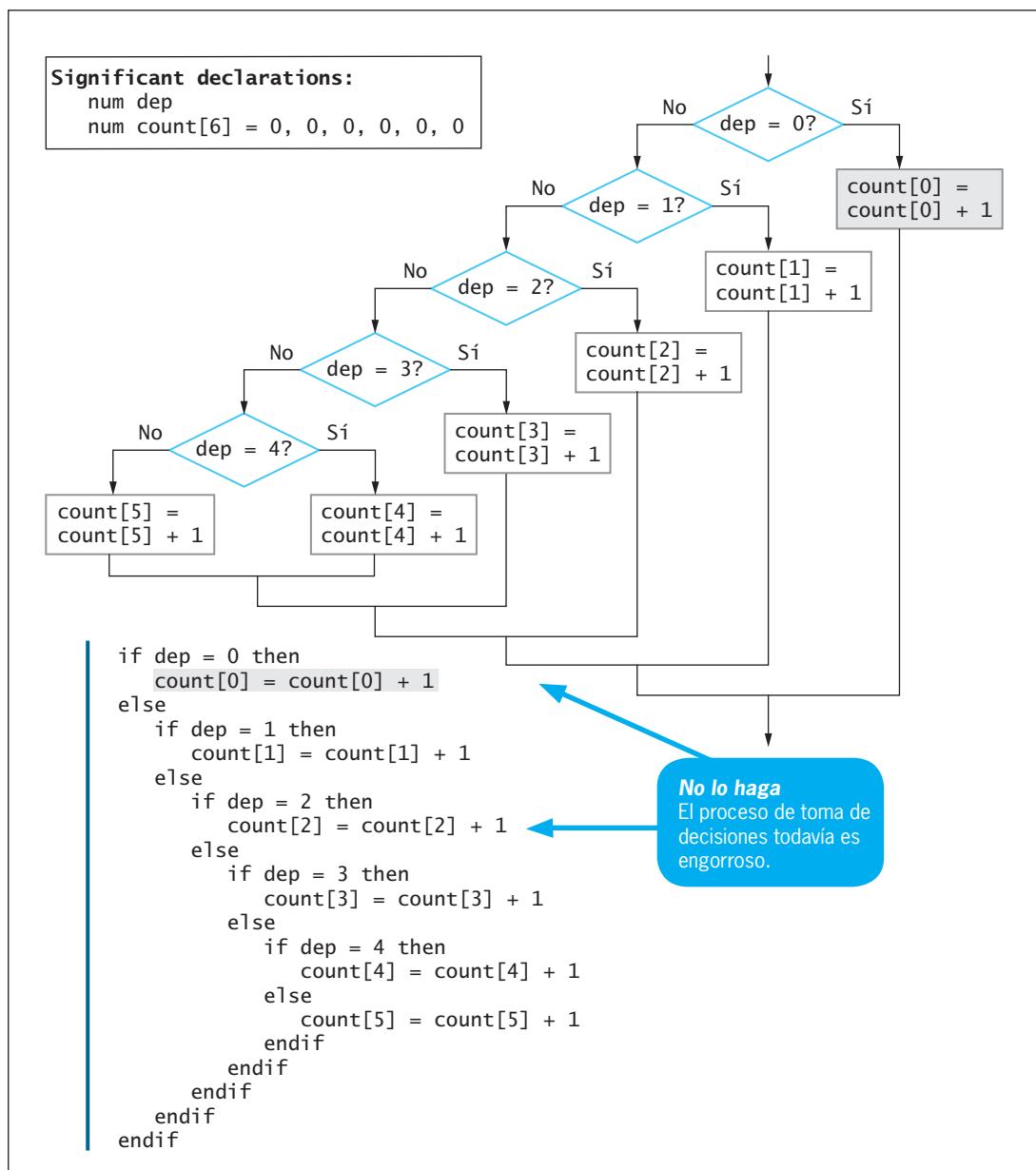


Figura 6-4 Diagrama de flujo y seudocódigo del proceso de toma de decisiones: pero todavía del modo difícil

El verdadero beneficio de utilizar un arreglo está en su capacidad para usar una variable como un subíndice para el arreglo, en lugar de una constante literal como 0 o 5. Note en la lógica de la figura 6-4 que, dentro de cada decisión, el valor comparado con `dep` y la constante que es el subíndice en el proceso *Sí* resultante siempre son idénticos. Es decir, cuando `dep` es 0, el subíndice que se usa para agregar 1 al arreglo `count` es 0; cuando `dep` es 1, el subíndice para

el arreglo count es 1, y así sucesivamente; por consiguiente, puede usar sólo dep como un subíndice para el arreglo. Puede reescribir el proceso de toma de decisiones como se muestra en la figura 6-5.

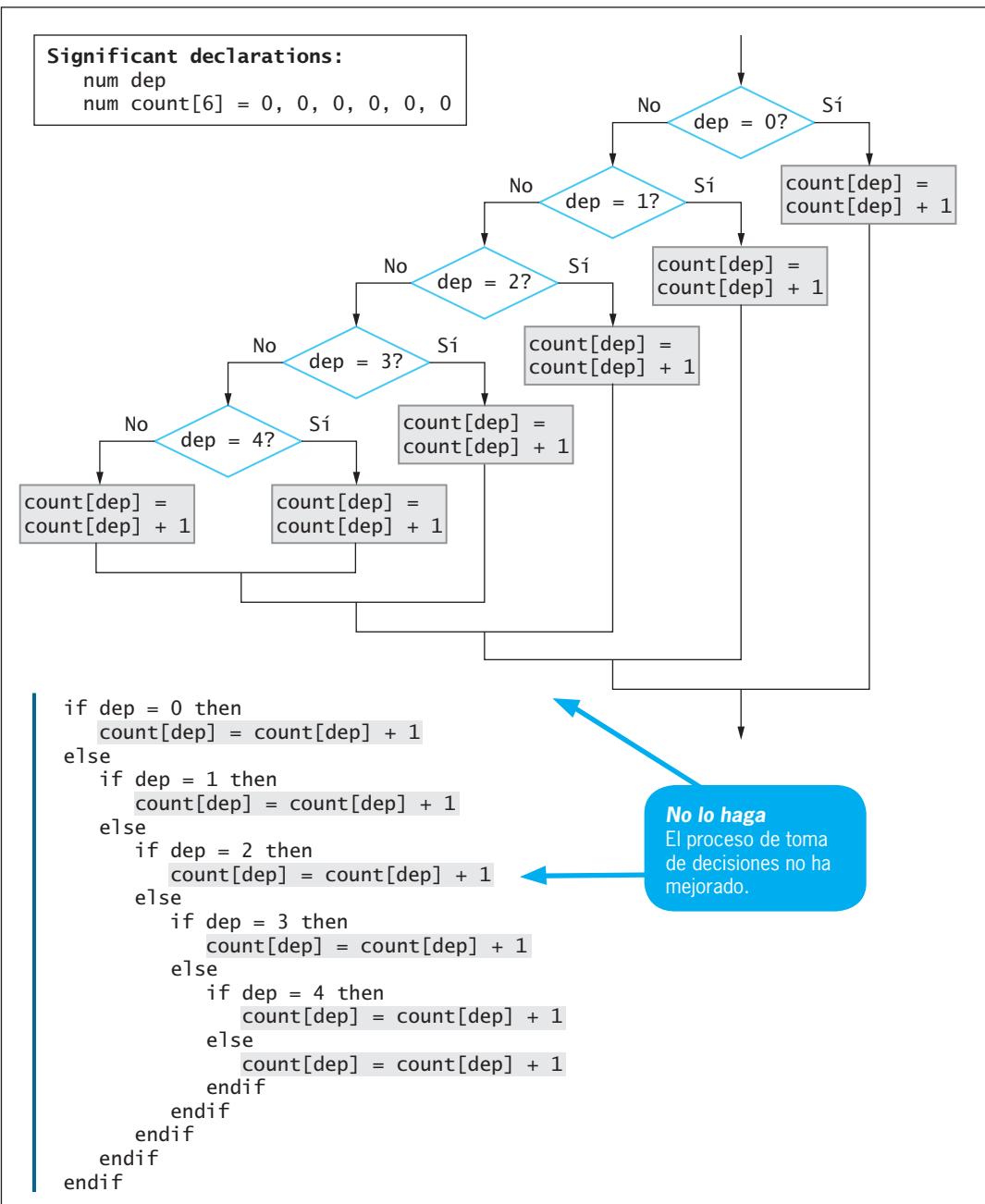


Figura 6-5 Diagrama de flujo y seudocódigo del proceso de toma de decisiones usando un arreglo: pero todavía de un modo difícil

El segmento de código en la figura 6-5 no se ve más eficiente que el de la figura 6-4. Sin embargo, note las declaraciones sombreadas en la 6-5; el proceso que ocurre después de cada decisión es exactamente el mismo. En cada caso, sin importar cuál sea el valor de `dep`, siempre agrega 1 a `count[dep]`. Si siempre emprendemos la misma acción sin importar cuál sea la respuesta a una pregunta, no hay necesidad de hacer la pregunta. En cambio, usted puede reescribir el proceso de toma de decisiones como se muestra en la figura 6-6.

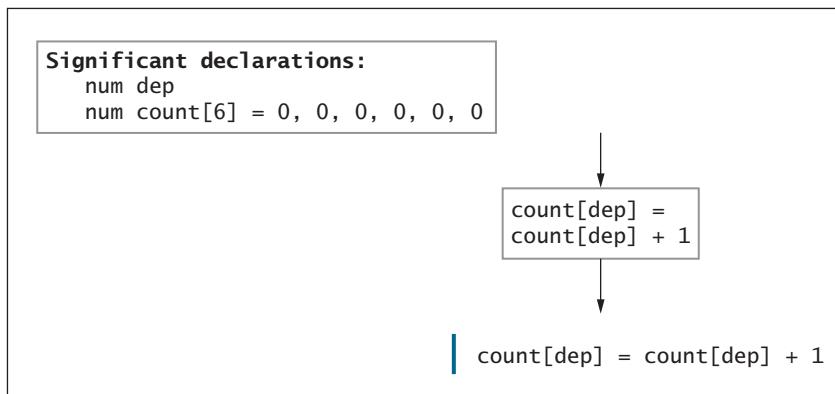


Figura 6-6 Diagrama de flujo y seudocódigo de un proceso de toma de decisiones eficiente usando un arreglo

¡La declaración única en la figura 6-6 elimina el proceso de toma de decisiones *entero* que era la sección original resaltada en la figura 6-5! Cuando `dep` es 2, se agrega 1 a `count[2]`; cuando `dep` es 4, se agrega 1 a `count[4]`, y así sucesivamente. *Ahora* usted ha mejorado de manera significativa la lógica original. Lo que es más, este proceso no cambia ya sea que haya 20, 30 o cualquier otro número de categorías posibles. Para usar más de cinco acumuladores, usted declararía elementos `count` adicionales en el arreglo, pero la lógica clasificadora permanecería igual que en la figura 6-6.

La figura 6-7 muestra un programa entero que obtiene ventaja del arreglo a fin de producir el informe que muestra los conteos para las categorías dependientes. Se declaran las variables y las constantes y, en el módulo `getReady()` se introduce un primer valor para `dep` en el programa. En el módulo `countDependents()`, se agrega 1 al elemento apropiado del arreglo `count` y se introduce el siguiente valor. El ciclo en la lógica de la línea principal en la figura 6-7 es indefinido; continúa en tanto el usuario no introduzca el valor centinela. Cuando la entrada de datos está completa, el módulo `finishUp()` despliega el informe. Primero, se da salida al encabezado, luego `dep` se reinicia a 0 y después se da salida a cada `dep` y `count[dep]` en un ciclo. La primera declaración de salida contiene 0 (como el número de dependientes) y el valor almacenado en `count[0]`. Luego, se agrega 1 a `dep` y se usa de nuevo el mismo conjunto de instrucciones con el objetivo de desplegar los conteos para cada número de dependientes. El ciclo en el módulo `finishUp()` es definido; se ejecuta precisamente seis veces.

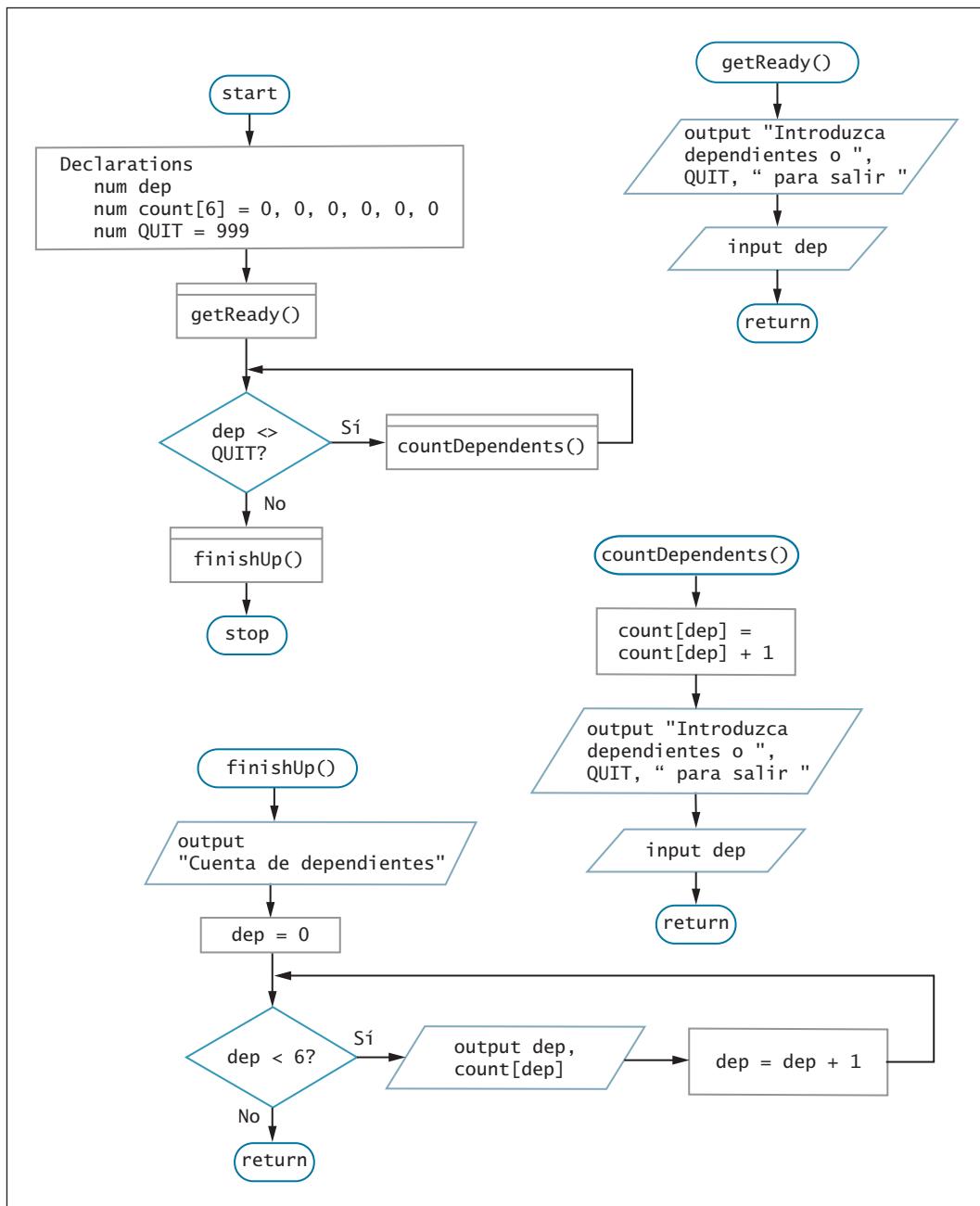


Figura 6-7 Diagrama de flujo y seudocódigo para el programa de informe Dependientes (continúa)

(continuación)

```
start
    Declarations
        num dep
        num count[6] = 0, 0, 0, 0, 0, 0
        num QUIT = 999
    getReady()
    while dep <> QUIT
        countDependents()
    endwhile
    finishUp()
stop

getReady()
    output "Introduzca dependientes o ", QUIT, " para salir "
    input dep
return

countDependents()
    count[dep] = count[dep] + 1
    output "Introduzca dependientes o ", QUIT, " para salir "
    input dep
return

finishUp()
    output "Cuenta de dependientes"
    dep = 0
    while dep < 6
        output dep, count[dep]
        dep = dep + 1
    endwhile
return
```

Figura 6-7 Diagrama de flujo y seudocódigo para el programa de informe Dependientes

El programa dependiente del conteo habría *funcionado* si contuviera una larga serie de decisiones y declaraciones de salida, pero es más fácil escribirlo cuando usted usa un arreglo y tiene acceso a sus valores usando el número de dependientes como un subíndice. Además, el programa nuevo es más eficiente y es más fácil que otros programadores lo entiendan y le den mantenimiento. Los arreglos nunca son obligatorios, pero con frecuencia pueden reducir de manera drástica su tiempo de programación y hacer que su lógica sea más fácil de entender.

Aprender a usar arreglos de manera apropiada puede hacer mucho más eficientes y profesionales diversas tareas de programación. Cuando entienda cómo usar los arreglos será capaz de proporcionar soluciones elegantes a los problemas que de otra manera requerirían pasos tediosos de programación.

DOS VERDADES Y UNA MENTIRA

Cómo un arreglo puede reemplazar decisiones anidadas

1. Usted puede usar un arreglo para reemplazar una larga serie de decisiones.
2. Usted experimenta un beneficio mayor de los arreglos cuando usa una constante literal numérica y no una variable como subíndice.
3. El proceso de desplegar cada elemento en un arreglo de 10 elementos en esencia no es diferente de desplegar cada elemento en un arreglo de 100.

La afirmación falsa es la número 2. Usted experimenta un beneficio mayor de los arreglos cuando usa una variable como subíndice en oposición a usar una constante.

Uso de constantes con arreglos

En el capítulo 2 usted aprendió que las constantes nombradas contienen valores que no cambian durante la ejecución de un programa. Cuando se trabaja con arreglos, puede usar constantes en varias formas:

- Para contener el tamaño de un arreglo
- Como valores del arreglo
- Como subíndices

Uso de una constante como el tamaño de un arreglo

El programa en la figura 6-7 todavía contiene un defecto menor. A lo largo de este libro usted ha aprendido a evitar los *números mágicos*; es decir, las constantes literales. Conforme se da salida a los totales en el ciclo al final del programa en la figura 6-7, el subíndice del arreglo se compara con la constante 6. El programa puede mejorar si usa en cambio una constante nombrada; esto hace que su código sea más fácil de modificar y entender. En la mayoría de los lenguajes de programación usted puede adoptar uno de estos enfoques:

- Declarar una constante numérica nombrada como `ARRAY_SIZE = 6`. Luego usar esta constante cada vez que tenga acceso al arreglo, asegurándose siempre de que cualquier subíndice que use permanezca menor que el valor constante.
- En muchos lenguajes se proporciona en forma automática una constante que representa el tamaño del arreglo para cada uno de éstos que cree. Por ejemplo, en Java, después de declarar un arreglo nombrado `count`, su tamaño se almacena en un campo llamado `count.length`. Tanto en C# como en Visual Basic, el tamaño del arreglo es `count.Length`, con *L* mayúscula.

Uso de constantes como valores de elemento del arreglo

En ocasiones los valores almacenados en arreglos deberían ser constantes porque no se cambian durante la ejecución del programa. Por ejemplo, suponga que crea un arreglo que contiene nombres para los meses del año. No confunda el identificador del arreglo con su contenido; la convención en este libro es usar todas las letras en mayúsculas en los identificadores de constantes, pero no necesariamente en los valores de arreglo. El arreglo puede declararse como sigue:

```
string MONTH[12] = "Enero", "Febrero", "Marzo", "Abril",
    "Mayo", "Junio", "Julio", "Agosto", "Septiembre", "Octubre",
    "Noviembre", "Diciembre"
```

Uso de una constante como subíndice de un arreglo

En ocasiones usted deseará usar una constante literal numérica como subíndice para un arreglo. Por ejemplo, para desplegar el primer valor en un arreglo nombrado salesArray podría escribir una declaración que usa una constante literal como sigue:

```
output salesArray[0]
```

También podría ser que necesite usar una constante nombrada como subíndice; por ejemplo, si salesArray contiene valores de ventas para cada uno de 20 estados cubiertos por su compañía e Indiana es el número 5, podría dar salida a este valor como sigue:

```
output salesArray[5]
```

Sin embargo, si declara una constante nombrada como num INDIANA = 5, entonces puede desplegar el mismo valor con esta declaración:

```
output salesArray[INDIANA]
```

Una ventaja de usar una constante nombrada en este caso es que la declaración se vuelve auto-documentada; cualquier persona que lea su declaración entenderá con mayor facilidad que su intención es desplegar el valor de ventas para Indiana.

DOS VERDADES Y UNA MENTIRA

Uso de constantes con arreglos

1. Si usted crea una constante nombrada igual al tamaño de un arreglo, puede usarla como subíndice para este último.
2. Si crea una constante nombrada igual al tamaño de un arreglo, puede usarla como un límite contra el cual comparar los valores de los subíndices.
3. Cuando declara un arreglo en Java, C# y Visual Basic, se proporciona de manera automática una constante que representa el tamaño del arreglo.

entonces es más grande que cualquier subíndice del arreglo válido.

La afirmación falsa es la número 1. Si la constante es igual al tamaño del arreglo,

Búsqueda de un arreglo para una correspondencia exacta

En la aplicación dependiente del conteo en este capítulo, la variable de subíndice del arreglo contenía de manera conveniente números enteros pequeños (el número de dependientes permitidos era 0 a 5) y la variable dep tenía acceso directo al arreglo. Por desgracia, la vida real no siempre transcurre en enteros pequeños; en ocasiones usted no tiene una variable que contenga de modo conveniente la posición de un arreglo; a veces tiene que buscar a lo largo de uno para encontrar el valor que necesita.

Consideré un negocio de paquetería en el que los clientes colocan pedidos que contienen nombre, dirección, número de artículo y cantidad ordenada. Suponga que los números de artículo entre los que un cliente puede elegir constan de tres dígitos, pero quizás no están numerados en forma consecutiva del 001 al 999. Digamos que ofrece seis artículos: 106, 108, 307, 405, 457 y 688, como se muestra en la declaración sombreada VALID_ITEM en la figura 6-8 (el arreglo se declara como constante debido a que los números de los artículos no cambian durante la ejecución del programa). Cuando un cliente ordena un artículo, un empleado de

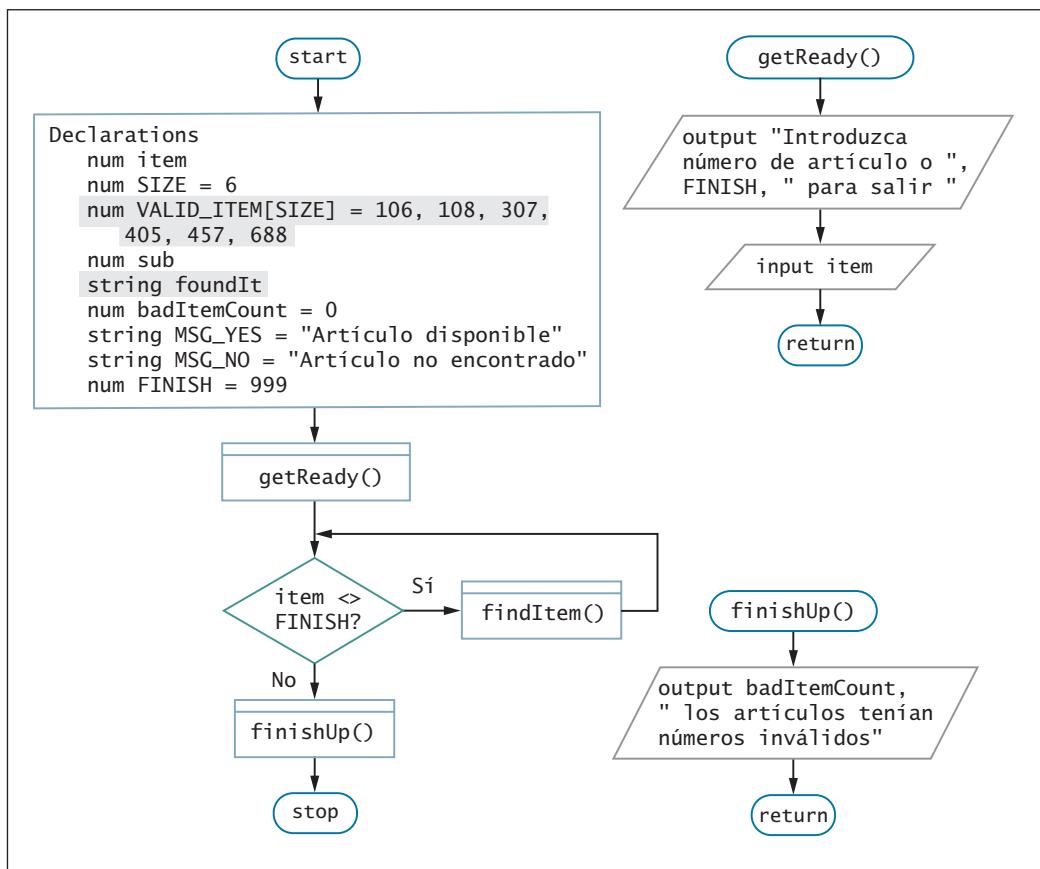


Figura 6-8 Diagrama de flujo y seudocódigo para un programa que verifica la disponibilidad de artículos (continúa)

oficina puede decir si el pedido es válido viendo la lista y verificando en forma manual que el número de artículo ordenado esté en ella. En forma similar, un programa de computadora puede usar un ciclo para probar el número de artículo ordenado contra cada VALID_ITEM, buscando una correspondencia exacta. Cuando busca en una lista de un extremo a otro lleva a cabo una **búsqueda lineal**.

(continuación)

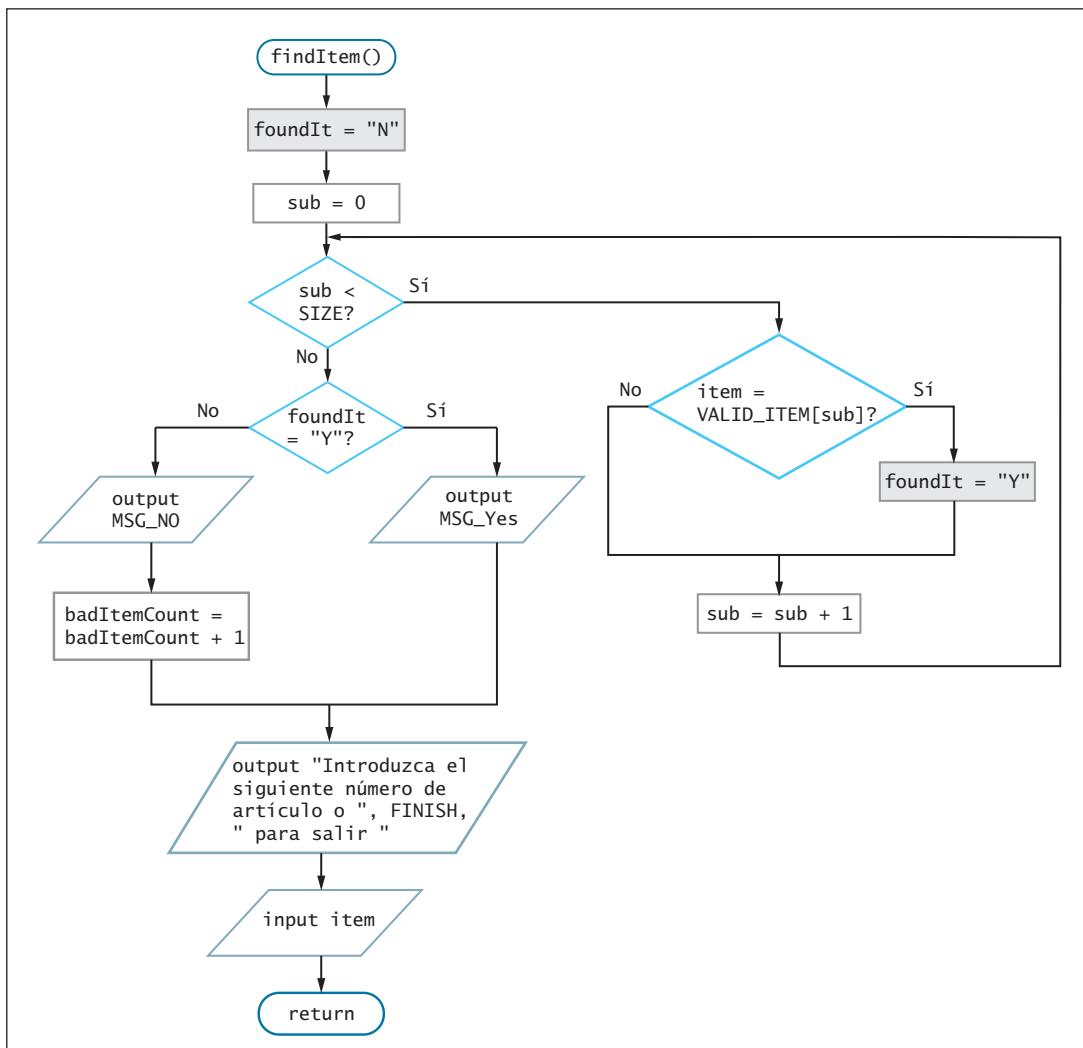


Figura 6-8 Diagrama de flujo y seudocódigo para un programa que verifica la disponibilidad de artículos (continúa)

(continuación)

```
start
    Declarations
        num item
        num SIZE = 6
        num VALID_ITEM[SIZE] = 106, 108, 307,
            405, 457, 688
        num sub
        string foundIt
        num badItemCount = 0
        string MSG_YES = "Artículo disponible"
        string MSG_NO = "Artículo no encontrado"
        num FINISH = 999
    getReady()
    while item <> FINISH
        findItem()
    endwhile
    finishUp()
stop

getReady()
    output "Introduzca número de artículo o ", FINISH,
        " para salir "
    input item
return

findItem()
    foundIt = "N"
    sub = 0
    while sub < SIZE
        if item = VALID_ITEM[sub] then
            foundIt = "Y"
        endif
        sub = sub + 1
    endwhile
    if foundIt = "Y" then
        output MSG_YES
    else
        output MSG_NO
        badItemCount = badItemCount + 1
    endif
    output "Introduzca el siguiente número de artículo o ", FINISH,
        " para salir "
    input item
return

finishUp()
    output badItemCount, " los artículos tenían números inválidos"
return
```

Figura 6-8 Diagrama de flujo y seudocódigo para un programa que verifica la disponibilidad de artículos

Para determinar si un número de artículo ordenado es válido, usted podría usar una serie de seis decisiones para comparar el número con cada uno de los seis valores permitidos. Sin embargo, el enfoque superior que se muestra en la figura 6-8 es crear un arreglo que contenga la lista de números de los artículos válidos y luego buscar a lo largo del arreglo una correspondencia exacta con el que se ha ordenado. Si busca en todo el arreglo sin encontrar una correspondencia para el artículo que el cliente ha ordenado, esto significa que el número del artículo ordenado no es válido.

El módulo `findItem()` en la figura 6-8 muestra los siguientes pasos para verificar que el número de un artículo existe:

- Una variable bandera nombrada `foundIt` se establece en "N". Una **bandera** es una variable que se establece para indicar si ha ocurrido algún evento. En este ejemplo, `N` indica que el número del artículo todavía no se ha encontrado en la lista (véase la primera declaración sombreada en el método `findItem()` en la figura 6-8).
- Un subíndice, `sub`, se establece en 0; se usará para tener acceso a cada elemento de `VALID_ITEM`.
- Se ejecuta un ciclo, variando `sub` de 0 hasta uno menos que el tamaño del arreglo. Dentro del ciclo, el número del artículo ordenado por el cliente se compara con cada número de artículo en el arreglo. Si el artículo ordenado por el cliente corresponde con cualquier artículo en el arreglo, a la variable bandera se le asigna "Y" (véase la última declaración sombreada en el método `findItem()` en la figura 6-8). Después de que los seis números de los artículos válidos se han comparado con el artículo ordenado, si el del cliente no corresponde con ninguno de ellos, entonces la variable bandera `foundIt` todavía mantendrá el valor "N".
- Si el valor de la variable bandera es "Y" después de que se ha buscado en toda la lista, esto significa que el artículo es válido y se despliega un mensaje apropiado; pero si no se ha asignado "Y" a la bandera, el artículo no fue encontrado en el arreglo de los que son válidos. En este caso se da salida a un mensaje de error y se agrega 1 a la cuenta de números de artículos deficientes.



Como una alternativa para el uso de la cadena de la variable `foundIt` en el método de la figura 6-8, usted quizás prefiera usar una variable numérica que se establece en 1 o 0. La mayoría de los lenguajes de programación también soportan un tipo de datos booleano que usted puede usar para `foundIt`; cuando declara que una variable será booleana puede establecer su valor como verdadero o falso.

DOS VERDADES Y UNA MENTIRA

Búsqueda de un arreglo para una correspondencia exacta

1. En los arreglos sólo pueden almacenarse números enteros.
2. Sólo pueden usarse números enteros como subíndices de los arreglos.
3. Una bandera es una variable que indica si ha ocurrido algún evento.

La afirmación falsa es la número 1. Pueden almacenarse números enteros en los arreglos, pero también muchos otros objetos, incluyendo cadenas y números con lugares decimales.

Uso de arreglos paralelos

Cuando usted acepta el número de un artículo en el programa de una compañía de paquetería, por lo general desea efectuar más que sólo verificar la existencia del artículo. Por ejemplo, tal vez desee determinar su nombre, precio o cantidad disponible. Tareas como éstas pueden completarse en forma eficiente usando **arreglos paralelos**; éstos son dos o más arreglos en los que cada elemento en uno de ellos está asociado con el elemento que se encuentra en la misma posición relativa en el otro arreglo. Aunque cualquier arreglo sólo puede contener un tipo de datos, cada uno en un conjunto de arreglos paralelos podría ser de un tipo diferente.

Suponga que tiene una lista de los números de los artículos y sus precios. Un arreglo que se ha nombrado VALID_ITEM contiene seis elementos; cada elemento es el número de un artículo válido. Su arreglo paralelo también tiene seis elementos; se nombra VALID_PRICE y cada elemento es el precio de un artículo. Cada precio en el arreglo VALID_PRICE está de manera conveniente y deliberada en la misma posición que el número de artículo correspondiente en el arreglo VALID_ITEM. La figura 6-9 muestra cómo se verían los arreglos paralelos en la memoria de la computadora.

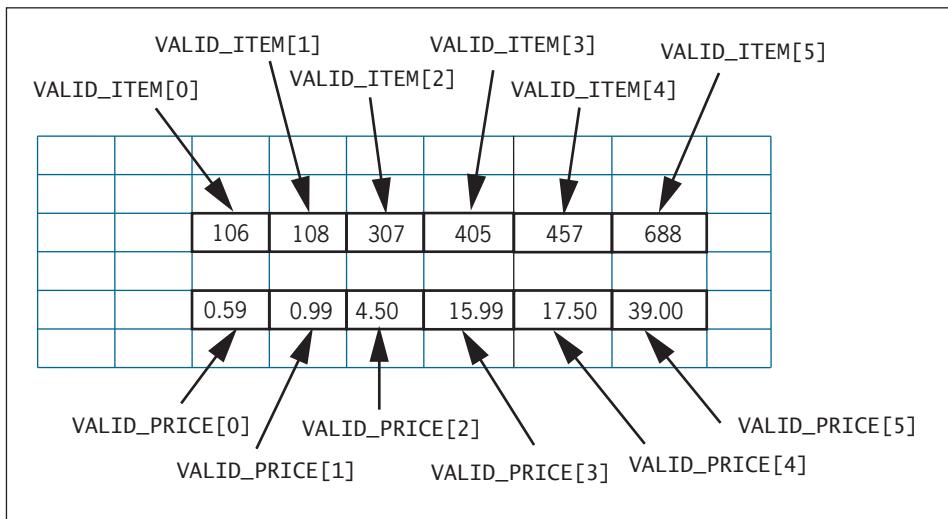


Figura 6-9 Arreglos paralelos en la memoria

Cuando usted usa arreglos paralelos:

- Dos o más arreglos contienen datos relacionados.
- Un subíndice relaciona los arreglos. Es decir, los elementos en la misma posición en cada arreglo están relacionados de manera lógica.

La figura 6-10 muestra un programa que declara arreglos paralelos. El arreglo VALID_PRICE está sombreado; cada elemento en él corresponde a un número de artículo válido.

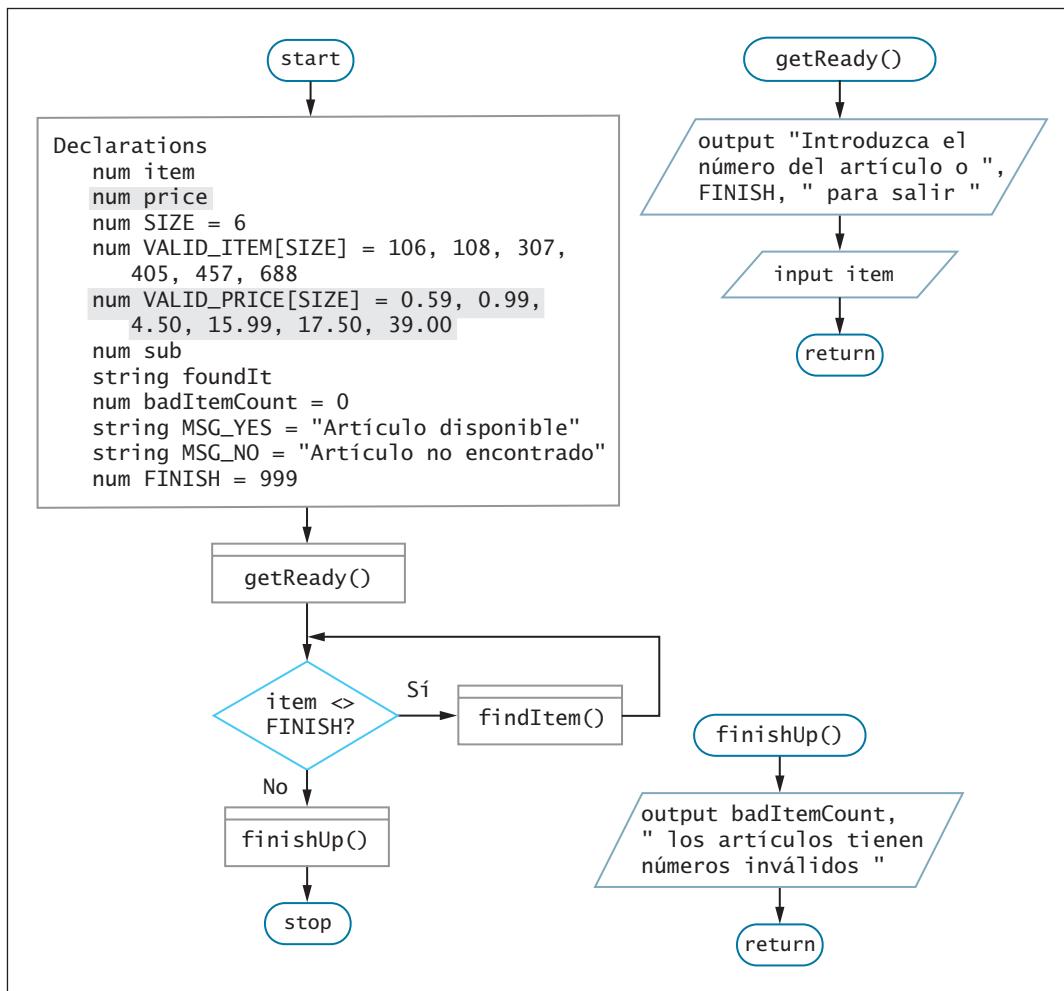


Figura 6-10 Diagrama de flujo y seudocódigo de un programa que encuentra el precio de un artículo usando arreglos paralelos (continúa)

(continuación)

232

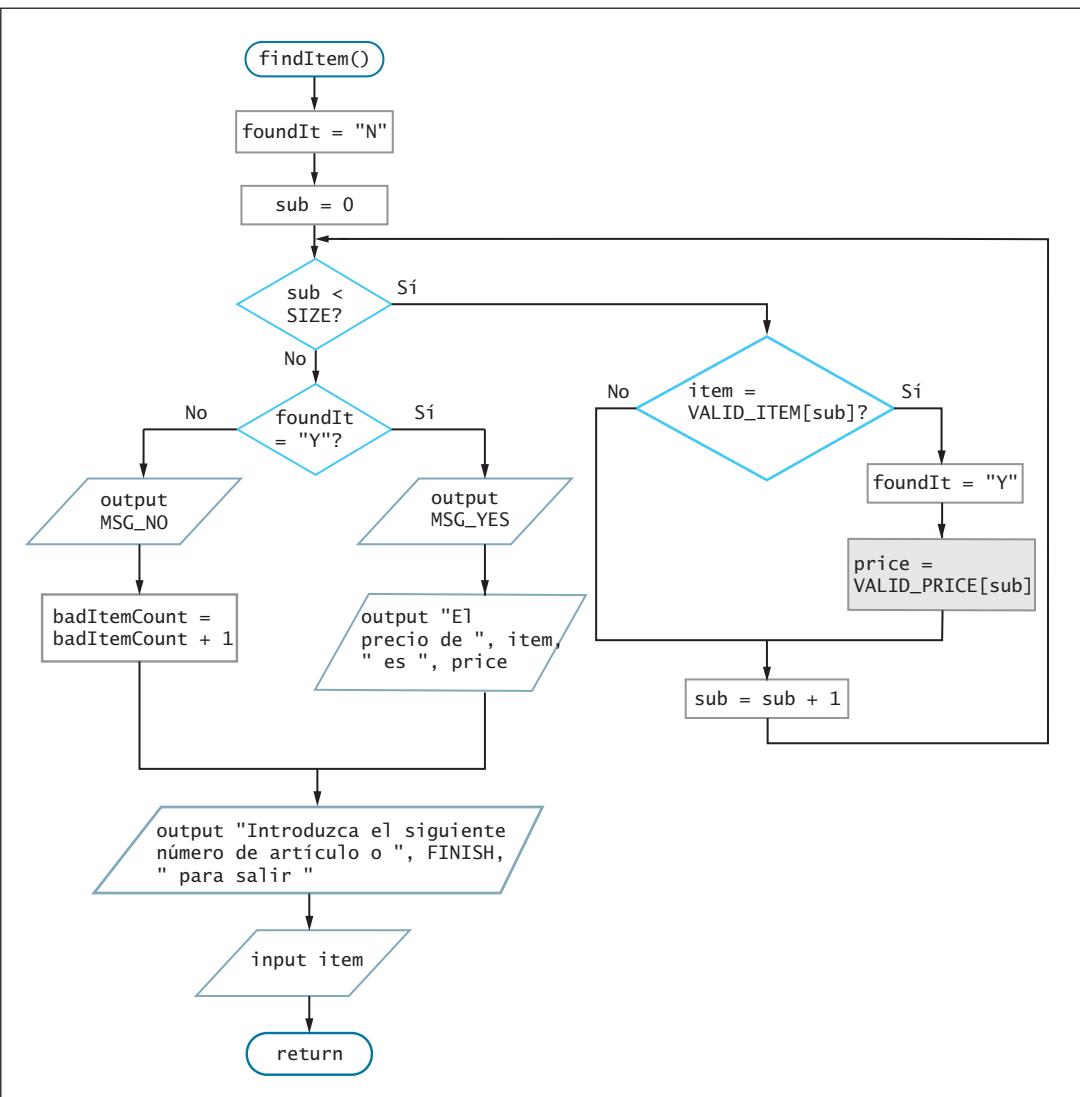


Figura 6-10 Diagrama de flujo y seudocódigo de un programa que encuentra el precio de un artículo usando arreglos paralelos (continúa)

(continuación)

```
start
    Declarations
        num item
        num price
        num SIZE = 6
        num VALID_ITEM[SIZE] = 106, 108, 307,
            405, 457, 688
        num VALID_PRICE[SIZE] = 0.59, 0.99,
            4.50, 15.99, 17.50, 39.00
        num sub
        string foundIt
        num badItemCount = 0
        string MSG_YES = "Artículo disponible"
        string MSG_NO = "Artículo no encontrado"
        num FINISH = 999
    getReady()
        while item <> FINISH
            findItem()
        endwhile
        finishUp()
    stop

    getReady()
        output "Introduzca el número de artículo o ", FINISH, " para salir "
        input item
    return

    findItem()
        foundIt = "N"
        sub = 0
        while sub < SIZE
            if item = VALID_ITEM[sub] then
                foundIt = "Y"
                price = VALID_PRICE[sub]
            endif
            sub = sub + 1
        endwhile
        if foundIt = "Y" then
            output MSG_YES
            output "El precio de ", item, " es ", price
        else
            output MSG_NO
            badItemCount = badItemCount + 1
        endif
        output "Introduzca el siguiente número de artículo o ", FINISH,
            " para salir "
        input item
    return

    finishUp()
        output badItemCount, " los artículos tienen números inválidos"
    return
```

Figura 6-10 Diagrama de flujo y seudocódigo de un programa que encuentra el precio de un artículo usando arreglos paralelos



Algunos programadores objetan usar un nombre de variable confuso para un subíndice, como `sub` en la figura 6-10, debido a que tales nombres no son descriptivos; preferirían uno como `priceIndex`. Otros aprueban los nombres breves cuando la variable sólo se usa en un área limitada de un programa, como se usa aquí, para pasar por un arreglo. Los programadores están en desacuerdo en muchas cuestiones de estilo como ésta. Como programador, es su responsabilidad averiguar qué convenciones se usan entre sus colegas en una organización.

234

Cuando el programa en la figura 6-10 recibe el pedido de un cliente busca en cada uno de los valores de `VALID_ITEM` por separado variando el subíndice `sub` de 0 hasta el número de artículos disponibles. Cuando se encuentra una correspondencia para el número del artículo, el programa extrae el precio paralelo de la lista de valores `VALID_PRICE` y lo almacena en la variable `price` (véanse las declaraciones sombreadas en la figura 6-10).

La relación entre el número de un artículo y su precio es una **relación indirecta**. Esto significa que usted no tiene acceso a un precio directamente al conocer el número del artículo.

En cambio, determina el precio al saber la posición en el arreglo de dicho número. Una vez que encuentra una correspondencia para el número del artículo ordenado en el arreglo `VALID_ITEM`, sabe que el precio del mismo está en la misma posición en el otro arreglo, `VALID_PRICE`. Cuando `VALID_ITEM[sub]` es el artículo correcto, `VALID_PRICE[sub]` debe ser el precio correcto, así que `sub` vincula los arreglos paralelos.

Los arreglos paralelos son más útiles cuando los pares de valores tienen una relación indirecta. Si los valores en su programa tienen una relación directa es probable que no necesite arreglos paralelos. Por ejemplo, si los artículos estuvieran numerados 0, 1, 2, 3, y así sucesivamente en forma consecutiva, usted podría usar el número de artículo como un subíndice para el arreglo de precio en lugar de un arreglo paralelo para contener los números de los artículos. Aun si éstos estuvieran numerados 200, 201, 202, y así sucesivamente en forma consecutiva, podría restar un valor constante (200) de cada uno y usar éste como subíndice en lugar de un arreglo paralelo.

Suponga que un cliente ordena el artículo 457; recorra la lógica para encontrar el precio correcto por artículo: \$17.50. Luego, suponga que un cliente ordena el artículo 458; recorra la lógica y vea si se despliega el mensaje apropiado *Artículo no encontrado*.

Mejora de la eficiencia de la búsqueda

El programa de pedidos por correo en la figura 6-10 todavía es un poco ineficiente. Cuando un cliente ordena los artículos 106 o 108, se encuentra una correspondencia en el primer o segundo repaso por el ciclo, y la continuación de la búsqueda no proporciona ningún beneficio. Sin embargo, aun después de que se encuentra una correspondencia, el programa en la figura 6-10 sigue buscando a lo largo del arreglo de artículos hasta que `sub` alcanza el valor `SIZE`. Una forma de detener la búsqueda cuando se ha encontrado el artículo y `foundIt` se ha establecido en "Y" es cambiar la pregunta que controla el ciclo. En lugar de sólo continuar el ciclo mientras el número de comparaciones no excede el subíndice del arreglo más alto permitido, debería continuarlo mientras no se encuentre el artículo buscado y el número de comparaciones no haya excedido el máximo. Salir del ciclo tan pronto como se encuentre una correspondencia mejora la eficiencia del programa; entre más grande sea el arreglo, más benéfico es salir del ciclo de búsqueda tan pronto como se encuentre el valor deseado.

La figura 6-11 muestra la versión mejorada del módulo `findItem()` con la pregunta sombreada del ciclo de control alterado.

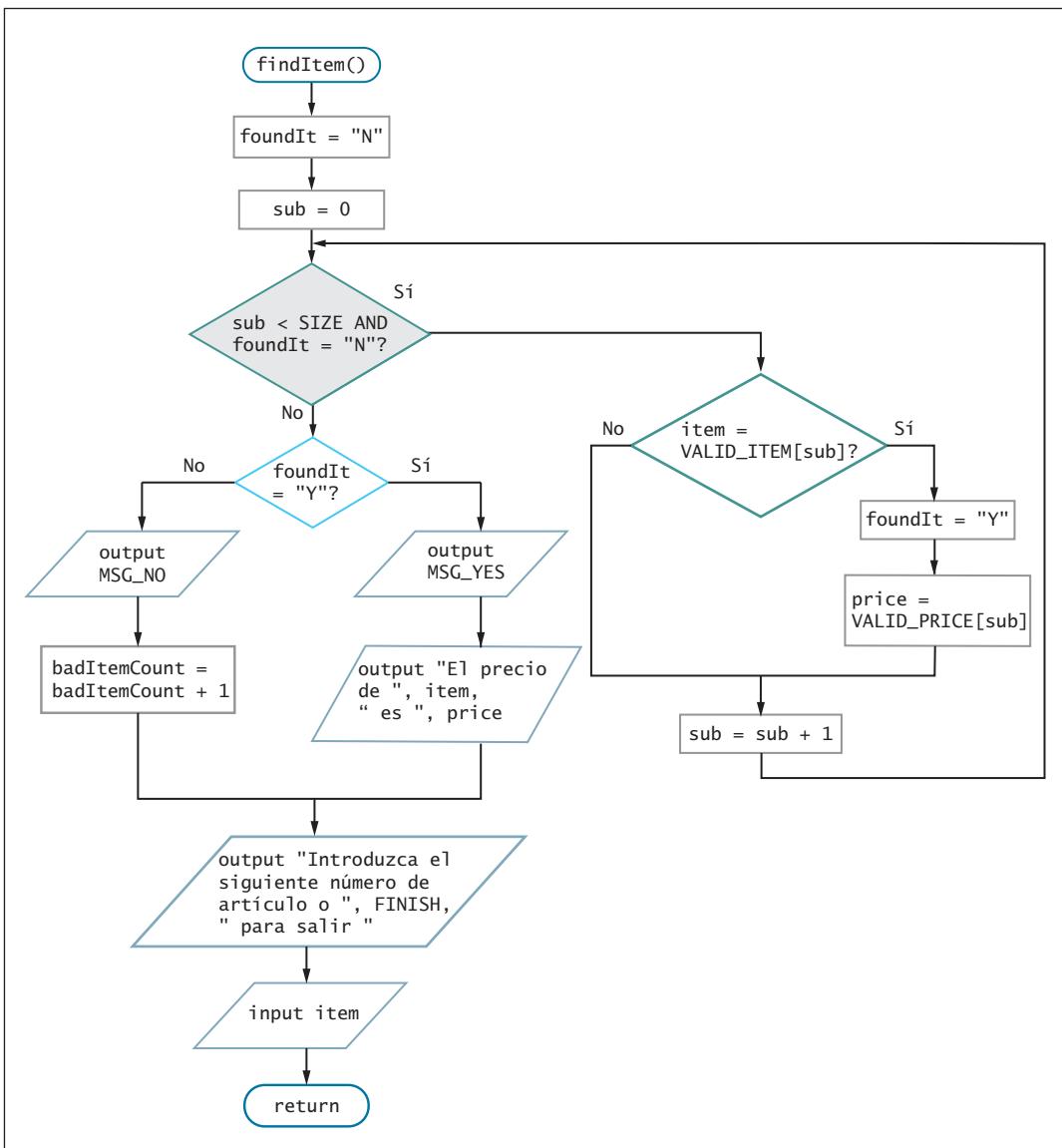


Figura 6-11 Diagrama de flujo y seudocódigo del módulo que encuentra el precio de un artículo y sale del ciclo tan pronto como lo encuentra (continúa)

(continuación)

236

```
findItem()
  foundIt = "N"
  sub = 0
  while sub < SIZE AND foundIt = "N"
    if item = VALID_ITEM[sub] then
      foundIt = "Y"
      price = VALID_PRICE[sub]
    endif
    sub = sub + 1
  endwhile
  if foundIt = "Y" then
    output MSG_YES
    output "El precio de ", item, " es ", price
  else
    output MSG_NO
    badItemCount = badItemCount + 1
  endif
  output "Introduzca el siguiente número de artículo o ", FINISH, " para salir "
  input item
return
```

Figura 6-11 Diagrama de flujo y seudocódigo del módulo que encuentra el precio de un artículo y sale del ciclo tan pronto como lo encuentra

Note que el programa para encontrar el precio ofrece la mayor eficiencia cuando los artículos que se ordenan se almacenan con mayor frecuencia al comienzo del arreglo, de modo que sólo los artículos que se ordenan rara vez requieran muchos ciclos antes de encontrar una correspondencia. A menudo, usted puede mejorar la eficiencia de la búsqueda al reordenar los elementos del arreglo.



Conforme estudie programación aprenderá otras técnicas de búsqueda. Por ejemplo, una **búsqueda binaria** comienza al observar en medio de una lista clasificada y luego determina si debería continuar hacia arriba o hacia abajo.

DOS VERDADES Y UNA MENTIRA

Uso de arreglos paralelos

1. Los arreglos paralelos deben ser del mismo tipo de datos.
2. Los arreglos paralelos por lo general contienen el mismo número de elementos.
3. Usted puede mejorar la eficiencia de búsqueda a lo largo de los arreglos paralelos usando una salida anticipada.

La afirmación falsa es la número 1. Los arreglos paralelos no necesitan ser del mismo tipo de datos. Por ejemplo, usted podría buscar un nombre en un arreglo de cadena para encontrar la edad de cada persona en un arreglo numérico paralelo.

Búsqueda en un arreglo para una correspondencia de rango

Los números de los artículos del pedido del cliente deben corresponder exactamente con los números de los disponibles para determinar su precio correcto. En ocasiones, sin embargo, los programadores desean trabajar con rangos de valores en los arreglos. En el capítulo 4 usted aprendió que un rango de valores es una serie de ellos; por ejemplo, 1 a 5 o 20 a 30.

Suponga que una compañía decide ofrecer descuentos por la cantidad cuando un cliente ordena múltiples artículos, como se muestra en la figura 6-12.

Usted desea leer los datos del pedido del cliente y determinar un porcentaje de descuento con base en la cantidad ordenada. Por ejemplo, si un cliente ha ordenado 20 artículos usted desea dar salida a *Su descuento es de 15%*. Un enfoque incorrecto sería establecer un arreglo con tantos elementos como un cliente podría ordenar alguna vez y almacenar el descuento apropiado para cada número posible, como se muestra en la figura 6-13. Este arreglo está configurado para contener el descuento para 0 artículos, 1 artículo, 2 artículos, y así sucesivamente, y tiene al menos tres inconvenientes:

- Requiere un arreglo muy grande que usa mucha memoria.
- Debe almacenar el mismo valor de manera repetida. Por ejemplo, cada uno de los primeros nueve elementos recibe el mismo valor, 0, y cada elemento de los siguientes cuatro recibe el mismo valor, 10.
- ¿Cómo sabe usted que el arreglo tiene suficientes elementos? ¿Son suficientes 75 artículos para la cantidad del pedido del cliente? ¿Qué pasa si un cliente ordena 100 o 1000 artículos? Sin importar cuántos elementos coloque en el arreglo, siempre hay una probabilidad de que un cliente ordene más.

Cantidad	% de descuento
0-8	0
9-12	10
13-25	15
26 o más	20

Figura 6-12 Descuentos en pedidos por cantidad

```

numeric DISCOUNT[76]
= 0, 0, 0, 0, 0, 0, 0, 0,
  0.10, 0.10, 0.10, 0.10,
  0.15, 0.15, 0.15, 0.15, 0.15,
  0.15, 0.15, 0.15, 0.15, 0.15,
  0.15, 0.15, 0.15,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20

```

No lo haga
Aunque este arreglo es utilizable, es repetitivo, propenso al error y difícil de usar.

Figura 6-13 Arreglo de descuentos utilizable, pero ineficiente

Un enfoque más apropiado es crear dos arreglos paralelos, cada uno con cuatro elementos, como se muestra en la figura 6-14. Cada tasa de descuento se lista una vez en el arreglo DISCOUNT y el extremo inferior de cada rango de cantidad se lista en el arreglo QUAN_LIMIT.

A fin de encontrar el descuento correcto para cualquier cantidad ordenada por el cliente, usted puede comenzar con el último límite de rango de cantidad (QUAN_LIMIT[3]). Si la cantidad ordenada es al menos ese valor, 26, nunca se entra al ciclo y el cliente obtiene la tasa de descuento más alta (DISCOUNT[3], o 20%). Si dicha cantidad no es al menos QUAN_LIMIT[3], es decir, si es menor que 26, entonces usted reduce el subíndice y verifica si la cantidad es al menos QUAN_LIMIT[2], o 13. De ser así, el cliente recibe DISCOUNT[2], o 15%, y así sucesivamente. La figura 6-15 muestra un programa que acepta una cantidad ordenada por un cliente y determina la tasa de descuento apropiada.

```

| num DISCOUNT[4] = 0, 0.10, 0.15, 0.20
| num QUAN_LIMIT[4] = 0, 9, 13, 26

```

Figura 6-14 Arreglos paralelos para usarlos a fin de determinar el descuento

Búsqueda en un arreglo para una correspondencia de rango

239

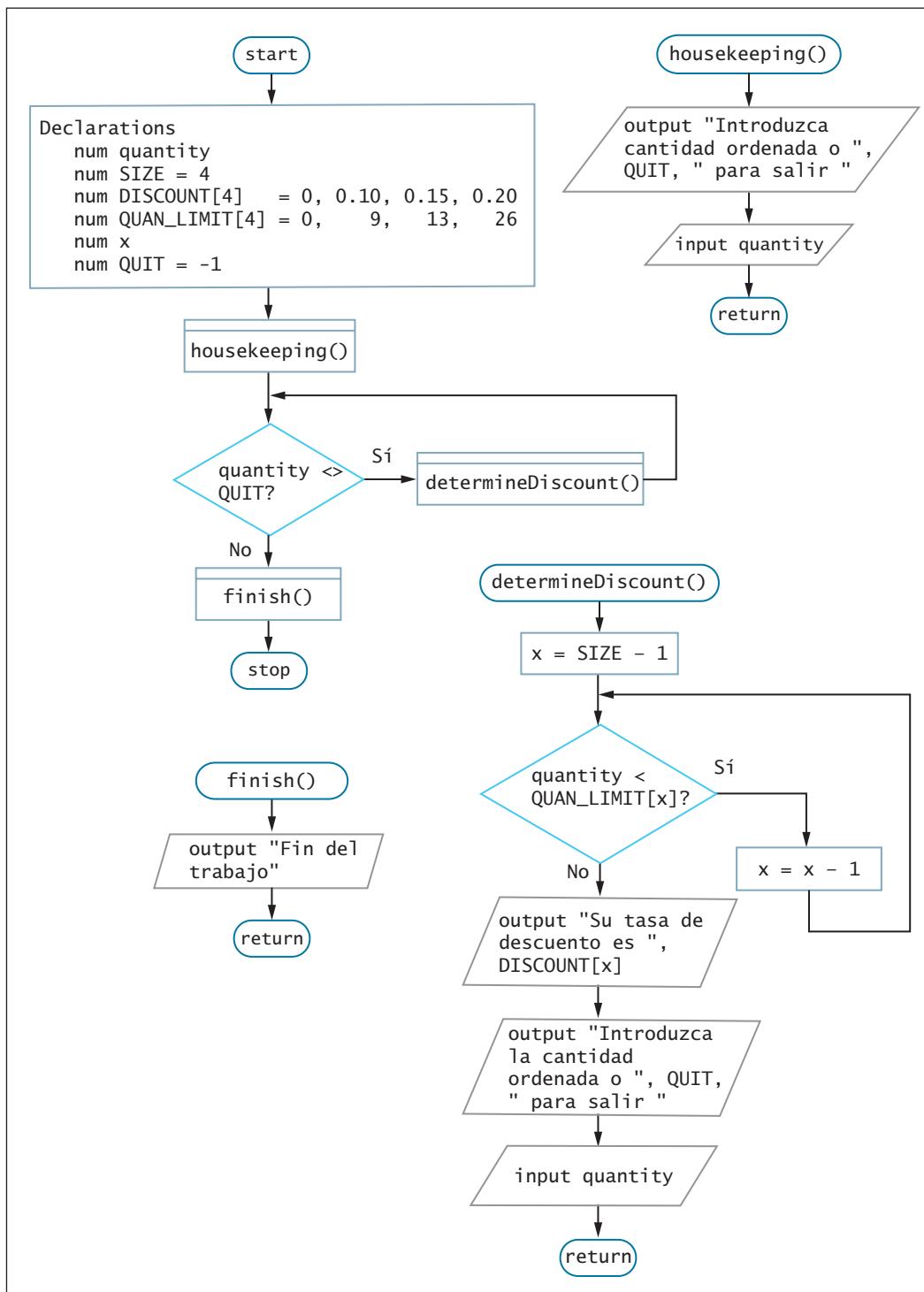


Figura 6-15 Programa que determina la tasa de descuento (continúa)

(continuación)

```
start
    Declarations
        num quantity
        num SIZE = 4
        num DISCOUNT[4] = 0, 0.10, 0.15, 0.20
        num QUAN_LIMIT[4] = 0, 9, 13, 26
        num x
        num QUIT = -1
    housekeeping()
    while quantity <> QUIT
        determineDiscount()
    endwhile
    finish()
stop

housekeeping()
    output "Introduzca la cantidad ordenada o ", QUIT, " para salir "
    input quantity
return

determineDiscount()
    x = SIZE - 1
    while quantity < QUAN_LIMIT[x]
        x = x - 1
    endwhile
    output "Su tasa de descuento es ", DISCOUNT[x]
    output "Introduzca la cantidad ordenada o ", QUIT, " para salir "
    input quantity
return

finish()
    output "Fin del trabajo"
return
```

Figura 6-15 Programa que determina la tasa de descuento

Un enfoque alternativo al que se adoptó en la figura 6-15 es almacenar el extremo superior de cada rango en un arreglo. Luego usted comienza con el elemento *menor* y verifica para valores *menores o iguales que* cada valor de los elementos del arreglo.

Cuando se usa un arreglo para almacenar límites de rangos se utiliza un ciclo para hacer una serie de comparaciones que de otra manera requerirían muchas decisiones separadas. El programa que determina las tasas de descuento del cliente en la figura 6-15 requiere menos instrucciones que uno que no use un arreglo, y las modificaciones al programa serán más fáciles de hacer en el futuro.

DOS VERDADES Y UNA MENTIRA

Búsqueda de un arreglo para una correspondencia de rango

1. Para localizar un rango dentro del que cae un valor, usted puede almacenar el valor más alto en cada rango en un arreglo.
2. Para localizar un rango dentro del que cae un valor, usted puede almacenar el valor más bajo en cada rango en un arreglo.
3. Cuando se usa un arreglo para almacenar límites de rango se usa una serie de comparaciones que de otra manera requerirían muchas estructuras de ciclo separadas.

La afirmación falsa es la número 3. Cuando se usa un arreglo para almacenar límites de rango, se utiliza un ciclo para hacer una serie de comparaciones que de otra manera requerirían muchas decisiones separadas.

Permanencia dentro de los límites del arreglo

Cada arreglo tiene un tamaño finito; usted puede pensar en dicho tamaño en una de dos formas: ya sea por el número de elementos o por el número de bytes en el arreglo. Los arreglos siempre están compuestos por elementos del mismo tipo de datos, y los elementos del mismo tipo de datos siempre ocupan el mismo número de bytes de memoria, así que el número de bytes en un arreglo siempre es un múltiplo del número de elementos que contiene. Por ejemplo, en Java, los enteros ocupan 4 bytes de memoria, así que un arreglo de 10 enteros ocupa exactamente 40 bytes.



Para una exposición completa de los bytes y cómo miden la memoria de la computadora, lea el apéndice A.

En todo lenguaje de programación, cuando se tiene acceso a los datos almacenados en un arreglo es preciso usar un subíndice que contenga un valor que acceda a la memoria ocupada por el arreglo. El subíndice en realidad se multiplica por el tamaño del tipo de datos en bytes y ese valor se agrega a la dirección del arreglo para encontrar la dirección del elemento apropiado. Así, si el subíndice es demasiado grande o demasiado pequeño, el programa intentará acceder a una dirección que no sea parte del espacio del arreglo.

Un error común de los programadores principiantes es olvidar que los subíndices de los arreglos comienzan con 0. Si usted supone que el primer subíndice de un arreglo es 1, siempre estará “desfasado por uno” en el manejo del mismo. Por ejemplo, si trata de manejar un arreglo de 10 elementos usando los subíndices 1 a 10, cometerá dos errores: no tendrá acceso al primer elemento que usa el subíndice 0 e intentará tener acceso a un elemento extra en la posición 10 cuando el subíndice utilizable más alto es 9.

Por ejemplo, examine el programa en la figura 6-16. El método acepta un valor numérico para `monthNum` y despliega el nombre asociado con ese mes. La lógica en la figura 6-16 hace una suposición cuestionable: que cada número introducido por el usuario es el número de un mes válido.

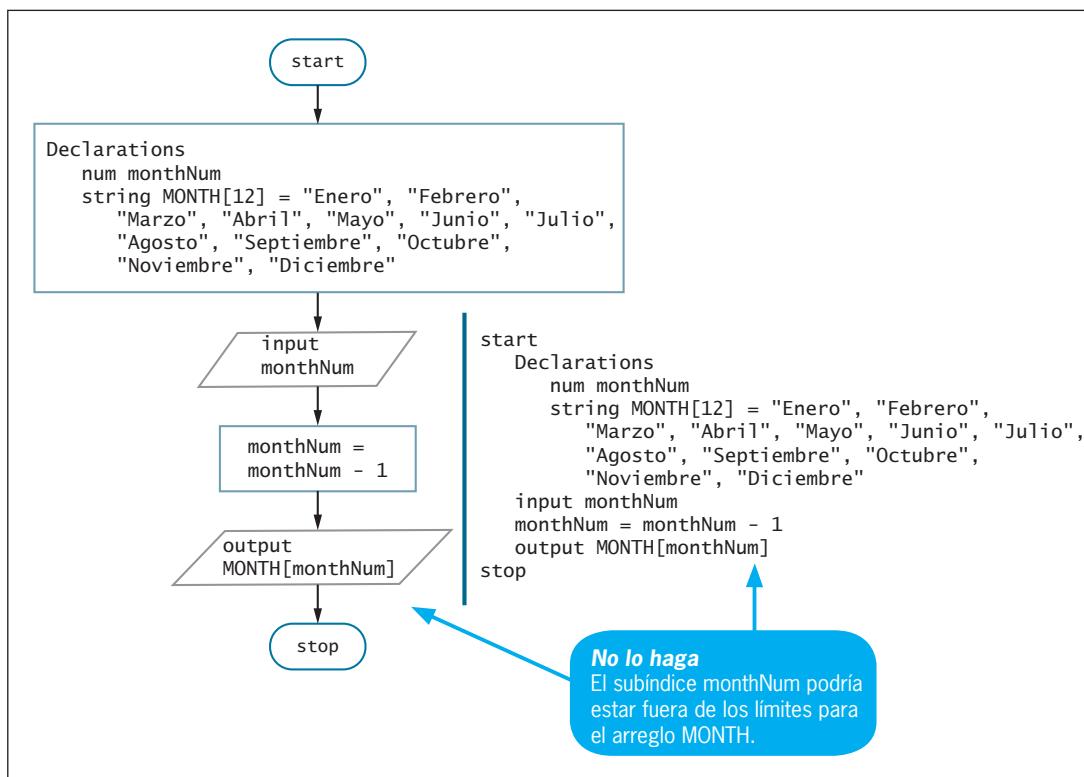


Figura 6-16 Determinación de la cadena del mes a partir de la entrada numérica de un usuario



En el programa de la figura 6-16, note que se resta 1 a `monthNum` antes de usarlo como subíndice. Aunque enero es el primer mes en el año, en el arreglo su nombre ocupa la ubicación con subíndice 0. Con valores que parecen empezar naturalmente con 1, como los números de los meses, algunos programadores preferirían crear un arreglo de 13 elementos y no usar el elemento en la posición cero. De esta manera, cada número de mes “natural” sería el valor correcto para tener acceso a sus datos sin restar. A otros programadores les disgusta desperdiciar memoria creando un elemento extra sin usar en el arreglo. Aunque pueden crearse programas factibles con o sin el elemento extra en el arreglo, los programadores profesionales deberían seguir las convenciones y preferencias de sus colegas y gerentes.

En la figura 6-16, si el usuario introduce un número que es demasiado pequeño o demasiado grande, sucederá una de dos cosas dependiendo del lenguaje de programación que use. Cuando usa un valor de subíndice que es negativo o mayor que el subíndice permitido más alto:

- Algunos lenguajes de programación detendrán la ejecución del programa y emitirán un mensaje de error.

- Otros lenguajes de programación no emitirán un mensaje de error pero tendrán acceso a un valor en una ubicación de memoria que esté fuera del área ocupada por el arreglo. Esta área podría contener basura, o peor, quizás el nombre de un mes incorrecto de manera accidental.

De cualquier manera ocurre un error lógico. Cuando usted usa un subíndice que no está dentro del rango de los aceptables se dice que está **fuerza de límites**. Los usuarios introducen datos incorrectos con frecuencia; un programa adecuado debería ser capaz de manejar el error y no permitir que el subíndice esté fuera de límites.



Un usuario podría introducir un número inválido o no introducir ninguno. En el capítulo 5 usted aprendió que muchos lenguajes tienen un método incorporado con un nombre como `isNumeric()` que puede probar tales errores.

Usted puede mejorar el programa en la figura 6-16 agregando una prueba que asegure que el subíndice que se usa para tener acceso al arreglo está dentro de los límites del mismo. Si encuentra que el valor de entrada no está entre 1 y 12 inclusive, podría adoptar uno de los siguientes enfoques:

- Desplegar un mensaje de error y terminar el programa.
- Usar un valor predeterminado para el mes. Por ejemplo, cuando un mes introducido es inválido, usted quizás suponga que es diciembre.
- Pedir continuamente al usuario un valor nuevo hasta que éste sea válido.

La forma en que maneje un mes inválido depende de los requerimientos de su programa tal como se los haya explicado con detalle su usuario, su supervisor o la política de la compañía.

DOS VERDADES Y UNA MENTIRA

Permanencia dentro de los límites del arreglo

- Los elementos en un arreglo con frecuencia son tipos de datos diferentes, así que es difícil calcular la cantidad de memoria que el arreglo ocupa.
- Si usted intenta tener acceso a un arreglo con un subíndice que es demasiado pequeño, algunos lenguajes de programación detendrán la ejecución del programa y emitirán un mensaje de error.
- Si intenta tener acceso a un arreglo con un subíndice que es demasiado grande, algunos lenguajes de programación acceden a una ubicación de memoria incorrecta fuera de los límites del arreglo.

La afirmación falsa es la número 1. Los elementos del arreglo siempre son del mismo tipo de datos, y los elementos del mismo tipo de datos siempre ocupan el mismo número de bytes de memoria, así que el número de bytes en un arreglo siempre es un múltiplo del número de elementos en un arreglo.

Uso de un ciclo for para procesar arreglos

En el capítulo 5 usted aprendió sobre el ciclo `for` que, en una sola declaración, inicializa una variable de control del ciclo, la compara con un límite y la altera. El ciclo `for` es una herramienta particularmente conveniente cuando se trabaja con los arreglos debido a que con frecuencia necesita procesar cada elemento de los mismos de principio a fin. Igual que con un ciclo `while`, cuando usted usa un ciclo `for`, debe tener cuidado de permanecer dentro de los límites del arreglo, recordando que el subíndice utilizable más alto es uno menos que el tamaño del arreglo. La figura 6-17 muestra un ciclo `for` que despliega en forma correcta todos los nombres de los departamentos de una compañía que están almacenados en un arreglo declarado como `DEPTS`. Note que `dep` se incrementa uno menos que el número de departamentos porque con un arreglo de cinco elementos los subíndices que puede usar son 0 a 4.

```

start
  Declarations
    num dep
    num SIZE = 5
    string DEPTS[SIZE] = "Contabilidad", "Personal",
      "Técnico", "Servicio al cliente", "Mercadotecnia"
    for dep = 0 to SIZE - 1 step 1
      output DEPTS[dep]
    endfor
  stop

```

Figura 6-17 Seudocódigo que usa un ciclo `for` para desplegar un arreglo de nombres de departamento

El ciclo en la figura 6-17 es ligeramente ineficiente porque, mientras se ejecuta cinco veces, la operación de resta que deduce 1 de `SIZE` ocurre todas las veces. Cinco operaciones de resta no consumen mucha energía o mucho tiempo de la computadora, pero en un ciclo que procesa miles o millones de elementos la eficiencia del programa estaría comprometida. La figura 6-18 muestra una solución superior. Una constante nueva llamada `ARRAY_LIMIT` se calcula una vez, luego se usa de manera repetida en la operación de comparación para determinar cuándo detener el ciclo a través del arreglo.

```

start
  Declarations
    num dep
    num SIZE = 5
    num ARRAY_LIMIT = SIZE - 1
    string DEPTS[SIZE] = "Contabilidad", "Personal",
      "Técnico", "Servicio al cliente", "Mercadotecnia"
    for dep = 0 to ARRAY_LIMIT step 1
      output DEPTS[dep]
    endfor
  stop

```

Figura 6-18 Seudocódigo que usa un ciclo `for` más eficiente para dar salida a los nombres de los departamentos

DOS VERDADES Y UNA MENTIRA

Uso de un ciclo for para procesar arreglos

1. El ciclo for es una herramienta particularmente conveniente cuando se trabaja con arreglos.
2. Con frecuencia usted necesita procesar todos los elementos de un arreglo desde el principio hasta el fin.
3. No preocuparse por los límites del arreglo es una ventaja de usar un ciclo for para procesar los elementos del arreglo.

ciclo for tenga cuidado de permanecer dentro de los límites del arreglo.

La afirmación falsa es la número 3. Igual que con un ciclo while, cuando use un

Resumen del capítulo

- Un arreglo es una serie o lista nombrada de valores en la memoria de la computadora, todos ellos tienen el mismo tipo de datos pero se distinguen con subíndices. Cada elemento del arreglo ocupa un área en la memoria junto a los otros o contiguo a ellos.
- Con frecuencia es posible usar una variable como subíndice para un arreglo, lo que permite reemplazar múltiples decisiones anidadas con mucho menos declaraciones.
- Es posible usar constantes para contener el tamaño de un arreglo o para representar sus valores. Usar una constante nombrada para el tamaño de un arreglo hace que el código sea más fácil de entender y con menos probabilidad de contener un error. Los valores del arreglo se declaran como constantes cuando no deberían cambiar durante la ejecución del programa.
- Buscar en un arreglo para encontrar un valor que usted necesita implica inicializar un subíndice, usar un ciclo para probar cada elemento del arreglo y establecer una bandera cuando se encuentre una correspondencia.
- Con los arreglos paralelos, cada elemento en uno de ellos se asocia con el elemento en la misma posición relativa en el otro.
- Cuando usted necesita comparar un valor con un rango de valores en un arreglo, puede almacenar ya sea el extremo inferior o el superior de cada rango para la comparación.
- Cuando se accede a los datos almacenados en un arreglo es importante usar un subíndice que contenga un valor con acceso a la memoria ocupada por el arreglo. Cuando usa un subíndice que no esté dentro del rango definido de subíndices aceptables, se dice que su subíndice está fuera de límites.
- El ciclo for es una herramienta particularmente conveniente cuando se trabaja con los arreglos debido a que con frecuencia necesita procesar todos los elementos de un arreglo de principio a fin.

Términos clave

Un **arreglo** es una serie o lista de valores en la memoria de la computadora, todos ellos tienen el mismo nombre pero se distinguen con números especiales llamados subíndices.

Un **elemento** es un elemento de datos en un arreglo.

El **tamaño del arreglo** es el número de elementos que puede contener.

Un **subíndice**, también llamado **índice**, es un número que indica la posición de un elemento particular dentro de un arreglo.

Poblar un arreglo es la acción de asignar valores a los elementos del arreglo.

Una **búsqueda lineal** es una búsqueda a lo largo de una lista de un extremo a otro.

Una **bandera** es una variable que indica si ha ocurrido algún evento.

En los **arreglos paralelos** cada elemento en uno de ellos se asocia con el que se encuentra en la misma posición relativa en el (los) otro(s) arreglo(s).

Una **relación indirecta** describe la relación entre los arreglos paralelos en que un elemento en el primero no accede directamente a su valor correspondiente en el segundo.

Una **búsqueda binaria** empieza en medio de una lista clasificada y luego determina si debería continuar hacia arriba o hacia abajo para encontrar un valor buscado.

Fuera de límites es un término que describe el subíndice de un arreglo que no está dentro del rango de subíndices aceptables para el mismo.

Preguntas de repaso

1. Un subíndice es un _____.
 - a) elemento en un arreglo
 - b) nombre alterno para un arreglo
 - c) número que representa el valor más alto almacenado dentro de un arreglo
 - d) número que indica la posición de un elemento en un arreglo
2. Cada variable en un arreglo debe tener el (la) mismo(a) _____ que las otras.

a) tipo de datos	c) valor
b) subíndice	d) ubicación en la memoria
3. Cada elemento de datos en un arreglo se llama _____.

a) tipo de datos	c) componente
b) subíndice	d) elemento
4. Los subíndices de cualquier arreglo siempre son _____.

a) enteros	c) caracteres
b) fracciones	d) cadenas de caracteres

5. Suponga que tiene un arreglo llamado `number` y dos de sus elementos son `number[1]` y `number[4]`. Usted sabe que _____.
- los dos elementos contienen el mismo valor
 - el arreglo contiene exactamente cuatro elementos
 - hay exactamente dos elementos entre estos dos elementos
 - los dos elementos están en la misma ubicación de memoria
6. Suponga que desea escribir un programa que introduce los datos del cliente y despliega un resumen del número de clientes que deben más de \$1000 cada uno, en cada una de 12 regiones de ventas. Las variables de los datos del cliente incluyen `name`, `zipCode`, `balanceDue` y `regionNumber`. En algún punto durante el proceso de registro, debe agregar 1 a un elemento del arreglo cuyo subíndice estaría representado por _____.
- `name`
 - `zipCode`
 - `balanceDue`
 - `regionNumber`
7. El tipo de subíndice más útil para manejar los arreglos es un(a) _____.
- constante numérica
 - variable
 - carácter
 - nombre de archivo
8. Un programa contiene un arreglo de siete elementos que contiene los nombres de los días de la semana. Al principio del programa, despliega los nombres del día usando un subíndice llamado `dayNum`. Despliega los mismos valores del arreglo una vez más al final del programa, donde usted _____ como un subíndice para el arreglo.
- debe usar `dayNum`
 - puede usar `dayNum`, pero también puede usar otra variable
 - no debe usar `dayNum`
 - debe usar una constante numérica en lugar de una variable
9. Suponga que ha declarado un arreglo como sigue: `num values[4] = 0, 0, 0, 0`. ¿Cuál de las siguientes es una operación permitida?
- `values[2] = 17`
 - `input values[0]`
 - `values[3] = values[0] + 10`
 - todas las anteriores
10. Llenar un arreglo con valores durante la ejecución de un programa se conoce como _____.
- ejecutarlo
 - colonizarlo
 - poblarlo
 - declararlo

11. Usar un arreglo puede hacer que un programa sea _____.
a) más fácil de entender c) difícil de mantener
b) ilegal en algunos lenguajes modernos d) todos los anteriores

12. Un _____ es una variable que puede establecerse para indicar si algún evento ha ocurrido.
a) subíndice c) contador
b) pancarta d) bandera

13. ¿Cómo llama a dos arreglos en los que cada elemento en un arreglo está asociado con el elemento en la misma posición relativa en el otro arreglo?
a) arreglos cohesivos c) arreglos ocultos
b) arreglos paralelos d) arreglos perpendiculares

14. En la mayoría de los lenguajes de programación modernos, el subíndice más alto que debería usar con un arreglo de 12 elementos es _____.
a) 10 c) 12
b) 11 d) 13

15. Los arreglos paralelos _____.
a) con frecuencia tienen una relación indirecta
b) nunca tienen una relación indirecta
c) deben ser del mismo tipo de datos
d) no deben ser del mismo tipo de datos

16. Cada elemento en un arreglo de siete elementos puede contener _____ valor(es).
a) un c) al menos siete
b) siete d) un número ilimitado de

17. Después de la exposición anual de perros en los que la Academia de Entrenamiento para Perros Barkley concede puntos a cada participante, la academia asigna una posición a cada perro con base en los criterios del cuadro 6-1.

Puntos ganados	Nivel de logro
0-5	Bueno
6-7	Excelente
8-9	Superior
10	Increíble

Cuadro 6-1 Niveles de logro de la Academia de Entrenamiento para Perros Barkley

La academia necesita un programa que compare los puntos obtenidos por un perro con la escala de calificación, de modo que cada perro reciba un certificado que reconozca el nivel de logro apropiado. De los siguientes, ¿cuál conjunto de valores sería más útil para el contenido de un arreglo que se usa en el programa?

- a) 0, 6, 9, 10
 - c) 5, 7, 9, 10
 - b) 5, 7, 8, 10
 - d) cualquiera de los anteriores
18. Cuando usted usa el valor de un subíndice que es negativo o mayor que el número de elementos en un arreglo, _____.
- a) la ejecución del programa se detiene y se emite un mensaje de error
 - b) se tendrá acceso a un valor en una ubicación de memoria que está fuera del área ocupada por el arreglo
 - c) se tendrá acceso a un valor en una ubicación de memoria que está fuera del área ocupada por el arreglo, pero sólo si el valor es del tipo de datos correcto
 - d) la acción resultante depende del lenguaje de programación que se usa
19. En todos los arreglos, un subíndice está fuera de límites cuando es _____.
- a) negativo
 - c) 1
 - b) 0
 - d) 999
20. Usted puede tener acceso a cada elemento de un arreglo usando un _____.
- a) ciclo `while`
 - c) los dos anteriores
 - b) ciclo `for`
 - d) ninguno de los anteriores

Ejercicios

1. a) Diseñe la lógica para un programa que permita a un usuario introducir 15 números y que luego los despliegue en el orden inverso al de entrada.
b) Modifique el programa de despliegue invertido de modo que el usuario pueda introducir cualquier cantidad de números menores que 15 hasta que se introduzca un valor centinela.
2. a) Diseñe la lógica para un programa que permita al usuario introducir 15 números, luego despliegue cada número y su diferencia a partir del promedio numérico de los números introducidos.
b) Modifique el programa del ejercicio 2a de modo que el usuario pueda introducir cualquier cantidad de números menores que 15 hasta que se introduzca un valor centinela.
3. a) Los empleados del registro en una conferencia para autores de libros infantiles han recopilado los datos sobre los participantes, incluyendo el número de libros que cada autor ha escrito y la edad de los lectores a los que se dirigen. Los participantes han escrito de 1 a 40 libros cada uno y las edades de los lectores tenían un rango de 0 a 16. Diseñe un programa que acepte en forma continua el número de libros

- escritos hasta que se introduzca un valor centinela y luego despliegue una lista de cuántos participantes han escrito cada número de libros (1 a 40).
- b) Modifique el programa de registro de autores de modo que se introduzca una edad para la audiencia de cada autor hasta que se ingrese un valor centinela. La salida es un conteo del número de libros escritos para cada uno de los siguientes grupos de edad: menos de 3, 3 a 7, 8 a 10, 11 a 13 y 14 y mayores.
4. a) El Estudio de Yoga Downdog ofrece cinco tipos de clases, como se muestra en el cuadro 6-2. Diseñe un programa que acepte un número que represente una clase y luego despliegue el nombre de la misma.
- b) Modifique el programa del Estudio de Yoga Downdog de modo que sea posible introducir continuamente solicitudes de clase numérica hasta que se introduzca un valor centinela. Luego despliegue cada número de clase, nombre y un conteo del número de solicitudes para cada clase.
- | Número de clase | Nombre de la clase |
|-----------------|---------------------------|
| 1 | Yoga 1 |
| 2 | Yoga 2 |
| 3 | Yoga para niños |
| 4 | Yoga prenatal |
| 5 | Yoga para adultos mayores |
- Cuadro 6-2** Clases del Estudio de Yoga Downdog
5. a) La Escuela Primaria Watson tiene 30 salones de clases numerados del 1 al 30. Cada salón puede contener cualquier número de estudiantes hasta 35. Cada estudiante presenta una prueba de aprovechamiento al final del año escolar y recibe una puntuación de 0 a 100. Escriba un programa que acepte los datos de cada estudiante en la escuela: identificación, número de salón y puntuación en la prueba de aprovechamiento. Diseñe un programa que liste los puntos totales logrados por cada uno de los 30 salones.
- b) Modifique el programa de la Escuela Primaria Watson de modo que dé salida al promedio de las puntuaciones de la prueba para cada salón de clases, en lugar de las puntuaciones totales para cada salón.
6. La cafetería Jumpin' Jive cobra \$2.00 por una taza de café y ofrece los agregados que se muestran en el cuadro 6-3.
- Diseñe la lógica para una aplicación que permita a un usuario introducir en forma continua los agregados ordenados hasta que se ingrese un valor centinela. Después de cada artículo, despliegue su precio o el mensaje *Lo sentimos, no lo tenemos* como salida. Después de introducir todos los artículos, despliegue el precio total para la orden.
7. Diseñe la lógica de la aplicación para una compañía que desea un informe que contenga un desglose de la nómina por departamento. La entrada incluye el número de
- | Producto | Precio (\$) |
|---------------------|-------------|
| Crema batida | 0.89 |
| Canela | 0.25 |
| Jarabe de chocolate | 0.59 |
| Amaretto | 1.50 |
| Whisky irlandés | 1.75 |
- Cuadro 6-3** Lista de agregados de la cafetería Jumpin' Jive

departamento de cada empleado, salario por hora y número de horas trabajadas. La salida es una lista de los siete departamentos en la compañía y la nómina bruta total (tarifa por horas) para cada departamento. Los nombres de los departamentos se muestran en el cuadro 6-4.

8. Diseñe un programa que calcule el pago para los empleados; permita que un usuario introduzca en forma continua los nombres de los mismos hasta que se ingrese un valor centinela apropiado. También introduzca el salario por hora de cada empleado y las horas trabajadas. Calcule el pago bruto de cada uno (horas por salario), retenga el porcentaje de impuesto (basado en el cuadro 6-5), retenga la cantidad de impuesto y el pago neto (pago bruto menos impuesto retenido). Despliegue todos los resultados para cada empleado. Después de que el último de ellos se ha ingresado, despliegue la suma de todas las horas trabajadas, la nómina bruta total, el total retenido para todos los empleados y la nómina neta total.
9. Countrywide Tours realiza viajes turísticos para grupos desde su sede en Iowa. Cree una aplicación que acepte en forma continua los datos de los viajes incluyendo un número de viaje de tres dígitos; los valores numéricos del mes, día y año que representen la fecha de su inicio; el número de viajeros, y un código numérico que represente el destino. Conforme se introducen los datos para cada viaje, verifique que el mes, día, año y código de destino son válidos; si cualquiera de éstos no lo es continúe pidiendo al usuario hasta que se introduzcan datos válidos. Los códigos de destino válidos se muestran en el cuadro 6-6.

Diseñe la lógica para una aplicación que dé como salida el número de cada viaje, la fecha de salida validada, el código de destino, el nombre del destino, el número de viajeros, el precio total bruto y el precio del viaje después del descuento. El precio total bruto es el precio del viaje por viajero multiplicado por el número de viajeros. El precio

Número de departamento	Nombre del departamento
1	Personal
2	Mercadotecnia
3	Manufactura
4	Servicios de cómputo
5	Ventas
6	Contabilidad
7	Embarque

Cuadro 6-4 Números y nombres de los departamentos

Pago semanal bruto (\$)	Porcentaje de retención (%)
0.00–300.00	10
300.01–550.00	13
550.01–800.00	16
800.01 en adelante	20

Cuadro 6-5 Porcentaje de retención basado en el pago bruto

Código	Destino	Precio por persona (\$)
1	Chicago	300.00
2	Boston	480.00
3	Miami	1050.00
4	San Francisco	1300.00

Cuadro 6-6 Códigos y precios de Countrywide Tours

final incluye un descuento para cada persona en los grupos grandes con base en el cuadro 6-7.

10. a) *Daily Life Magazine* desea analizar las características demográficas de sus lectores. El departamento de mercadotecnia ha recabado registros de las encuestas con los lectores que contienen la edad, el género, el estado civil y el ingreso anual de los lectores. Diseñe una aplicación que permita a un usuario introducir los datos de los lectores y, cuando la entrada de datos esté completa, produzca un registro de ellos por grupos de edad como sigue: menores de 20, 20-29, 30-39, 40-49 y 50 y mayores.
- b) Modifique el programa de *Daily Life Magazine* de modo que produzca un registro de lectores por género dentro de cada grupo de edad; es decir, mujeres menores de 20, hombres menores de 20, y así sucesivamente.
- c) Modifique el programa de *Daily Life Magazine* de modo que produzca un registro de lectores por grupos de ingresos (en dólares) como sigue: menos de \$30,000, \$30,000-\$49,999, \$50,000-\$69,999 y \$70,000 y más.

11. Venta de Propiedades Vacacionales Glen Ross emplea siete vendedores, como se muestra en el cuadro 6-8.

Cuando un vendedor hace una venta se crea un registro, incluyendo la fecha, hora y cantidad en dólares; la hora se expresa en horas y minutos, con base en un reloj de 24 horas. La cantidad de la venta se menciona en dólares enteros. El vendedor gana una comisión que difiere para cada venta, con base en el programa de tasas en el cuadro 6-9.

Diseñe una aplicación que produzca:

- a) Una lista del número, el nombre, las ventas totales y las comisiones totales de cada vendedor.
- b) Una lista de cada mes del año tanto en número como en palabra (por ejemplo, *01 Enero*) y las ventas totales del mes para todos los vendedores.
- c) Una lista de las ventas totales y de las comisiones totales ganadas por todos

Número de turistas	Descuento por turista (\$)
1-5	0
6-12	75
13-20	125
21-50	200
51 y más	300

Cuadro 6-7 Descuentos de Countrywide Tours

Número de identificación	Nombre del vendedor
103	Darwin
104	Kratz
201	Shulstad
319	Fortune
367	Wickert
388	Miller
435	Vick

Cuadro 6-8 Vendedores de Glen Ross

Cantidad de la venta (\$)	Tasa de comisión (%)
0-50,999	4
51,000-125,999	5
126,000-200,999	6
201,000 en adelante	7

Cuadro 6-9 Programa de comisiones de Glen Ross

los vendedores para cada uno de los siguientes marcos de tiempo, con base en la hora del día: 00-05, 06-12, 13-18 y 19-23.

12. Diseñe una aplicación en la que el número de días para cada mes en el año se almacene en un arreglo (por ejemplo, enero tiene 31 días, febrero 28, y así sucesivamente). Suponga que el año no es bisiesto). Indique a un usuario que introduzca un mes y día de nacimiento, y continúe solicitándolo hasta que el día introducido esté en un rango para el mes. Calcule la posición numérica del día en el año (por ejemplo, 2 de febrero es el día 33). Luego, con arreglos paralelos, encuentre y despliegue el signo del zodiaco tradicional para la fecha, digamos, el signo para el 2 de febrero es Acuario.



Encuentre los errores

13. Sus archivos descargables para el capítulo 6 incluyen DEBUG06-01.txt, DEBUG06-02.txt y DEBUG06-03.txt. Cada archivo comienza con algunos comentarios que describen el problema. Los comentarios son líneas que comienzan con dos diagonales (//). Después de los comentarios, cada archivo contiene pseudocódigo que tiene uno o más errores que usted debe encontrar y corregir. (NOTA: Estos archivos se encuentran disponibles sólo para la versión original en inglés.)



Zona de juegos

14. Cree la lógica para un juego Bola Mágica 8 en el que el usuario introduce una pregunta como: *¿Qué me depara el futuro?* La computadora selecciona al azar una de ocho posibles respuestas imprecisas, como *Está por verse*.
15. Cree la lógica para una aplicación que contenga un arreglo de 10 preguntas de opción múltiple relacionadas con su pasatiempo favorito. Cada pregunta contiene tres opciones de respuesta. También cree un arreglo paralelo que contenga la respuesta correcta para cada pregunta: A, B o C. Despliegue cada pregunta y verifique que el usuario introduce sólo A, B o C como respuesta; si no, siga pidiéndole una entrada hasta que se introduzca una respuesta válida. Si el usuario responde una pregunta correctamente, despliegue *Correcto!*; de lo contrario, despliegue *La respuesta correcta es* y la letra de la respuesta correcta. Después de que el usuario responde todas las preguntas, despliegue el número de respuestas correctas e incorrectas.
16. a) Cree la lógica para un juego de dados. La aplicación “lanza” aleatoriamente cinco dados para la computadora y cinco dados para el jugador; después de cada tiro aleatorio, almacene los resultados en un arreglo. La aplicación despliega todos los valores, que pueden ser de 1 a 6 inclusive para cada dado. Decida el ganador con base en la siguiente jerarquía de valores del dado; cualquier combinación más alta vence a una más baja; por ejemplo, cinco iguales vencen a cuatro iguales.

- ◆ Cinco iguales
- ◆ Cuatro iguales
- ◆ Tres iguales
- ◆ Un par

Para este juego los valores numéricos de los dados no cuentan. Por ejemplo, si ambos jugadores tienen tres iguales es un empate sin importar cuáles sean los valores de los tres dados. Además, el juego no reconoce un full (tres iguales más dos iguales). La figura 6-19 muestra cómo podría jugarse en un ambiente de línea de comandos.

- b) Mejore el juego de dados de modo que cuando ambos jugadores tengan el mismo número de dados iguales, el valor más alto gane. Por ejemplo, dos 6 vencen a dos 5.
17. Diseñe la lógica para el juego del “Ahorcado”, en el que el usuario adivina las letras en una palabra oculta. Almacene las letras de una palabra en un arreglo de caracteres; despliegue una raya para cada letra faltante. Permita que el usuario adivine en forma continua una letra hasta que todas las letras en la palabra se adivinen. Conforme el usuario introduce cada adivinanza, despliegue la palabra de nuevo, llenando la letra adivinada si fue correcta. Por ejemplo, si la palabra oculta es *computadora*, primero despliegue una serie de once rayas: ----- . Después de que el usuario adivina *p*, el despliegue se vuelve ---*p*----- . Asegúrese de que cuando un usuario adivina, se llenen todas las letras correspondientes. Por ejemplo, si la palabra es *banana* y el usuario adivina *a*, deberán llenarse los tres caracteres *a*.
18. Cree dos arreglos paralelos que representen un mazo estándar de 52 naipes. Un arreglo es numérico y contiene los valores 1 a 13 (que representan As, 2 a 10, Sota, Reina y Rey). El otro es un arreglo de cadena que contiene palos (Tréboles, Diamantes, Corazones y Picas). Cree los arreglos de modo que se representen los 52 naipes. Luego, cree un juego de naipes Guerra que seleccione al azar dos naipes (uno para el jugador y uno para la computadora) y declare un ganador o un empate con base en el valor numérico de los dos naipes. El juego debería durar 26 rondas y usar un mazo completo sin naipes repetidos. Para este juego, suponga que el naipe menor es el As. Despliegue los valores de los naipes del jugador y de la computadora, compare sus valores y determine el ganador. Cuando todos los naipes en el mazo se agoten, despliegue un conteo del número de veces que ganó el jugador, el número de veces que ganó la computadora y el número de empates.

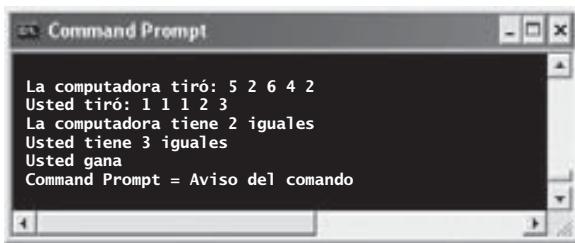


Figura 6-19 Ejecución típica del juego de dados

Aquí hay algunas sugerencias:

- Empiece creando un arreglo para los 52 naipes.
- Seleccione un número aleatorio para la posición del mazo del primer naipe del jugador y asigne el naipe en esa posición del arreglo para el jugador.
- Mueva todos los naipes con posiciones más altas en el mazo “hacia abajo” un lugar para llenar el hueco. En otras palabras, si el primer número aleatorio del jugador es 49, seleccione el naipe en la posición 49 (tanto el valor numérico como la cadena), mueva el naipe que estaba en la posición 50 a la posición 49 y el que estaba en la posición 51 a la posición 50. Sólo quedan 51 naipes en el mazo después de que se reparte el primer naipe del jugador, así que el arreglo de naipes disponibles se reduce en uno.
- De la misma manera, seleccione al azar un naipe para la computadora y “retire” el del mazo.



Para discusión

19. El horario de un tren es el ejemplo de un arreglo de la vida real. Identifique al menos cuatro más.
20. Cada elemento en un arreglo siempre tiene el mismo tipo de datos. ¿Por qué es necesario esto?

7

CAPÍTULO

Manejo de archivos y aplicaciones

En este capítulo usted aprenderá sobre:

- ◎ Archivos de computadora
- ◎ Jerarquía de datos
- ◎ Ejecutar operaciones con archivos
- ◎ Archivos secuenciales y lógica de control de interrupciones
- ◎ Unir archivos
- ◎ Procesamiento de archivos maestros y de transacción
- ◎ Archivos de acceso aleatorio

Comprensión de los archivos de computadora

En el capítulo 1 usted aprendió que la memoria de la computadora, o la memoria de acceso aleatorio (RAM), es un almacenamiento temporal volátil. Cuando escribe un programa que almacena un valor en una variable usa el almacenamiento temporal; el valor que almacena se pierde cuando el programa termina o la computadora se apaga.

El almacenamiento permanente no volátil, por otra parte, no se pierde cuando una computadora se apaga. Cuando escribe un programa y lo guarda en un disco usa el almacenamiento permanente.



Cuando se analiza el almacenamiento en una computadora, los términos *temporal* y *permanente* se refieren a la volatilidad, no a la duración. Por ejemplo, una variable *temporal* podría estar por varias horas en un programa muy grande o en uno que corre en un ciclo infinito, pero un usuario podría guardar y luego eliminar una pieza de datos *permanente* en algunos segundos. Debido a que usted puede borrar los datos de los archivos, algunos programadores prefieren el término *almacenamiento persistente* al de almacenamiento permanente. En otras palabras, puede eliminar datos de un archivo almacenado en un dispositivo, como una unidad de disco, de modo que técnicamente no es permanente. Sin embargo, los datos permanecen en el archivo aun cuando la computadora pierde energía, así que, a diferencia de la RAM, los datos persisten.

Un **archivo de computadora** es un conjunto de datos almacenados en un dispositivo no volátil en un sistema de cómputo. Los archivos existen en los **dispositivos de almacenamiento permanentes**, como discos duros, DVD, unidades USB y carretes de cinta magnética. Las dos extensas categorías de archivos son:

- **Archivos de texto**, contienen datos que pueden ser leídos en un editor de texto porque se han codificado usando un esquema como ASCII o Unicode; estos archivos podrían incluir hechos y cifras que utilizan los programas de negocios, como un archivo de nómina que contiene números de empleados, nombres y salarios. Los programas en este capítulo usarán archivos de texto.
- **Archivos binarios**, contienen datos que no se han codificado como texto; los ejemplos incluyen imágenes y música.

Aunque su contenido varía, los archivos tienen muchas características comunes:

- Cada archivo tiene un nombre. El nombre con frecuencia incluye un punto y una extensión de archivo que describe el tipo del mismo; por ejemplo, la extensión *.txt* se refiere a un archivo de texto sencillo, *.dat* es común para un archivo de datos y *.jpg* se usa en los archivos de imagen en formato del Grupo Conjunto de Expertos en Fotografía (*Joint Photographic Experts Group*).
- Cada archivo tiene tiempos específicos asociados; por ejemplo, el de su creación y el de su última modificación.
- Cada archivo ocupa espacio en una sección de un dispositivo de almacenamiento; es decir, cada archivo tiene un tamaño; y los tamaños se miden en bytes. Un **byte** es una unidad pequeña de almacenamiento; por ejemplo, en un archivo de texto simple, un byte contiene sólo un carácter. Debido a que un byte es tan pequeño, los tamaños de los archivos por lo general se expresan en **kilobytes** (miles de bytes), **megabytes** (millones de bytes) o **gigabytes** (miles de millones de bytes). El apéndice A contiene más información sobre los bytes y cómo se expresan los tamaños de los archivos.

La figura 7-1 muestra cómo son algunos archivos cuando usted los ve en Microsoft Windows.

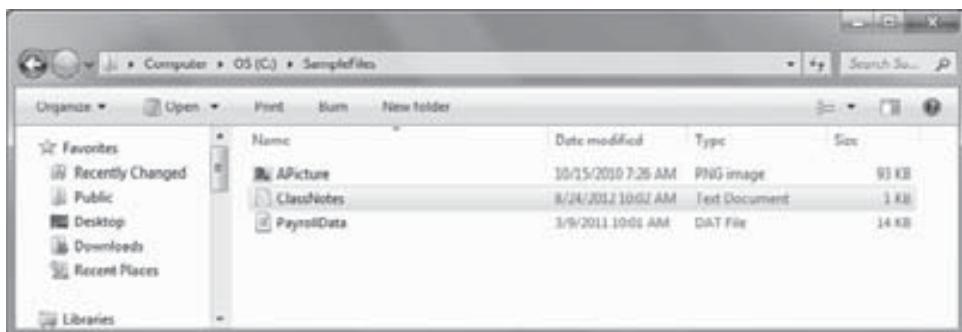


Figura 7-1 Tres archivos almacenados y sus atributos

Organización de los archivos

Los archivos de computadora en un dispositivo de almacenamiento son el equivalente electrónico de los que son de papel y se guardan en archiveros. Cuando se trata de papel, la forma más fácil de almacenar un documento es colocarlo en el cajón de un archivero sin una carpeta. Sin embargo, para lograr una mejor organización, la mayoría de los empleados de oficina colocan los documentos de papel en carpetas; y la mayoría de los usuarios de computadoras organizan sus archivos en carpetas o directorios. Los **directorios** y **carpetas** son unidades de organización en los dispositivos de almacenamiento; cada uno puede contener múltiples archivos al igual que directorios adicionales. La combinación de la unidad de disco más la jerarquía completa de directorios en los que reside un archivo es una **ruta**. Por ejemplo, en el sistema operativo Windows, la siguiente línea sería la ruta completa para un archivo llamado PayrollData.dat en la unidad C en una carpeta llamada SampleFiles dentro de una carpeta llamada Logic:

C:\Logic\SampleFiles\PayrollData.dat



Los términos *directorio* y *carpeta* se usan como sinónimos para referirse a una entidad que organiza los archivos. Directorio es el término más general; carpeta se usa en los sistemas gráficos. Por ejemplo, Microsoft comenzó a llamar carpetas a los directorios con la introducción de Windows 95.

DOS VERDADES Y UNA MENTIRA

Comprensión de los archivos de computadora

1. El almacenamiento temporal es volátil.
2. Los archivos de computadora existen en dispositivos de almacenamiento permanentes, como la RAM.
3. La ruta de un archivo es la jerarquía de carpetas en las que está almacenado.

La afirmación falsa es la número 2. Los archivos de computadora existen en los dispositivos de almacenamiento permanente, como discos duros, DVD, unidades USB y carretes de cinta magnética.

Comprensión de la jerarquía de datos

Cuando los negocios almacenan elementos de datos en los sistemas de cómputo, con frecuencia lo hacen en una estructura llamada **jerarquía de datos** que describe las relaciones entre los componentes de los mismos. La jerarquía de datos consiste en lo siguiente:

- Los **caracteres** son letras, números y símbolos especiales, como A, 7 y \$. Cualquier cosa que usted pueda ingresar desde el teclado con un solo golpe es un carácter, incluyendo los que en apariencia son “vacíos” como los espacios y los tabuladores. Las computadoras también reconocen caracteres que usted no puede introducir desde un teclado estándar, como los de alfabetos extranjeros como φ o Σ. Los caracteres están formados por elementos más pequeños llamados bits, pero del mismo modo en que la mayoría de los seres humanos pueden usar un lápiz sin preocuparse por si los átomos vuelan en su interior, los usuarios de computadoras almacenan los caracteres sin pensar en estos bits.
- Los **campos** son elementos de datos que representan un solo atributo de un registro y se componen de uno o más caracteres; incluyen elementos como lastName, middleInitial, streetAddress o annualSalary.
- Los **registros** son grupos de campos que están juntos por alguna razón lógica. Un nombre, dirección y salario aleatorios no son muy útiles, pero si son *su* nombre, *su* dirección y *su* salario, entonces se trata de *su* registro. El registro de un inventario podría contener campos para el número, color, tamaño y precio de un artículo; el de un estudiante quizás contenga un número de identificación, promedio de calificaciones y especialidad.
- Los **archivos** son grupos de registros relacionados. Los registros individuales de cada estudiante en una clase podrían estar juntos en un archivo llamado Students.dat. Del mismo modo, los registros de cada persona en una empresa podrían encontrarse en un archivo llamado Personnel.dat. Algunos archivos pueden tener sólo algunos registros; por ejemplo, uno de estudiantes de un seminario universitario podría tener sólo 10 registros. Otros, como el archivo de tenedores de tarjetas de crédito para una cadena de tiendas de departamentos importante o tenedores de pólizas de una compañía grande de seguros, pueden contener miles o incluso millones de registros.



Una **base de datos** contiene grupos de archivos o **tablas** que en conjunto sirven a las necesidades de información de una organización. El software de bases de datos establece y mantiene relaciones entre los campos en estas tablas, de modo que los usuarios coloquen juntos los elementos de datos relacionados en un formato que permita a los gerentes tomar decisiones de manera más eficiente.

DOS VERDADES Y UNA MENTIRA

261

Comprensión de la jerarquía de datos

1. En la jerarquía de datos, un campo es un solo elemento de datos, como `lastName`, `streetAddress` o `annualSalary`.
2. En la jerarquía de datos, los campos se agrupan para formar un registro; los registros son grupos de campos que están juntos por alguna razón lógica.
3. En la jerarquía de datos, los registros relacionados se agrupan para formar un campo.

La afirmación falsa es la número 3. Los registros relacionados forman un archivo.

Ejecución de operaciones con archivos

Para usar los archivos de datos en sus programas, usted necesita entender varias operaciones que se realizan con ellos:

- Declarar un archivo
- Abrir un archivo
- Leer datos de un archivo
- Escribir datos en un archivo
- Cerrar un archivo

Declarar un archivo

La mayoría de los lenguajes soportan varios tipos de archivos, pero una forma de clasificarlos consiste en determinar si pueden usarse para entrada o para salida. Del mismo modo en que las variables y las constantes tienen tipos de datos como `num` y `string`, cada archivo tiene un tipo de datos que se define en el lenguaje que usted use. Por ejemplo, uno podría ser `InputFile`. Del mismo modo que las variables y las constantes, los archivos se declaran cuando se da a cada uno un tipo de datos y un identificador. Como ejemplos, usted podría declarar dos archivos como sigue:

```
InputFile employeeData
OutputFile updatedData
```

Los tipos `InputFile` y `OutputFile` se escriben con mayúsculas y minúsculas en este libro debido a que sus equivalentes así se escriben en la mayoría de los lenguajes de programación.

Este enfoque ayuda a distinguir estos tipos complejos de los tipos simples como `num` y `string`. Los identificadores que se dan a los archivos, como `employeeData` y `updatedData`, son internos al programa, del mismo modo que los nombres de las variables. Para hacer que un programa lea los datos de un archivo desde un dispositivo de almacenamiento, usted también necesita asociar el nombre del archivo interno del programa con el nombre del sistema operativo para el archivo; con frecuencia, esta asociación se consigue cuando éste se abre.

Abrir un archivo

En la mayoría de los lenguajes de programación, antes de que una aplicación pueda usar un archivo de datos es preciso **abrir el archivo**. Esta acción lo localiza en un dispositivo de almacenamiento y asocia con el archivo un nombre de variable dentro de su programa. Por ejemplo, si el identificador `employeeData` se ha declarado como tipo `InputFile`, entonces podría hacer una declaración parecida a la siguiente:

```
open employeeData "EmployeeData.dat"
```

Esta declaración asocia el archivo nombrado `EmployeeData.dat` en el dispositivo de almacenamiento con el nombre interno del programa `employeeData`. Por lo general, usted también puede especificar una ruta más completa cuando el archivo de datos no está en el mismo directorio que el programa, como en el siguiente:

```
open employeeData "C:\CompanyFiles\CurrentYear\Employeedata.dat"
```

Leer datos de un archivo

Antes de que usted pueda usar los datos almacenados en un programa debe cargarlos en la memoria de la computadora. Nunca use directamente los valores de datos que estén almacenados en un dispositivo de almacenamiento; en cambio, use una copia que se transfiera a la memoria. Cuando usted copia los datos de un archivo que se encuentran en un dispositivo de almacenamiento a la RAM, usted **lee del archivo**.



De manera especial, cuando los elementos de datos están almacenados en un disco duro su ubicación quizás no sea clara para usted: los datos sólo parecen estar “en la computadora”. Para un usuario ocasional, las líneas entre almacenamiento permanente y memoria temporal con frecuencia son confusas debido a que muchos programas nuevos guardan por usted los datos en forma automática y periódica sin pedirle permiso. Sin embargo, en cualquier momento, la versión de un archivo en la memoria podría diferir de la que se guardó por última vez en un dispositivo de almacenamiento.

Si los elementos de datos se han almacenado en un archivo y un programa los necesita, usted puede escribir declaraciones de programación separadas para introducir cada campo, como en el siguiente ejemplo:

```
input name from employeeData  
input address from employeeData  
input payRate from employeeData
```

La mayoría de los lenguajes también le permiten escribir una sola declaración en el siguiente formato:

```
input name, address, payRate from employeeData
```



La mayoría de los lenguajes de programación proporcionan una forma para que usted use un nombre de grupo para los datos de registro, como en la siguiente declaración:

```
input EmployeeRecord from employeeData
```

Al usar este formato usted necesita definir los campos separados que componen un `EmployeeRecord` cuando declare las variables para el programa.

Por lo general usted no desea introducir varios elementos en una sola declaración cuando lee los datos desde un teclado, debido a que quiere indicar al usuario que introduzca cada elemento por separado conforme usted lo introduce. Sin embargo, cuando usted recupera los datos de un archivo, no se necesitan indicadores. En cambio, cada elemento se recupera en secuencia y se almacena en la memoria en la ubicación nombrada apropiada.



La forma en que un programa sabe cuántos datos introducir para cada variable difiere entre los lenguajes de programación. En muchos de ellos un delimitador, como una coma, se almacena entre los campos de datos. En otros, la cantidad recuperada depende de los tipos de datos de las variables en la declaración de entrada.

La figura 7-2 muestra cómo funciona una declaración de entrada. Cuando ésta se ejecuta cada campo se copia y coloca en la variable apropiada en la memoria de la computadora. Nada en el disco indica un nombre de campo asociado con cualquiera de los datos; los nombres de variable sólo existen dentro del programa. Por ejemplo, otro programa podría usar el mismo archivo como entrada y llamar a los campos `surname`, `street` y `salary`.

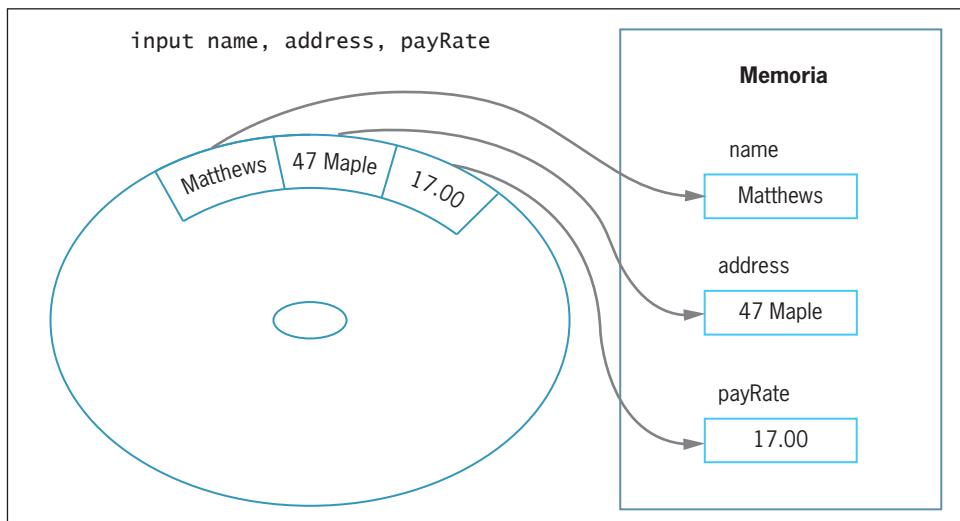


Figura 7-2 Lectura de tres elementos de datos desde un dispositivo de almacenamiento a la memoria



En algunos lenguajes usted debe especificar la longitud de cada campo que se lee desde un archivo de datos. En otros, el tipo de sus datos determina la longitud de cada campo.

Cuando usted lee datos desde un archivo, debe leer todos los campos que están almacenados aun cuando quizás no desee usarlos todos. Por ejemplo, suponga que quiere leer un archivo de datos de los empleados que contiene nombres, direcciones y tarifa de pago para cada uno, y desea dar salida a una lista de nombres. Aun cuando no se interese en los campos de dirección o tarifa de pago, debe leerlos en su programa para cada empleado antes que pueda obtener el nombre para el siguiente.

Escribir datos en un archivo

Cuando usted almacena los datos de un archivo de computadora en un dispositivo de almacenamiento persistente, **escribe en el archivo**. Esto significa que copia los datos de la RAM al archivo. Cuando escribe los datos a un archivo, escribe el contenido de los campos usando una declaración como la siguiente:

```
output name, address, payRate to employeeData
```

Cuando usted escribe los datos en un archivo, por lo general no incluye explicaciones que hagan que las personas los interpreten con mayor facilidad; sólo escribe hechos y cifras. Por ejemplo, no incluye encabezados de columna ni explicaciones como *La tarifa de pago es*, ni comas, signos de dólar o signos de porcentaje en los valores numéricos. Esos adornos son apropiados para dar salida en un monitor o en papel, pero no para el almacenamiento.

Cerrar un archivo

Cuando usted termina de usarlo, el programa deberá **cerrar el archivo**; un archivo cerrado ya no está disponible para su aplicación. No cerrar un archivo de entrada (del cual ha leído datos) por lo general no tiene consecuencias graves; los datos aún existen en el archivo. Sin embargo, si no cierra un archivo de salida (en el que escribe datos), los datos quizás no se guarden en forma correcta y podrían volverse inaccesibles. Siempre deberá cerrar todos los archivos que abra y debería hacerlo tan pronto como ya no los necesite. Cuando deja un archivo abierto sin razón usa recursos de la computadora y el desempeño de ésta lo resiente. Además, en particular dentro de una red, otro programa podría estar esperando para usarlo.



En la mayoría de los lenguajes de programación, si usted lee los datos desde un teclado o los escribe para desplegarlos en el monitor, no necesita abrir o cerrar el dispositivo. El teclado y el monitor son los **dispositivos de entrada y de salida predeterminados**, respectivamente.

Un programa que ejecuta operaciones de archivo

La figura 7-3 contiene un programa que abre dos archivos, lee los datos del empleado del archivo de entrada, altera la tarifa de pago, escribe el registro actualizado en un archivo de salida y los cierra. Las declaraciones que se usan en los archivos están sombreadas. La convención en este libro es colocar en los diagramas de flujo las declaraciones de abrir archivo y cerrar archivo en paralelogramos, debido a que son operaciones relacionadas con la entrada y la salida.

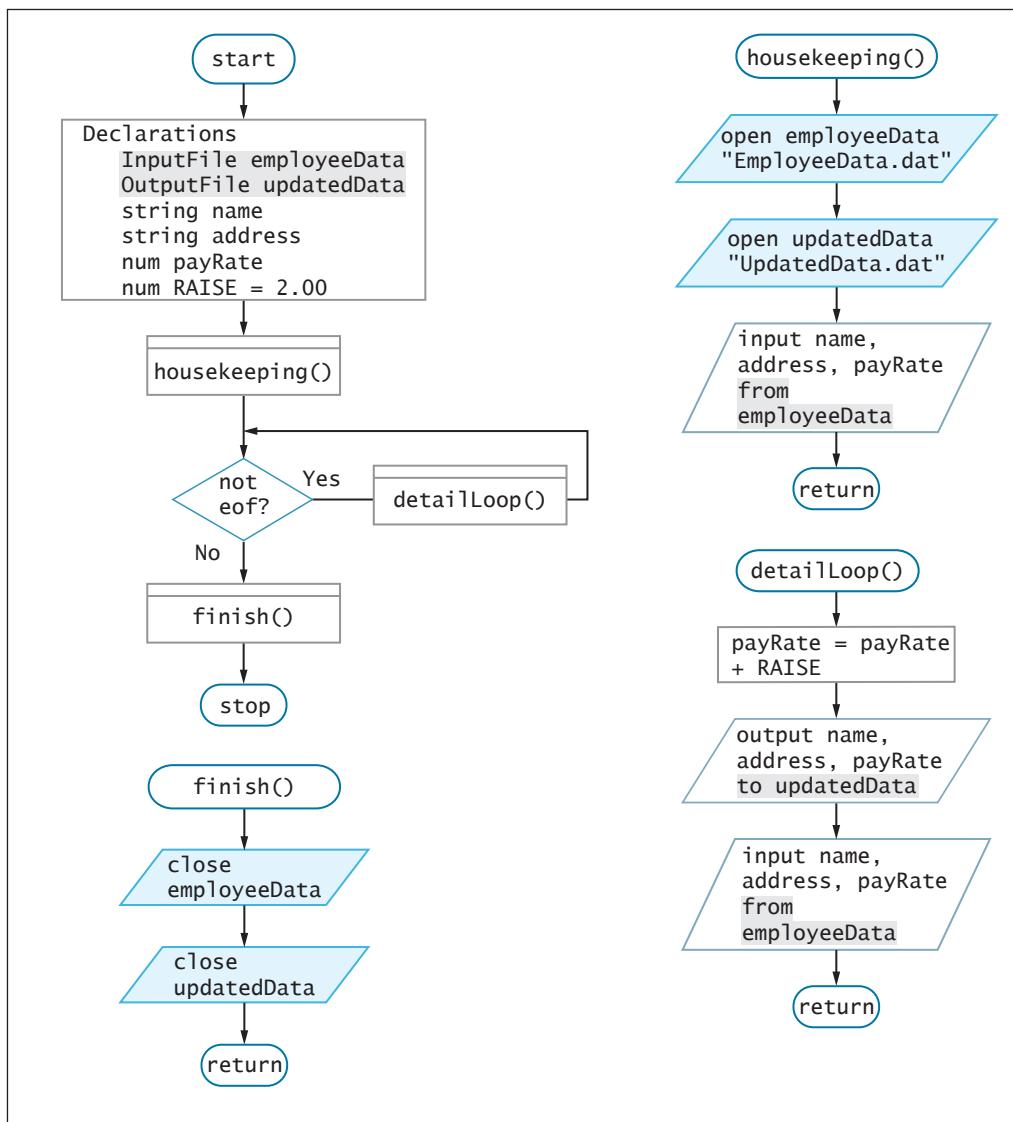


Figura 7-3 Diagrama de flujo y seudocódigo para un programa que usa archivos (continúa)

(continuación)

```
start
    Declarations
        InputFile employeeData
        OutputFile updatedData
        string name
        string address
        num payRate
        num RAISE = 2.00
    housekeeping()
    while not eof
        detailLoop()
    endwhile
    finish()
stop

housekeeping()
    open employeeData "EmployeeData.dat"
    open updatedData "UpdatedData.dat"
    input name, address, payRate from employeeData
return

detailLoop()
    payRate = payRate + RAISE
    output name, address, payRate to updatedData
    input name, address, payRate from employeeData
return

finish()
    close employeeData
    close updatedData
return
```

Figura 7-3 Diagrama de flujo y seudocódigo para un programa que usa archivos

En el programa de la figura 7-3, cada dato del empleado se lee en la memoria. Luego la variable `payRate` en la memoria se aumenta \$2.00. El valor de la tarifa de pago en el dispositivo de almacenamiento de entrada no se altera. Después de que se aumenta la tarifa, el nombre, la dirección y los valores de dicha tarifa recién alterados se almacenan en el archivo de salida. Cuando el procesamiento está completo, el archivo de entrada retiene los datos originales y el de salida contiene los datos revisados. Muchas organizaciones mantendrían el archivo original como respaldo. Un **archivo de respaldo** es una copia que se conserva en caso de que los valores necesiten restablecerse a su estado original. La copia de respaldo se llama **archivo padre** y la copia recién revisada es un **archivo hijo**.

 Lógicamente, los verbos *imprimir*, *escribir* y *desplegar* significan lo mismo: todos producen una salida. Sin embargo, en la conversación, los programadores por lo general reservan la palabra *imprimir* para situaciones en las que se refieren a *producir una salida en papel*. Es más probable que usen *escribir* cuando hablan de enviar registros hacia un archivo de datos y *desplegar* cuando los envían hacia un monitor. En algunos lenguajes de programación no hay diferencia en el verbo que se usa para la salida al margen del hardware; usted sólo asigna distintos dispositivos de salida (impresoras, monitores y unidades de disco) según sea necesario a los objetos que los representan, nombrados por el programador.

A lo largo de este libro se le ha alentado a pensar en la entrada básicamente como el mismo proceso, ya sea que provenga de un usuario que mecanografía en forma interactiva en un teclado o de un archivo almacenado en un disco u otros medios. El concepto sigue siendo válido para este capítulo, que expone las aplicaciones que en general usan datos de archivo almacenados. Aunque los datos que se usan en una aplicación podrían introducirse en un teclado durante la ejecución de un programa, en este capítulo se supone que los elementos de datos se han introducido, validado y clasificado antes en otra aplicación, y luego se han procesado como entradas desde los archivos para lograr los resultados.

Clasificar es el proceso de colocar los registros en orden de acuerdo con el valor en uno o varios campos específicos. Los archivos pueden clasificarse en forma manual o con ayuda de un programa antes de guardarse. En este capítulo, se supone que el proceso de clasificación de registros ya se ha realizado.

DOS VERDADES Y UNA MENTIRA

Ejecución de operaciones con archivos

1. Usted da a un archivo un nombre interno en un programa y luego lo asocia con el nombre del sistema operativo para el archivo.
2. Cuando usted lee desde un archivo, copia los valores de la memoria a un dispositivo de almacenamiento.
3. Si no cierra un archivo de entrada por lo general no habrá consecuencias graves; los datos todavía existen en el mismo.

La afirmación falsa es la número 2. Cuando lee desde un archivo, copia los valores desde un dispositivo de almacenamiento hacia la memoria. Cuando escribe en un archivo, copia los valores de la memoria a un dispositivo de almacenamiento.

Comprensión de los archivos secuenciales y la lógica de control de interrupciones

Un **archivo secuencial** es un archivo en el que los registros se almacenan uno detrás de otro en cierto orden. Con frecuencia, los registros en un archivo secuencial están organizados con base en el contenido de uno o más campos; algunos ejemplos son:

- Un archivo referente a los empleados almacenado en orden por número de identificación
- Un archivo de componentes para una compañía manufacturera almacenado en orden por número de componente
- Un archivo de clientes para un negocio almacenado en orden alfabético por apellido

Comprensión de la lógica de control de interrupciones

Un **control de interrupciones** es una desviación temporal en la lógica de un programa. En particular, los programadores usan un **programa de control de interrupciones** cuando un cambio en un valor inicia acciones o procesamiento especiales; por lo general estos se escriben con el objetivo de organizar la salida para los programas que manejan registros de datos organizados de manera lógica en grupos basados en el valor en un campo o campos. Conforme lee los registros examina el mismo campo en cada uno, y cuando encuentra alguno que contiene un valor diferente de los que lo precedieron, ejecuta una acción especial. Por ejemplo, podría generar un informe que liste en orden a todos los clientes de una compañía de acuerdo con su estado de residencia, con un conteo de los mismos después de la lista de cada estado. Véase la figura 7-4 que muestra un ejemplo de un **informe de control de interrupciones** que se interrumpe después de cada cambio de estado.

Clientes de la compañía por estado de residencia			
Nombre	Ciudad	Estado	
Albertson	Birmingham	Alabama	
Davis	Birmingham	Alabama	
Lawrence	Montgomery	Alabama	
		Cuenta para Alabama	3
Smith	Anchorage	Alaska	
Young	Anchorage	Alaska	
Davis	Fairbanks	Alaska	
Mitchell	Juneau	Alaska	
Zimmer	Juneau	Alaska	
		Cuenta para Alaska	5
Edwards	Phoenix	Arizona	
		Cuenta para Arizona	1

Figura 7-4 Informe de control de interrupciones con totales después de cada estado

Otros ejemplos de estos informes generados por los programas de control de interrupciones podrían ser:

- Todos los empleados listados en orden por número de departamento, con una página nueva empezada para cada departamento
- Todos los libros que hay a la venta en una librería listados en orden por categoría (como referencia o autoayuda), con un conteo después de cada categoría
- Todos los artículos vendidos ordenados de acuerdo con la fecha de venta, con tinta de un color diferente para cada mes

Cada uno de estos informes comparte dos rasgos:

- Los registros que se usan en cada informe se listan en orden de acuerdo con una variable específica: estado, departamento, categoría o fecha.
- Cuando esa variable cambia el programa emprende una acción especial: empieza una página nueva, imprime un conteo o total, o cambia el color de la tinta.

Para generar un informe de control de interrupciones, los registros que usted haya introducido deben estar organizados en orden secuencial con base en el campo que causará las interrupciones. En otras palabras, para escribir un programa que genere un informe de clientes por estado, como el de la figura 7-4, los registros deben estar agrupados por estado antes de comenzar a procesarlos. Con frecuencia este agrupamiento significaría colocar los registros en orden alfabético por estado, aunque con la misma facilidad podrían estar ordenados por población, nombre del gobernador o cualquier otro factor, mientras todos los registros de un estado estén juntos.



Con algunos lenguajes más recientes, como SQL, los detalles de los controles de interrupciones se manejan de manera automática. Aun así, entender cómo funcionan los programas de control de interrupciones mejora su competencia como programador.

Suponga que tiene un archivo de entrada que contiene los nombres de clientes, ciudades y estados, y desea generar un informe como el de la figura 7-4. La lógica básica del programa funciona así:

- Cada vez que usted lee el registro de un cliente del archivo de entrada determina si reside en el mismo estado que el cliente anterior.
- De ser así, simplemente da salida a los datos del cliente, agrega 1 a un contador y lee otro registro, sin procesamiento especial. Si hay 20 clientes en un estado, estos pasos se repiten 20 veces seguidas: leer sus datos, contarlos y darle salida.
- Al final usted leerá un registro para un cliente que no se encuentra en el mismo estado. En ese punto, antes de dar salida a los datos para el primer cliente en el nuevo estado debe dar salida a la cuenta para el estado previo. También debe reiniciar el contador en 0 de modo que esté listo para empezar a contar los clientes en el siguiente estado. Luego puede proceder a manejar los registros de los clientes para el nuevo estado y continuar así hasta la siguiente vez que encuentre un cliente de un estado diferente.

Este tipo de programa contiene un **control de interrupciones de nivel único**, una interrupción en la lógica del programa (en este caso, hacer una pausa o tomar una desviación para dar salida a un conteo) que se basa en el valor de una sola variable (en este caso, el estado). La técnica que usted debe usar para “recordar” el estado anterior de modo que pueda compararlo con cada estado nuevo es crear una variable especial, llamada **campo de control de interrupciones**, para contener el estado previo. Mientras lee cada registro nuevo, al comparar los valores del estado nuevo y el estado anterior determina cuándo es tiempo de dar salida al conteo para el estado previo.

La figura 7-5 muestra la lógica de línea principal y el módulo `getReady()` para un programa que genera el informe de la figura 7-4. En la lógica de línea principal, la variable de control de interrupciones `oldState` se encuentra en la declaración sombreada. En el módulo `getReady()` se da salida a los encabezados del informe, se abre el archivo y se lee el primer registro en la memoria. Luego, el valor del estado en el primer registro se copia a la variable `oldState` (véase el sombreado). Note que sería incorrecto inicializar `oldState` cuando se declara. En el momento en que usted declara las variables al comienzo del programa principal, todavía no ha leído el primer registro; por consiguiente, no sabe cuál será el valor del primer estado. Podría suponer que es *Alabama* debido a que es el primero en orden alfabético y quizás tenga razón, pero tal vez en este conjunto de datos el primer estado sea *Alaska* o incluso *Wyoming*. Estará seguro de almacenar el valor del primer estado correcto si lo copia del primer registro de entrada.

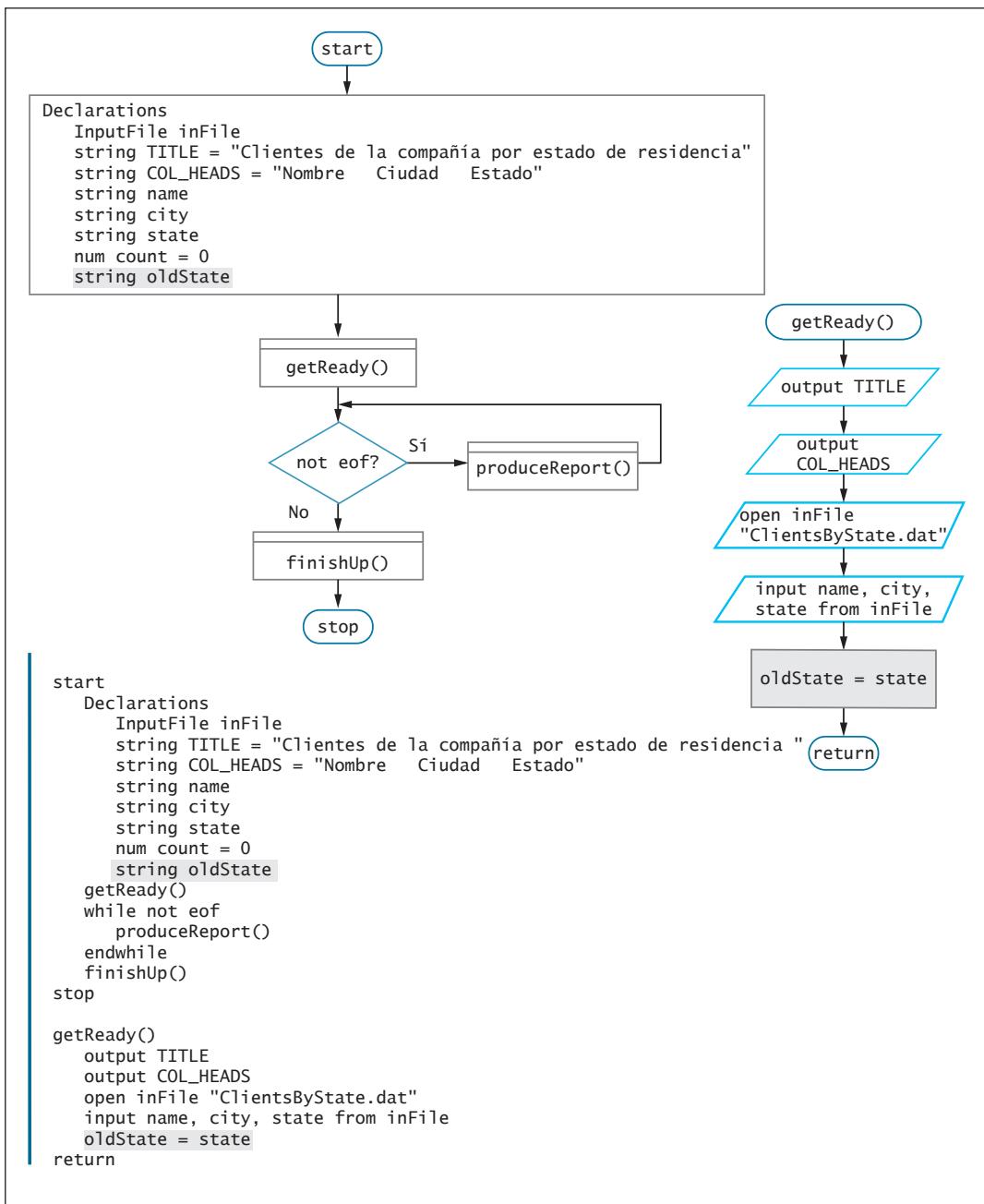


Figura 7-5 Lógica de línea principal y módulo `getReady()` para el programa que genera un informe de los clientes por estado

Dentro del módulo `produceReport()` en la figura 7-6, la primera tarea es verificar si `state` contiene el mismo valor que `oldState`. Para el primer registro, en la primera pasada por este método, los valores son iguales (debido a que usted los establece así justo después de obtener el primer registro de entrada en el módulo `getReady()`). Por consiguiente, usted da salida a los datos del primer cliente, agregando 1 a `count`, e introduciendo el siguiente registro.

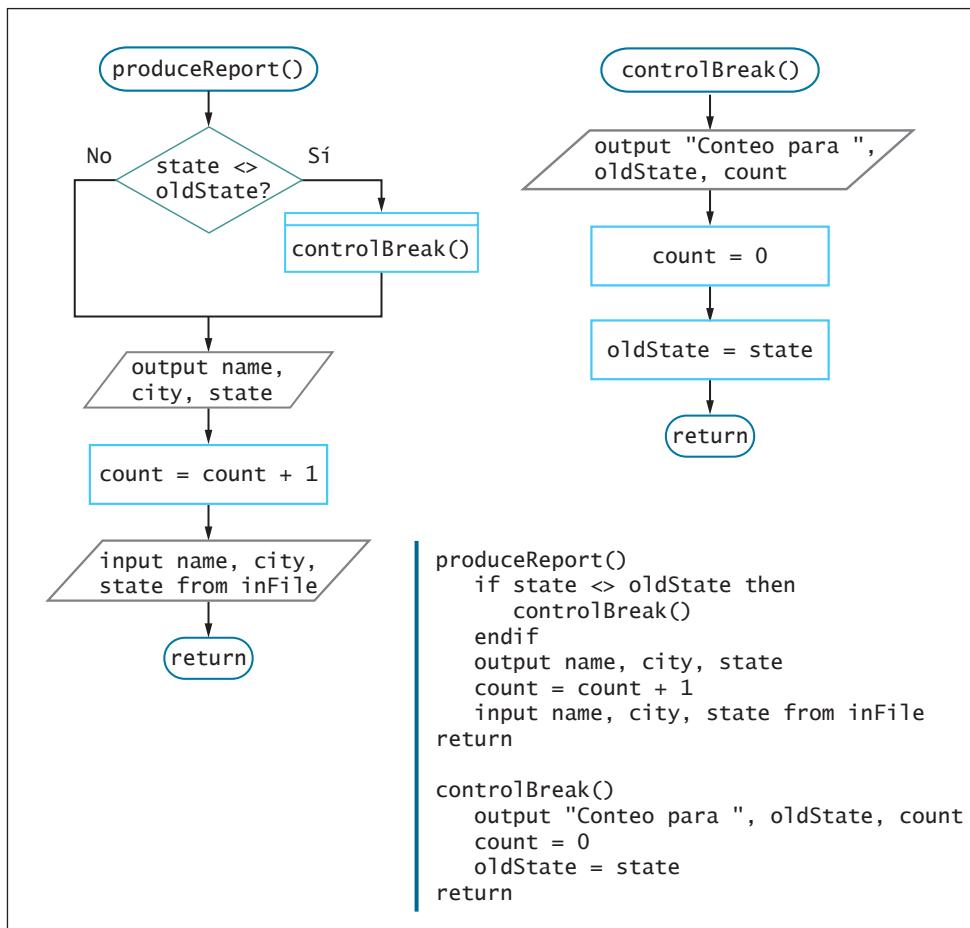


Figura 7-6 Los módulos `produceReport()` y `controlBreak()` para el programa que produce clientes por estado

En tanto cada registro nuevo contenga el mismo valor `state`, usted continuará dando salida, contando e introduciendo, sin hacer pausa nunca para dar salida al conteo. Al final, leerá en un cliente cuyo estado será diferente del anterior. Allí ocurre el control de interrupciones. Siempre que un estado nuevo difiere del antiguo deben realizarse tres tareas:

- Dar salida al conteo para el estado previo.
- Reiniciar el conteo en 0 de modo que comience a contar los registros para el estado nuevo.
- Actualizar el campo del control de interrupciones.

Cuando el módulo `produceReport()` recibe el registro de un cliente para el que `state` no es el mismo que `oldState`, usted causará una interrupción en el flujo normal del programa. El nuevo registro del cliente debe “esperar” mientras se da salida al conteo para el estado que acaba de terminar y `count` y el campo de control de interrupciones `oldState` adquieren valores nuevos.

El módulo `produceReport()` continúa dando salida a los nombres de los clientes, ciudades y estados hasta que se llega al final del archivo; entonces se ejecuta el módulo `finishUp()`. Como se muestra en la figura 7-7, el módulo que se ejecuta después de procesar el último registro en un programa de control de interrupciones debe completar cualquier procesamiento requerido para el último registro que se manejó. En este caso, el módulo `finishUp` debe desplegar el conteo para el último estado que se procesó. Después de que el archivo de entrada se cierra, la lógica puede regresar al programa principal, donde el programa termina.

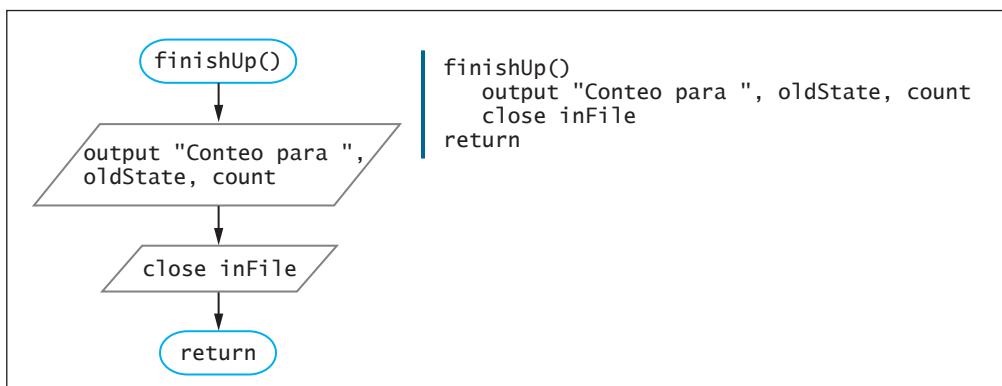


Figura 7-7 El módulo `finishUp()` para el programa que genera un informe de clientes por estado

DOS VERDADES Y UNA MENTIRA

Comprensión de los archivos secuenciales y la lógica del control de interrupciones

1. En un programa de control de interrupciones, un cambio en el valor de una variable inicia acciones o procesamiento especiales.
2. Cuando una variable de control de interrupciones cambia, el programa emprende una acción especial.
3. Para generar un informe de control de interrupciones, sus registros de entrada deben estar organizados en orden secuencial con base en el primer campo en el registro.

La afirmación falsa es la número 3. Sus registros de entrada deben estar organizados en orden secuencial con base en el campo que causará las interrupciones.

Unión de archivos secuenciales

En los negocios con frecuencia se necesitan que dos o más archivos secuenciales se unan. **Unir archivos** implica combinar dos o más mientras se mantiene el orden secuencial. Por ejemplo:

- Piense que tiene un archivo de los empleados actuales y otro de empleados recién contratados, ambos ordenados por número de identificación. Necesita unirlos en un archivo combinado antes de correr el programa de nómina de esta semana.
- Suponga que tiene un archivo de componentes manufacturados en la fábrica Northside y otro de componentes fabricados en Southside, ambos ordenados por número de componente. Necesita unir estos dos archivos en uno combinado, creando una lista maestra de componentes disponibles.
- Imagine que tiene un archivo que lista los clientes del último año y otro con los del presente año, ambos en orden alfabético. Usted desea crear una lista de correo de todos los clientes ordenados por apellido.

Antes de que pueda unir archivos con facilidad, deben cumplirse dos condiciones; cada archivo:

- debe contener el mismo diseño de registros.
- debe estar clasificado en el mismo orden con base en el mismo campo. El **orden ascendente** describe los registros de los valores ordenados de menor a mayor; el **orden descendente** los describe de mayor a menor.

Por ejemplo, suponga que su negocio tiene dos ubicaciones, una en la Costa Este y otra en la Costa Oeste, y cada una mantiene un archivo de clientes en orden alfabético por nombre. Cada archivo contiene campos para el nombre y el saldo del cliente. Usted puede llamar a los campos en el archivo de la Costa Este `eastName` y `eastBalance`, y a los de la Costa Oeste `westName` y `westBalance`. Desea unir ambos archivos para crear uno combinado que contenga los registros de todos los clientes. La figura 7-8 presenta algunos datos de muestra para los archivos; usted desea crear uno como el de la figura 7-9.

Archivo de la Costa Este		Archivo de la Costa Oeste	
<code>eastName</code>	<code>eastBalance</code>	<code>westName</code>	<code>westBalance</code>
Able	100.00	Chen	200.00
Brown	50.00	Edgar	125.00
Dougherty	25.00	Fell	75.00
Hanson	300.00	Grand	100.00
Ingram	400.00		
Johnson	30.00		

Figura 7-8 Datos de muestra contenidos en dos archivos de clientes

mergedName	mergedBalance
Able	100.00
Brown	50.00
Chen	200.00
Dougherty	25.00
Edgar	125.00
Fell	75.00
Grand	100.00
Hanson	300.00
Ingram	400.00
Johnson	30.00

Figura 7-9 Archivo de clientes combinado

La lógica de línea principal para un programa que une dos archivos es similar a la que ha usado antes en otros programas: contiene tareas administrativas preliminares; un módulo detallado que se repite hasta el final del programa, y algunas tareas de limpieza de fin del trabajo.

Sin embargo, la mayoría de los programas que ha estudiado procesaban los registros hasta que se cumplía una condición eof, ya sea porque un archivo de datos de entrada alcanzaba su final o porque un usuario introducía un valor centinela en un programa interactivo. En un programa que une dos archivos de entrada, verificar el eof sólo en uno de ellos es insuficiente. En cambio, el programa puede comprobar una variable bandera con un nombre como `areBothAtEnd`. Por ejemplo, usted podría inicializar una variable de cadena `areBothAtEnd` en "N", pero cambiar su valor a "Y" después de que ha encontrado eof en ambos archivos de entrada. (Si el lenguaje que usa soporta un tipo de datos booleano, puede usar los valores `true` y `false` en lugar de cadenas.)

La figura 7-10 muestra la lógica de línea principal para un programa que une los archivos que se muestran en la figura 7-8. Después de que se ejecuta el módulo `getReady()`, la pregunta sombreada que envía la lógica al módulo `finishUp()` prueba la variable `areBothAtEnd`. Cuando contiene "Y", el programa termina.

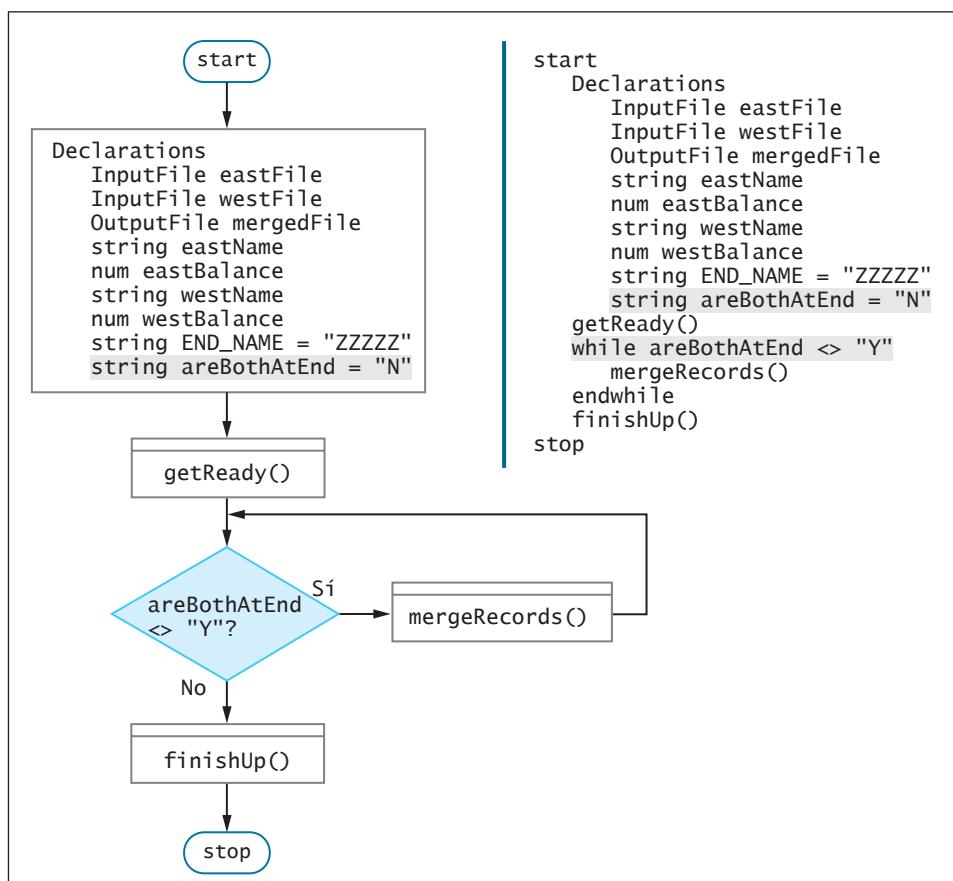


Figura 7-10 Lógica de línea principal de un programa que une archivos

El módulo `getReady()` se muestra en la figura 7-11. Abre tres archivos: los de entrada para los clientes del este y del oeste, y uno de salida en el que se colocarán los registros unidos. El programa lee entonces un registro de cada archivo de entrada. Si cualquier archivo ha llegado a su fin, la constante `END_NAME` se asigna a la variable que contiene el nombre del cliente del

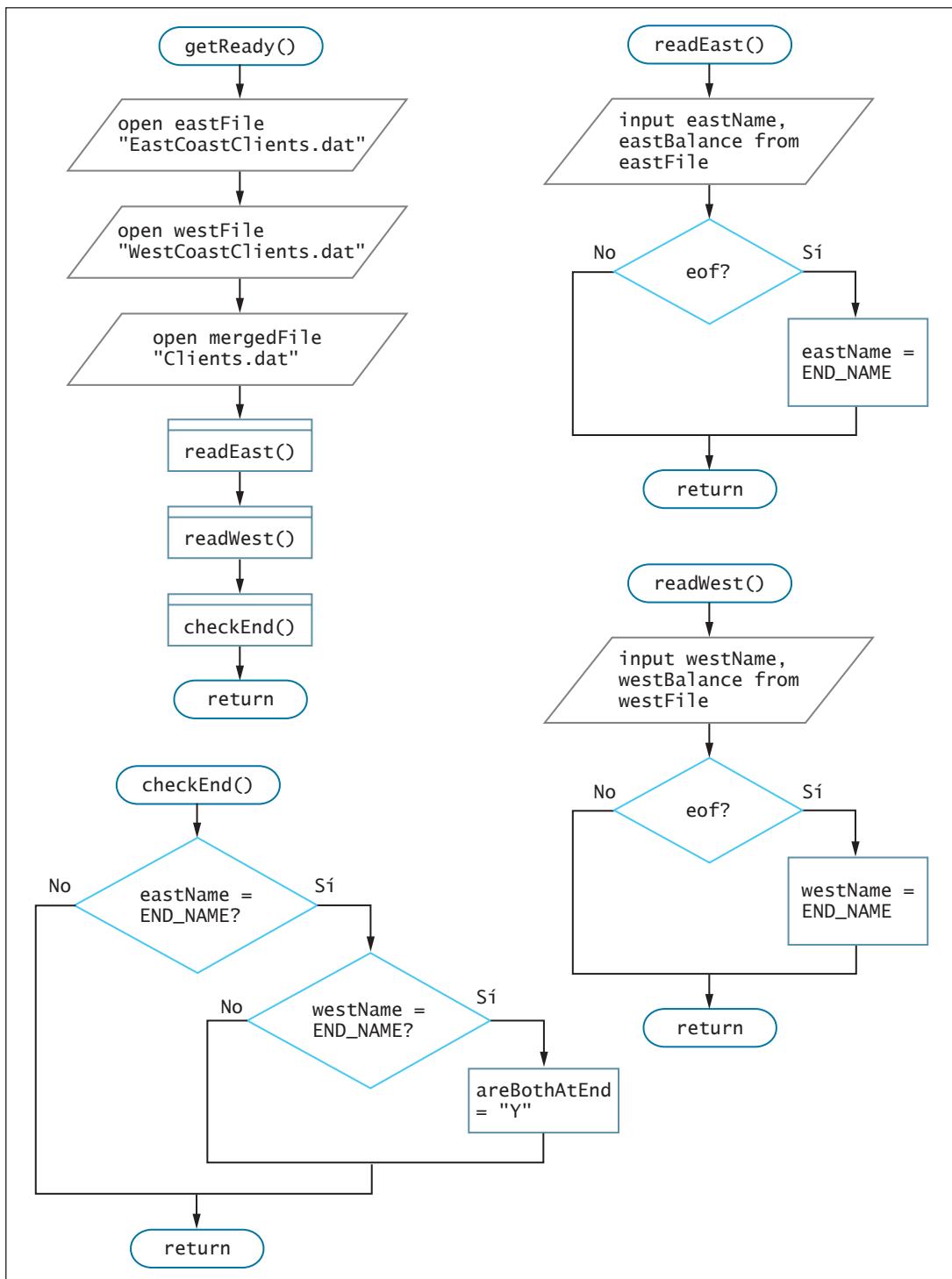


Figura 7-11 El módulo `getReady()` para un programa que une archivos, y los métodos a los que llama (continúa)

(continuación)

```
getReady()
    open eastFile "EastCoastClients.dat"
    open westFile "WestCoastClients.dat"
    open mergedFile "Clients.dat"
    readEast()
    readWest()
    checkEnd()
    return

    readEast()
        input eastName, eastBalance from eastFile
        if eof then
            eastName = END_NAME
        endif
    return

    readWest()
        input westName, westBalance from westFile
        if eof then
            westName = END_NAME
        endif
    return

    checkEnd()
        if eastName = END_NAME then
            if westName = END_NAME then
                areBothAtEnd = "Y"
            endif
        endif
    return
```

Figura 7-11 El módulo `getReady()` para un programa que une archivos, y los métodos a los que llama

archivo. El módulo `getReady()` comprueba entonces si ambos archivos han terminado (la verdad es que es una ocurrencia rara en la porción `getReady()` de la ejecución del programa) y establece la variable bandera `areBothAtEnd` en “Y” si es el caso. Suponiendo que al menos un registro está disponible, la lógica entonces introduce el módulo `mergeRecords()`.

Cuando usted comienza el módulo `mergeRecords()` en el programa usando los archivos que se muestran en la figura 7-8, en la memoria de la computadora se colocan dos registros, uno de `eastFile` y otro de `westFile`. Uno de ellos debe escribirse primero en el nuevo archivo de salida. ¿Cuál? Debido a que los dos archivos de entrada contienen registros almacenados en orden alfabético y usted desea que el nuevo archivo los almacene también en orden alfabético, primero da salida al registro de entrada que tenga el valor alfabético más bajo en el campo de nombre. Por consiguiente, el proceso comienza como se presenta en la figura 7-12.

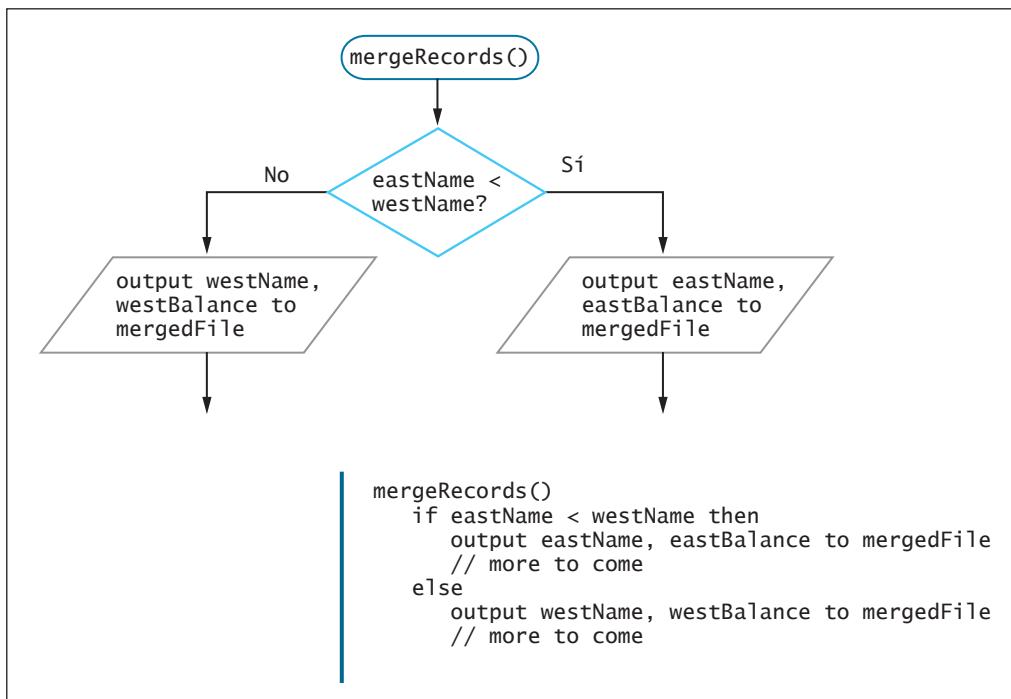


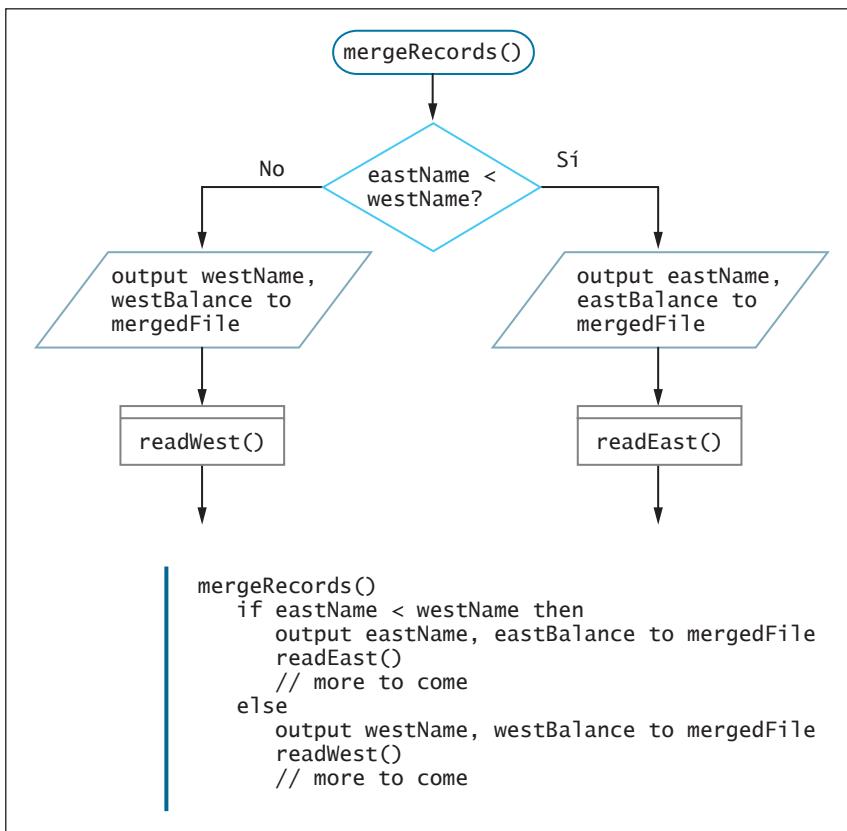
Figura 7-12 Comienzo del proceso de unión

Usando los datos de muestra de la figura 7-8, usted puede ver que el registro *Able* del archivo de la Costa Este debería escribirse en el archivo de salida, mientras el registro *Chen* del archivo de la Costa Oeste espera en la memoria. El valor *Able* de `eastName` es alfabéticamente menor que el valor *Chen* de `westName`.

Después de que usted escribe el registro de Able, ¿debería escribir a continuación el de Chen en el archivo de salida? No necesariamente. Esto depende del siguiente `eastName` en el registro de Able en `eastFile`. Cuando los registros de datos se leen en la memoria desde un archivo, un programa por lo común no “mira hacia adelante” para determinar los valores almacenados en el siguiente registro; en cambio, por lo general lee el registro en la memoria antes de tomar decisiones sobre su contenido. En este programa, usted necesita leer el siguiente registro de `eastFile` en la memoria y compararlo con *Chen*. Debido a que en este caso el siguiente registro en `eastFile` contiene el nombre *Brown*, otro registro de `eastFile` se escribe; ningún archivo de `westFile` se escribe todavía.

Después de los primeros dos registros de `eastFile`, ¿es el turno de escribir Chen? En realidad usted no lo sabe hasta que lea otro registro de `eastFile` y compare su valor de nombre con *Chen*. Debido a que este registro contiene el nombre *Dougherty*, en efecto es tiempo de escribir el registro de Chen. Después de que lo hace, ¿debería escribir el de *Dougherty*? Hasta que lea el siguiente registro de `westFile`, no sabe si ese debería colocarse antes o después del de *Dougherty*.

Por consiguiente, el método de unión procede así: compara dos registros, escribe el que tiene el menor nombre alfabético y lee otro del *mismo* archivo de entrada. Véase la figura 7-13.

**Figura 7-13** Continuación del proceso de unión

Recuerde los nombres de los dos archivos originales en la figura 7-8 y recorra los pasos del procesamiento.

1. Compare *Able* y *Chen*. Escriba el registro de Able. Lea el registro de Brown de `eastFile`.
2. Compare *Brown* y *Chen*. Escriba el registro de Brown. Lea el registro de Dougherty de `eastFile`.
3. Compare *Dougherty* y *Chen*. Escriba el registro de Chen. Lea el registro de Edgar de `westFile`.
4. Compare *Dougherty* y *Edgar*. Escriba el registro de Dougherty. Lea el registro de Hanson de `eastFile`.
5. Compare *Hanson* y *Edgar*. Escriba el registro de Edgar. Lea el registro de Fell de `westFile`.
6. Compare *Hanson* y *Fell*. Escriba el registro de Fell. Lea el registro de Grand de `westFile`.
7. Compare *Hanson* y *Grand*. Escriba el registro de Grand. Lea de `westFile` y encuentre `eof`. Esto hace que `westName` se establezca como `END_NAME`.

¿Qué sucede cuando usted llega al final del archivo de la Costa Oeste? ¿Se termina el programa? Esto no debería ser porque los registros de Hanson, Ingram y Johnson deben ser incluidos en el nuevo archivo de salida y ninguno se ha escrito todavía. Debido a que el campo

`westName` se establece en `END_NAME`, y `END_NAME` tiene un valor alfabético muy alto (`ZZZZZ`), cada `eastName` subsiguiente será menor que el valor de `westName`, y se procesará el resto del archivo `eastName`. Con un conjunto de datos diferente, `eastFile` podría haber terminado primero. En ese caso, `eastName` se establecería en `END_NAME` y se procesaría cada registro subsiguiente de `westFile`.

La figura 7-14 muestra los módulos `mergeRecords()` y `finishUp()` completos.

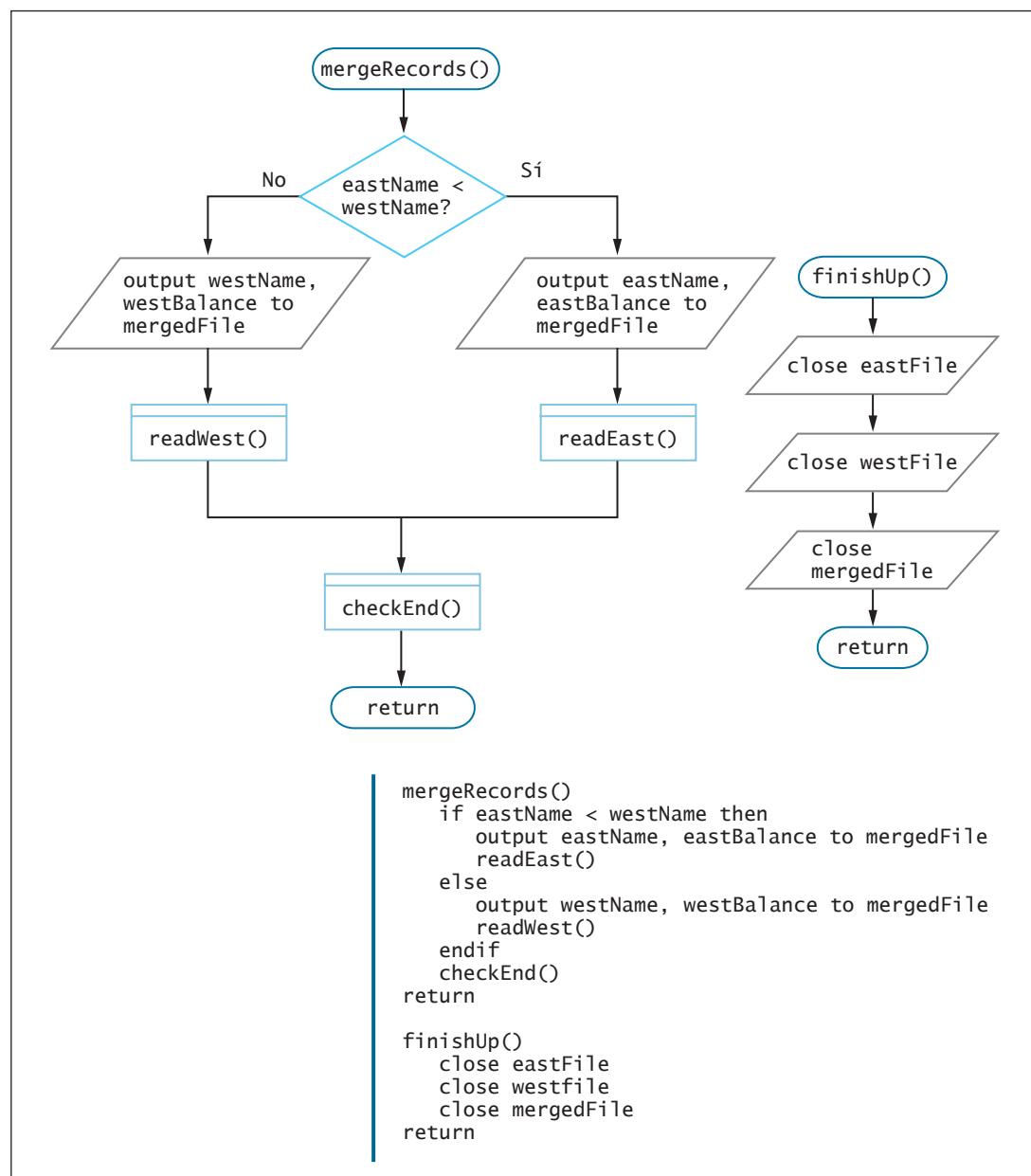


Figura 7-14 Los módulos `mergeRecords()` y `finishUp()` para el programa de unión de archivos



Como valor para END_NAME, usted quizá elija usar 10 o 20 Z en lugar de sólo cinco. Aunque es improbable que una persona tenga el apellido ZZZZZ, debería asegurarse de que el valor alto que escoja es en realidad más alto que cualquier valor legítimo.

Después de que se procesa el registro de Grand, se lee `westFile` y se encuentra `eof`, así que `westName` se establece en `END_NAME`. Ahora, cuando usted entra de nuevo al ciclo, se comparan `eastName` y `westName`, y `eastName` todavía tiene *Hanson*. El valor de `eastName` (*Hanson*) es menor que el valor de `westName` (ZZZZZ), así que se escriben los datos para el registro de `eastName` al archivo de salida, y se lee otro registro de `eastFile` (*Ingram*).

La corrida completa del programa de unión de archivos entonces ejecuta los primeros seis de los siete pasos que se listaron antes, y luego procede como se muestra en la figura 7-14 y como sigue, empezando con un paso 7 modificado:

7. Compare *Hanson* y *Grand*. Escriba el registro de *Grand*. Lea de `westFile`, encuentra `eof` y establece `westName` en ZZZZZ.
8. Compare *Hanson* y ZZZZZ. Escriba el registro de *Hanson*. Lea el registro de *Ingram*.
9. Compare *Ingram* y ZZZZZ. Escriba el registro de *Ingram*. Lea el registro de *Johnson*.
10. Compare *Johnson* y ZZZZZ. Escriba el registro de *Johnson*. Lea de `eastFile`, encuen-tra `eof` y establece `eastName` en ZZZZZ.
11. Ahora que ambos nombres son ZZZZZ, establezca la bandera `areBothAtEnd` en “Y”.

Cuando la variable bandera `areBothAtEnd` es igual a “Y”, el ciclo termina, los archivos se ciertran y el programa termina.



Si dos nombres son iguales durante el proceso de unión, por ejemplo, cuando hay un registro *Hanson* en cada archivo, entonces ambos *Hanson* se incluirán en el archivo final. Cuando `eastName` y `westName` concuerdan, `eastName` no es menor que `westName`, así que usted escribe el registro *Hanson* de `westFile`. Después de leer el siguiente registro de `westFile`, `eastName` será menor que el siguiente `westName` y se dará salida al registro *Hanson* de `eastFile`. Un programa de unión más complicado podría verificar otro campo, como el nombre, cuando los valores del apellido concuerden.

Usted puede unir cualquier cantidad de archivos. Para unir más de dos, la lógica sólo es un poco más complicada; debe comparar los campos clave de todos los archivos antes de decidir cuál es el siguiente candidato para salida.

DOS VERDADES Y UNA MENTIRA

Unión de archivos secuenciales

1. Un archivo secuencial es uno en el que se almacenan los registros uno detrás de otro en algún orden; con mayor frecuencia, se almacenan con base en el contenido de uno o más campos dentro de cada registro.
2. Unir archivos implica combinar dos o más de ellos mientras se mantiene el orden secuencial.
3. Antes de que pueda unir archivos con facilidad, cada archivo debe contener el mismo número de registros.

La afirmación falsa es la número 3. Antes de que pueda unir archivos con facilidad, cada uno debe contener el mismo diseño de registros y cada uno de los que se usan en la unión debe estar clasificado en el mismo orden con base en el mismo campo.

Procesamiento de archivos maestros y de transacción

En la última sección usted aprendió cómo unir archivos secuenciales relacionados en los que cada registro en cada archivo contenía los mismos campos. Algunos archivos secuenciales relacionados, sin embargo, no contienen los mismos campos; en cambio, algunos tienen una relación maestro-transacción. Un **archivo maestro** contiene datos completos y relativamente permanentes; un **archivo de transacción** contiene datos más temporales. Por ejemplo, un archivo maestro de clientes podría contener los nombres, direcciones y números telefónicos de los clientes, y uno de transacción quizás contenga los datos que describen la compra más reciente de cada cliente.

Por lo común, usted recopila transacciones por un periodo, las almacena en un archivo y luego usa una por una para actualizar los registros concordantes en un archivo maestro. Usted **actualiza el archivo maestro** haciendo cambios apropiados a los valores en sus campos con base en las transacciones recientes. Por ejemplo, un archivo que contenga los datos de las transacciones de compra para un cliente podría usarse para actualizar cada campo de saldo deudor en un archivo maestro de registros del cliente.

Aquí hay algunos ejemplos de archivos que tienen una relación maestro-transacción:

- Una biblioteca mantiene un archivo maestro de todos los usuarios y uno de transacción con información sobre cada libro u otros artículos que se han sacado.
- Un colegio mantiene un archivo maestro de todos los estudiantes y uno de transacción para el registro de cada curso.

- Una compañía telefónica mantiene un archivo maestro para cada línea (número) y uno de transacción con información sobre cada llamada.

Cuando usted actualiza un archivo maestro puede adoptar dos enfoques:

- Cambiar en realidad la información en el archivo maestro. Cuando usa este enfoque, se pierde la información que existía en el archivo maestro antes del procesamiento de la transacción.
- Crear una copia del archivo maestro, haciendo los cambios en la nueva versión. Luego, puede almacenar la versión padre previa del archivo maestro por un periodo, en caso de que haya preguntas o discrepancias respecto al proceso de actualización. La versión hija actualizada se vuelve el nuevo archivo maestro que se usa en el procesamiento subsiguiente. Este enfoque se aplica en un programa más adelante en este capítulo.



Cuando se actualiza un archivo hijo se convierte en un padre, y su padre se vuelve un abuelo. Las organizaciones individuales crean políticas concernientes al número de generaciones de archivos de respaldo que guardarán antes de desecharlos. Los términos *padre* e *hijo* se refieren a las generaciones de respaldo de archivos, pero se usan para un propósito diferente en la programación orientada hacia los objetos. Cuando basa una clase en otra usando la herencia, la clase original es el padre y la derivada es el hijo.

La lógica que usted usa para ejecutar una correspondencia entre los registros del archivo maestro y del de transacción es similar a la que utiliza para realizar una unión. Igual que con una unión, debe comenzar con ambos archivos clasificados en el mismo orden y en el mismo campo. La figura 7-15 muestra la lógica de línea principal para un programa que relaciona archivos. El archivo maestro contiene el número de cliente, su nombre y un campo que contiene la cantidad total en dólares de todas las compras que ha hecho previamente. El archivo de transacción contiene datos para las ventas, incluyendo un número de transacción, el número del cliente que la realizó y la cantidad de la misma.

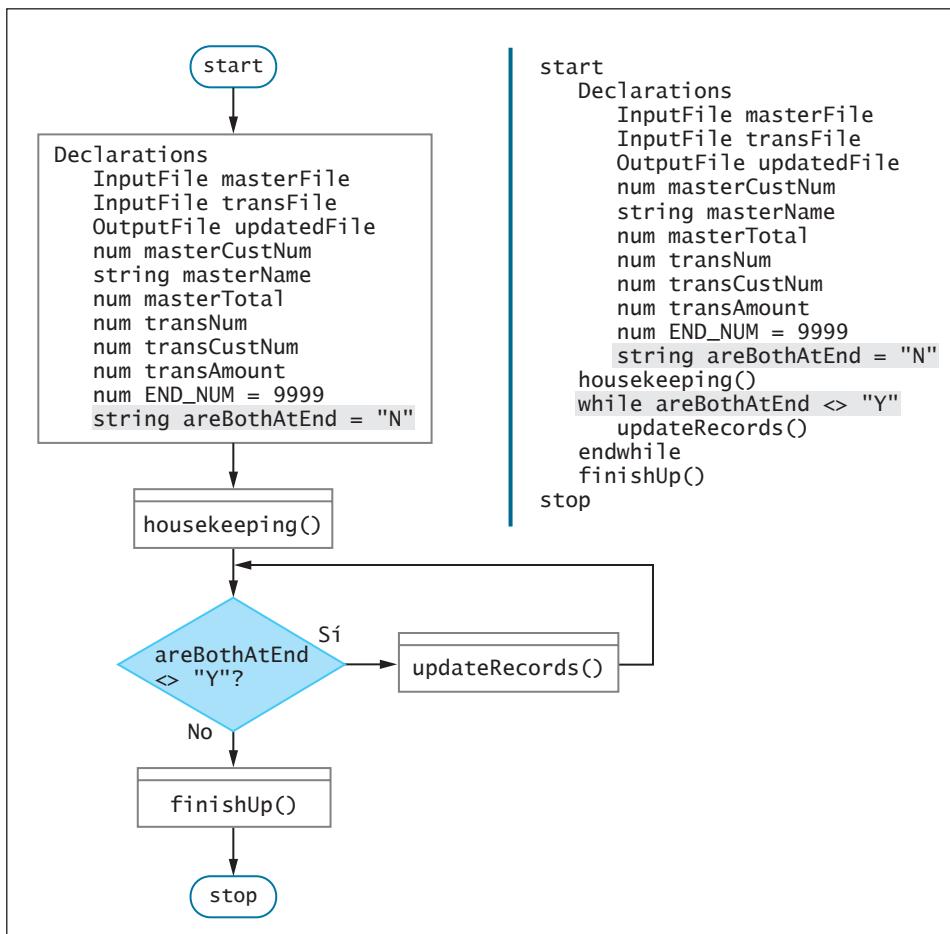


Figura 7-15 La lógica de línea principal para el programa maestro-transacción

La figura 7-16 contiene el módulo `housekeeping()` para el programa y los módulos a los que llama. Estos últimos son muy similares a sus contrapartes en el programa de unión de archivos que se estudió antes en el capítulo. Cuando el programa comienza, se lee un registro de cada archivo; cuando cualquier archivo termina, el campo que se usó para relacionar se establece en un valor alto, 9999, y cuando ambos archivos terminan se establece una variable bandera de modo que la lógica de línea principal pueda probar para el final del procesamiento. En el programa de unión de archivos que se presentó antes en este capítulo, usted colocó la cadena "ZZZZZ" en el campo del nombre del cliente al final del archivo porque se comparaban campos de cadena. En este ejemplo, debido a que usa campos numéricos (números de clientes), puede almacenar 9999 en ellos al final del archivo. La suposición es que 9999 es más alto que cualquier número de cliente válido.

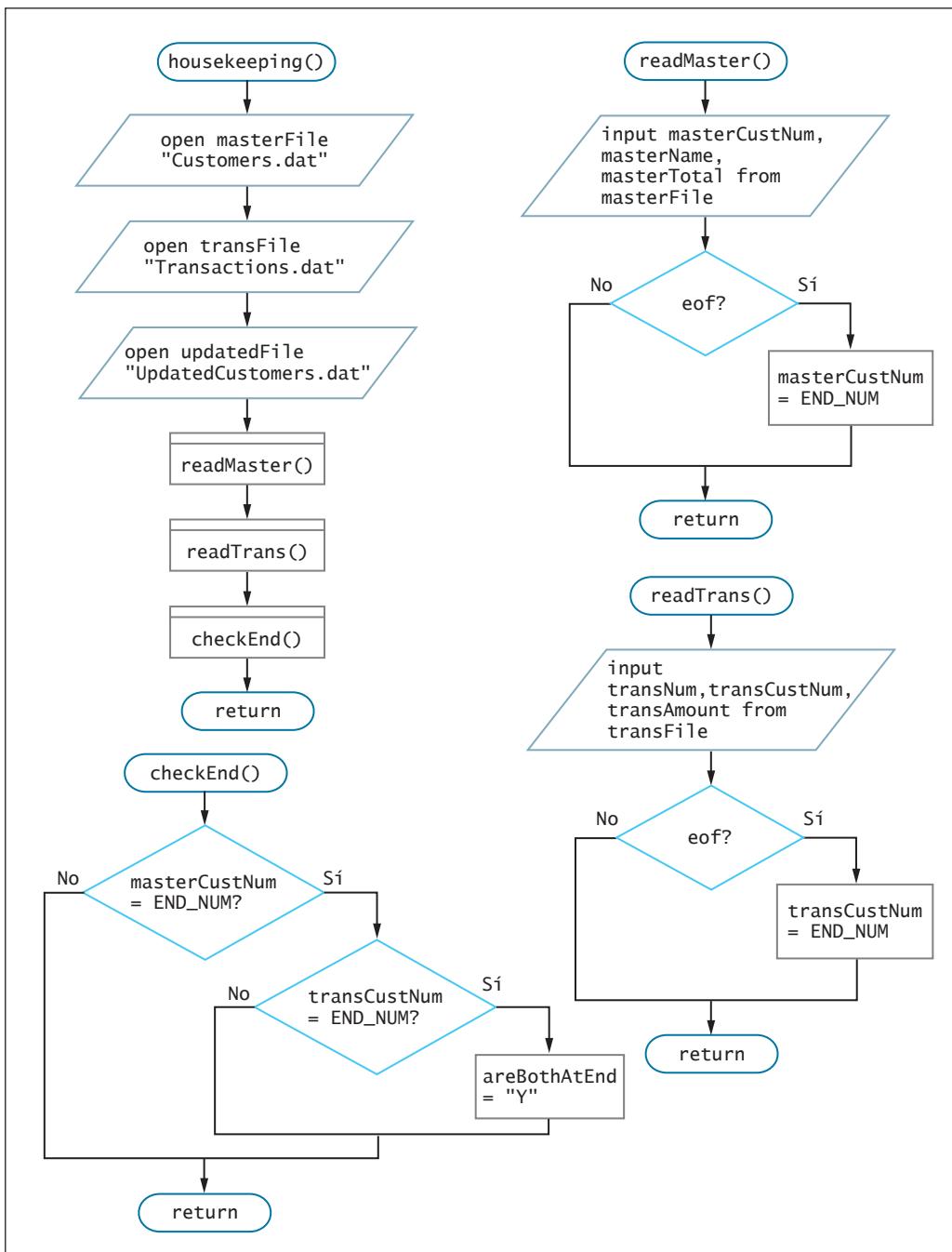


Figura 7-16 El módulo `housekeeping()` para el programa maestro-transacción y los módulos a los que llama (continúa)

(continuación)

```
housekeeping()
    open masterFile "Customers.dat"
    open transFile "Transactions.dat"
    open updatedFile "UpdatedCustomers.dat"
    readMaster()
    readTrans()
    checkEnd()
    return

    readMaster()
        input masterCustNum, masterName, masterTotal from masterFile
        if not eof then
            masterCustNum = END_NUM
        endif
    return

    readTrans()
        input transNum, transCustNum, transAmount from transFile
        if not eof then
            transCustNum = END_NUM
        endif
    return

    checkEnd()
        if masterCustNum = END_NUM then
            if transCustNum = END_NUM then
                areBothAtEnd = "Y"
            endif
        endif
    return
```

Figura 7-16 El módulo housekeeping() para el programa maestro-transacción y los módulos a los que llama

Imagine que actualizará los registros del archivo maestro que tiene a la mano en lugar de usar un programa de computadora y que todos los registros maestro y de transacción están almacenados en una hoja de papel separada. La forma más fácil de lograr la actualización es clasificar todos los registros maestros por número de cliente y colocarlos en un montón, y luego clasificar todas las transacciones por número de cliente (no por número de transacción) y colocarlas en otro montón. Entonces usted examinaría la primera transacción y buscaría los registros maestros hasta que encontrara una correspondencia. Cualesquier registros maestros sin transacciones se colocarían en un montón “completado” sin cambios. Cuando una transacción correspondiera con un registro maestro, usted corregiría este registro usando la nueva cantidad de transacción y luego pasaría a la siguiente transacción. Por supuesto, si no hubiera un registro maestro correspondiente para una transacción, entonces se daría cuenta de que ha ocurrido un error y quizás haría la transacción a un lado antes de continuar. El módulo updateRecords() funciona exactamente en la misma forma.

En el programa de unión de archivos que se presentó antes en este capítulo, su primera acción en el ciclo detallado del programa fue determinar cuál archivo contenía el registro con el valor menor; luego usted escribió ese registro. En un programa de correspondencia,

usted intenta determinar no sólo si el campo de comparación de un archivo es más grande que el de otro; también es importante saber si son *iguales*. En este ejemplo desea actualizar el campo `masterTotal` de los registros del archivo maestro sólo si el campo `transCustNum` del registro de transacción contiene una correspondencia exacta para el número de cliente en el archivo maestro. Por consiguiente, usted compara `masterCustNum` del archivo maestro y `transCustNum` del de transacción. Existen tres posibilidades:

- El valor `transCustNum` es igual a `masterCustNum`. En este caso, usted agrega `transAmount` a `masterTotal` y luego escribe el registro maestro actualizado en el archivo de salida. Luego, lee en un registro maestro y uno de transacción nuevos.
- El valor `transCustNum` es más alto que `masterCustNum`. Esto significa que no se registró una venta para ese cliente. Es correcto; no todos los clientes hacen una transacción en cada periodo, así que usted simplemente escribe el registro original del cliente con la misma información que contenía cuando se introdujo. Después obtiene el registro del siguiente cliente para ver si hizo la transacción que se considera actualmente.
- El valor `transCustNum` es menor que `masterCustNum`. Esto significa que usted intenta aplicar una transacción para la que no existe un registro maestro, de modo que debe haber un error ya que una transacción siempre debe tener un registro maestro. Puede manejar este error en diversas formas; aquí escribirá un mensaje de error en un dispositivo de salida antes de leer el siguiente registro de transacción. Un operador humano puede leer entonces el mensaje y efectuar la acción apropiada.



La lógica que se usa aquí supone que hay sólo una transacción por cliente. En los ejercicios al final de este capítulo usted elaborará la lógica para un programa en el que el cliente puede tener múltiples transacciones.

Ya sea que `transCustNum` fuera más alto, más bajo o igual a `masterCustNum`, después de leer la siguiente transacción o registro maestro (o ambos), verificará si `masterCustNum` y `transCustNum` se han establecido en 9999. Cuando ambos son 9999 usted establece la bandera `areBothAtEnd` en "Y".

La figura 7-17 ilustra el módulo `updateRecords()` que lleva a cabo la lógica del proceso de relación de archivos. La figura 7-18 presenta algunos datos de muestra que puede usar para recorrer la lógica para este programa.

Procesamiento de archivos maestros y de transacción

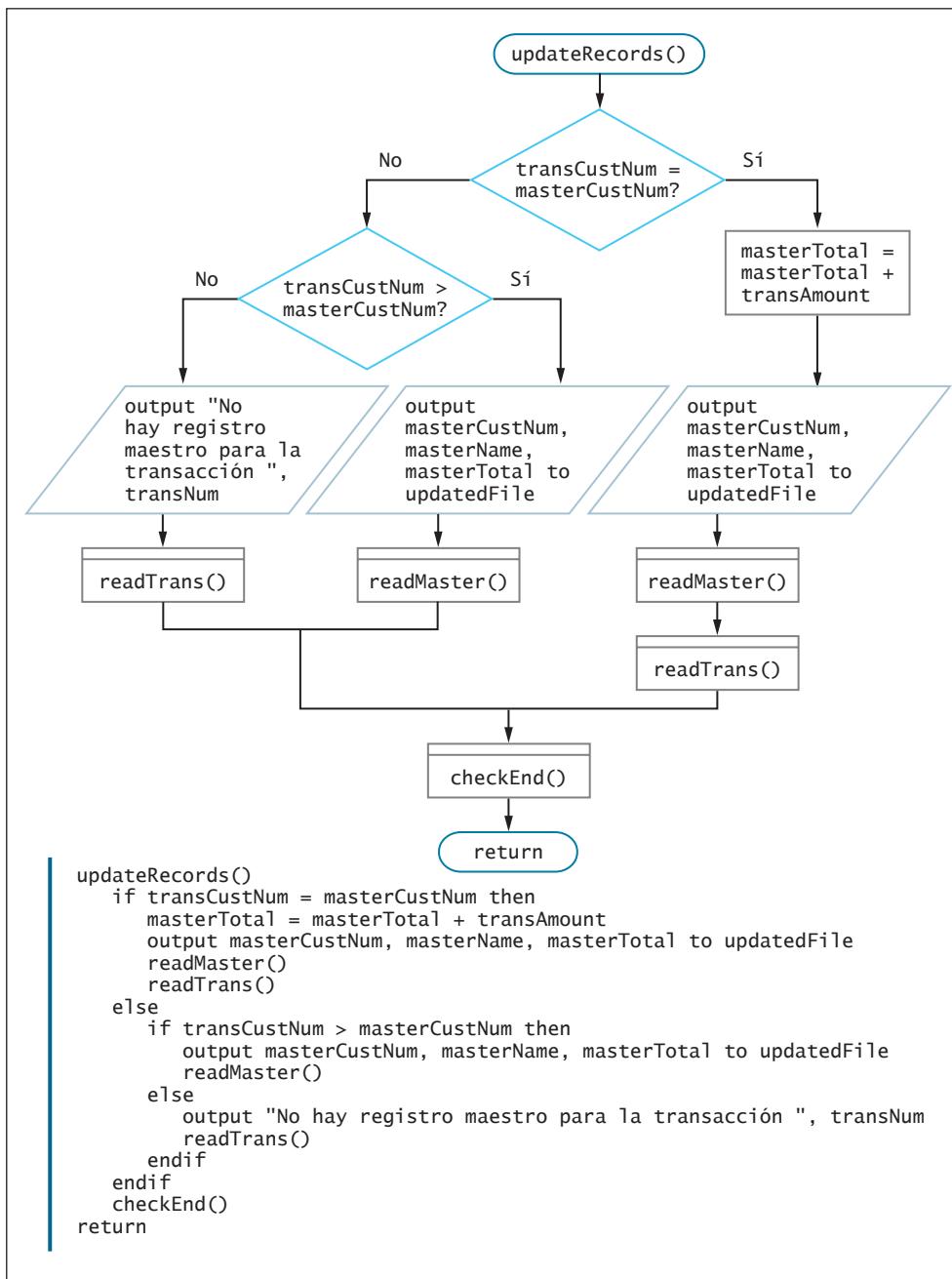


Figura 7-17 El módulo updateRecords() para el programa maestro-transacción

Archivo maestro		Archivo de transacción	
masterCustNum	masterTotal	transCustNum	transAmount
100	1000.00	100	400.00
102	50.00	105	700.00
103	500.00	108	100.00
105	75.00	110	400.00
106	5000.00		
109	4000.00		
110	500.00		

Figura 7-18 Datos de muestra para el programa que relaciona archivos

El programa procede así:

1. Lea el cliente 100 del archivo maestro y el cliente 100 del archivo de transacción. Los números de cliente son iguales, así que 400.00 del archivo de transacción se agregan a 1000.00 en el maestro y se escribe un registro en el archivo maestro nuevo con una cifra de ventas total de 1400.00. Después lea un nuevo registro desde cada archivo de entrada.
2. El número de cliente en el archivo maestro es 102 y en el archivo de transacción es 105, así que hoy no hay transacción para el cliente 102. Escriba el registro maestro exactamente en la forma en que entró ylea un registro maestro nuevo.
3. Ahora, el número del cliente maestro es 103 y el número de cliente de la transacción todavía es 105. Esto significa que el cliente 103 no tiene transacciones, así que usted escribe el registro maestro tal como está y lee uno nuevo.
4. Ahora, el número del cliente maestro es 105 y el número de transacción es 105. Debido a que el cliente 105 tenía un saldo de 75.00 y ahora tiene una transacción de 700.00, el nuevo total de ventas para el archivo maestro es 775.00 y se escribe un nuevo registro maestro. Lea un registro de cada archivo.
5. Ahora, el número maestro es 106 y el número de transacción es 108. Escriba el registro del cliente 106 tal como está ylea otro maestro.
6. Ahora, el número maestro es 109 y el número de transacción es 108. Ha ocurrido un error; el registro de transacción indica que hizo una venta al cliente 108, pero no hay registro maestro para el cliente número 108. La transacción es incorrecta (hay un error en el número de cliente) o es correcta pero no ha podido crear un registro maestro. De cualquier manera, se escribe un mensaje de error de modo que se notifique a algún empleado para que pueda manejar el problema. Luego, obtenga un nuevo registro de transacción.
7. Ahora, el número maestro es 109 y el número de transacción es 110. Escriba el registro maestro 109 sin cambios ylea uno nuevo.
8. Ahora, el número maestro es 110 y el número de transacción es 110. Agregue la transacción de 400.00 al saldo previo de 500.00 en el archivo maestro y escriba un registro maestro nuevo con 900.00 en el campo masterTotal. Lea un registro de cada archivo.

9. Debido a que ambos archivos han terminado, finalice el trabajo. El resultado es un archivo maestro nuevo en el que algunos registros contienen exactamente los mismos datos que al entrar, pero otros (para los que ha ocurrido una transacción) se han actualizado con una cifra de ventas totales nuevas. Los archivos maestro y de transacción originales que se usaron como entrada pueden guardarse como respaldo por algún tiempo.

La figura 7-19 muestra el módulo `finishUp()` para el programa. Después de que se cierran todos los archivos, el archivo maestro de clientes actualizado contiene todos los registros de los clientes que tenía originalmente y cada uno cuenta con un total actual basado en el conjunto reciente de transacciones.

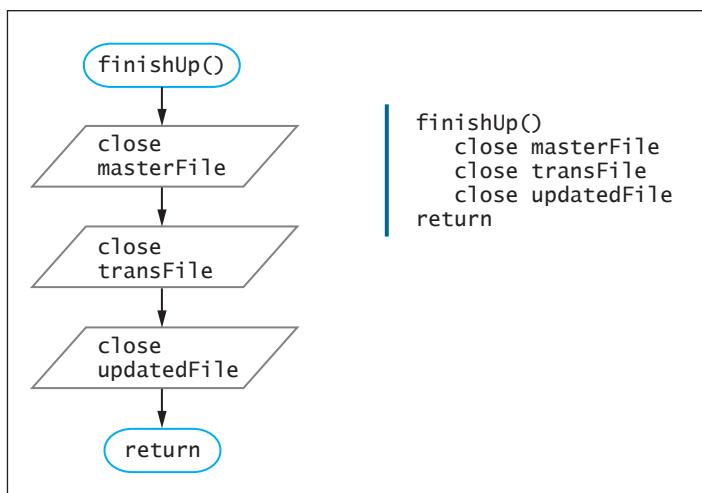


Figura 7-19 El módulo `finishUp()` para el programa maestro-transacción

DOS VERDADES Y UNA MENTIRA

Procesamiento de archivos maestro y de transacción

1. Usted usa un archivo maestro para contener los datos temporales relacionados con los registros del archivo de transacción.
2. Usted usa un archivo de transacción que contenga los datos que se usan para actualizar un archivo maestro.
3. La versión guardada de un archivo maestro es el archivo padre; la versión actualizada es el archivo hijo.

datos relativamente permanentes.

La afirmación falsa es la número 1. Usted usa un archivo maestro para contener

Archivos de acceso aleatorio

Los ejemplos de archivos que se han escrito y leído en este capítulo son de acceso secuencial, lo que significa que acceden a los registros en el orden de una secuencia de principio a fin. Por ejemplo, si usted escribe el registro de un empleado con un número de identificación 234 y luego crea un segundo registro con el número 326, cuando recupere los registros verá que permanecen en el orden original en que los datos entraron. En los negocios los datos se almacenan en orden secuencial cuando se usan los registros para el **procesamiento por lotes**, o que implica ejecutar las mismas tareas con muchos registros, uno detrás otro. Por ejemplo, cuando una empresa genera los cheques de pago, los registros para el periodo de pago se recopilan en un lote y los cheques se calculan e imprimen en secuencia. En realidad no importa de quién es el cheque que se genera primero porque ninguno se entrega a los empleados hasta que todos se han impreso. El procesamiento por lotes por lo general implica cierta demora en el procesamiento; es decir, los registros se recopilan durante un periodo y se procesan juntos después. Por ejemplo, cuando una empresa genera los cheques de pago, los registros podrían recopilarse cada día durante dos semanas antes de que el procesamiento ocurra.



Además de indicar un sistema que trabaja con muchos registros, el término *procesamiento por lotes* puede designar a un sistema en el que usted emite muchos comandos del sistema operativo como un grupo.

Para diversas aplicaciones el acceso secuencial es ineficiente. Éstas, conocidas como aplicaciones **en tiempo real**, requieren que se tenga acceso inmediato a un registro mientras un cliente está en espera. Un programa en el que el usuario hace solicitudes directas es un **programa interactivo**. Por ejemplo, si un cliente habla por teléfono a una tienda de departamentos para preguntar sobre una factura mensual, el representante de servicio al cliente no necesita ni desea tener acceso a las cuentas de todos los clientes en secuencia. Con decenas de miles de registros de cuentas por leer, tomaría demasiado tiempo tener acceso al de ese cliente. En cambio, los representantes de servicio al cliente requieren **archivos de acceso aleatorio**, en los que es posible localizar los registros en cualquier orden. Los archivos para los que es preciso tener acceso inmediato a los registros también se llaman **archivos de acceso instantáneo**. Debido a que permiten localizar directamente un registro en particular (sin leer todos los precedentes), los archivos de acceso aleatorio también se llaman **archivos de acceso directo**. Usted puede declarar un archivo de acceso aleatorio de manera similar a la siguiente:

```
RandomFile customerFile
```

Usted asocia este nombre con un archivo almacenado de igual modo en que vincula un identificador con archivos de entrada y salida secuenciales. Con un archivo de acceso aleatorio también puede usar operaciones de leer, escribir y cerrar; sin embargo, con este tipo de archivos tiene la capacidad adicional de encontrar un registro en forma directa. Por ejemplo, podría usar una declaración similar a la siguiente para encontrar al cliente número 712:

```
seek record 712
```

Esta característica es útil en el procesamiento de acceso aleatorio. Considere un negocio con 20,000 cuentas de clientes. Cuando el cliente cuyo registro en el archivo es 14,607 adquiere

un número de teléfono nuevo, es conveniente tener acceso al registro 14,607 en forma directa escribiendo el nuevo número de teléfono en la ubicación donde se había almacenado el anterior.

DOS VERDADES Y UNA MENTIRA

Archivos de acceso aleatorio

1. Un programa por lotes por lo general usa archivos de acceso instantáneo.
2. En una aplicación en tiempo real, se tiene acceso inmediato a un registro mientras un cliente espera.
3. Un programa interactivo por lo general usa archivos de acceso aleatorio.

La afirmación falsa es la número 1. Un programa por lotes por lo general usa archivos secuenciales; los programas interactivos usan archivos de acceso instantáneo aleatorios.

Resumen del capítulo

- Un archivo de computadora es un conjunto de datos almacenados en un dispositivo no volátil en un sistema de cómputo. Aunque su contenido difiere, cada archivo ocupa un espacio en alguna sección de un dispositivo de almacenamiento y cada uno tiene un nombre y tiempos específicos asociados con él. Se organizan en directorios o carpetas. La lista completa de directorios de un archivo es una ruta.
- Los elementos de datos en un archivo por lo general se almacenan en una jerarquía. Los caracteres son letras, números y símbolos especiales, como *A*, *7* y *\$*. Los campos son elementos de datos que representan un solo atributo de un registro y se componen de uno o más caracteres. Los registros son grupos de campos que se encuentran juntos por alguna razón lógica. Los archivos son grupos de registros relacionados.
- Cuando usted usa un archivo de datos en un programa debe declararlo y abrirlo; al abrir un archivo se asocia un identificador interno del programa con el nombre de un archivo físico en un dispositivo de almacenamiento. Cuando lee de un archivo, los datos se copian a la memoria; cuando lo escribe los datos se copian de la memoria a un dispositivo de almacenamiento. Cuando ha terminado de usarlo, lo cierra.
- Un archivo secuencial es uno en el que se almacenan los registros uno detrás de otro en cierto orden. Un programa de control de interrupciones es el que lee un archivo secuencial y ejecuta el procesamiento especial basado en un cambio en uno o más campos en cada registro en el archivo.
- Unir los archivos implica combinar dos o más mientras se mantiene el orden secuencial.
- Algunos archivos secuenciales relacionados son archivos maestros que contienen datos relativamente permanentes y archivos de transacción que contienen datos temporales. Por lo común, usted recopila transacciones por un periodo, las almacena y luego usa una por una para actualizar los registros correspondientes en un archivo maestro.
- Las aplicaciones interactivas en tiempo real requieren archivos de acceso aleatorio en los que es posible localizar los registros en cualquier orden. Los archivos en los que es preciso tener acceso inmediato a los registros también se denominan de acceso instantáneo y de acceso directo.

Términos clave

Un **archivo de computadora** es un conjunto de datos almacenados en un dispositivo no volátil en un sistema de cómputo.

Los **dispositivos de almacenamiento permanente** contienen datos no volátiles, como discos duros, DVD, unidades USB y carretes de cintas magnéticas.

Los **archivos de texto** contienen datos que pueden leerse en un editor de textos.

Los **archivos binarios** contienen datos que no han sido codificados como texto.

Un **byte** es una unidad pequeña de almacenamiento; por ejemplo, en un archivo de texto simple, un byte contiene sólo un carácter.

Un **kilobyte** consta aproximadamente de 1000 bytes.

Un **megabyte** es un millón de bytes.

Un **gigabyte** son mil millones de bytes.

Los **directorios** son unidades de organización en dispositivos de almacenamiento; cada uno puede contener múltiples archivos y directorios adicionales. En un sistema gráfico, los directorios con frecuencia se llaman *carpetas*.

Las **carpetas** son unidades de organización en dispositivos de almacenamiento; cada una puede contener múltiples archivos y carpetas adicionales. Las carpetas son directorios gráficos.

Una **ruta** de archivo es la combinación de la unidad de disco y la jerarquía completa de directorios en la que reside el archivo.

La **jerarquía de datos** es una estructura que describe las relaciones entre los componentes de datos; contiene caracteres, campos, registros y archivos.

Los **caracteres** son letras, números y símbolos especiales, como A, 7 y \$.

Los **campos** son elementos de datos que representan un solo atributo de un registro y están compuestos por uno o más caracteres.

Los **registros** son grupos de campos que se encuentran juntos por alguna razón lógica.

Los **archivos** son grupos de registros relacionados.

Una **base de datos** contiene grupos de archivos y proporciona métodos para recuperarlos y organizarlos con facilidad.

294

Las **tablas** son archivos en una base de datos.

Abrir un archivo quiere decir que se localiza en un dispositivo de almacenamiento y se le asocia un nombre de variable dentro del programa.

Leer de un archivo implica que se copian los datos del archivo que se encuentra en un dispositivo de almacenamiento hacia la RAM.

Escribir en un archivo significa que se copian los datos de la RAM hacia un almacenamiento persistente.

Cerrar un archivo quiere decir que ya no está disponible para una aplicación.

Los **dispositivos de entrada y salida predeterminados** son aquellos que no requieren abrirse; por lo general son el teclado y el monitor, respectivamente.

Un **archivo de respaldo** es una copia que se conserva en caso de que se necesite restaurar los valores a su estado original.

Un **archivo padre** es una copia de un archivo antes de su revisión.

Un **archivo hijo** es una copia de un archivo después de su revisión.

Clasificar es el proceso de colocar los registros en orden por el valor en un campo o campos específicos.

Un **archivo secuencial** es uno en el que los registros se almacenan uno detrás otro en cierto orden.

Un **control de interrupciones** es una desviación temporal en la lógica de un programa.

Un **programa de control de interrupciones** es uno en el que un cambio en el valor de una variable inicia acciones o procesamiento especiales.

Un **informe de control de interrupciones** es una forma de salida que incluye procesamiento especial después de cada grupo de registros.

Un **control de interrupciones de nivel único** es una interrupción en la lógica de un programa para ejecutar un procesamiento especial basado en el valor de una sola variable.

Un **campo de control de interrupciones** contiene un valor que causa un procesamiento especial en un programa de control de interrupciones.

Unir archivos implica combinar dos o más de ellos mientras mantienen el orden secuencial.

El **orden ascendente** describe los registros que se han ordenado de menor a mayor con base en el valor en un campo.

El **orden descendente** describe registros que se han ordenado de mayor a menor con base en el valor de algún campo.

Un **archivo maestro** contiene datos completos y relativamente permanentes.

Un **archivo de transacción** contiene datos temporales que usted usa para actualizar un archivo maestro.

Actualizar un archivo maestro implica hacer cambios a los valores en sus campos con base en transacciones.

El **procesamiento por lotes** implica ejecutar las mismas tareas con muchos registros, uno detrás de otro.

Las aplicaciones **en tiempo real** requieren que se tenga acceso inmediato a un registro mientras un cliente espera.

En un **programa interactivo**, el usuario hace solicitudes directas, en oposición a aquel en el que la entrada proviene de un archivo.

En los **archivos de acceso aleatorio**, los registros pueden localizarse en cualquier orden.

Los **archivos de acceso instantáneo** son aquellos de acceso aleatorio en los que es preciso tener acceso inmediato a los registros.

Los **archivos de acceso directo** son aquellos de acceso aleatorio.

Preguntas de repaso

1. La memoria de acceso aleatorio es _____.
 - a) permanente
 - b) volátil
 - c) persistente
 - d) continua
2. ¿Cuál afirmación es verdadera en relación con los archivos de texto?
 - a) Contienen datos que pueden leerse en un editor de textos.
 - b) Por lo común contienen imágenes y música.
 - c) Las dos anteriores.
 - d) Ninguna de las anteriores.

3. Cada archivo en un dispositivo de almacenamiento tiene un _____.
 - a) nombre
 - b) tamaño
 - c) las dos anteriores
 - d) ninguna de las anteriores
4. ¿Cuál de las siguientes afirmaciones es verdadera respecto a la jerarquía de datos?
 - a) Los archivos contienen registros.
 - b) Los caracteres contienen campos.
 - c) Los campos contienen archivos.
 - d) Los campos contienen registros.
5. El proceso de _____ un archivo lo localiza en un dispositivo de almacenamiento y le asocia un nombre de variable dentro de su programa.

a) abrir	c) declarar
b) cerrar	d) definir
6. Cuando escribe en un archivo, usted _____.
 - a) mueve los datos desde el dispositivo de almacenamiento hacia la memoria
 - b) copia datos desde un dispositivo de almacenamiento hacia la memoria
 - c) mueve datos desde la memoria hacia un dispositivo de almacenamiento
 - d) copia datos desde la memoria hacia un dispositivo de almacenamiento
7. A diferencia de cuando imprime un informe, cuando la salida de un programa es un archivo de datos, usted no _____.
 - a) incluye encabezados u otros formatos
 - b) abre los archivos
 - c) incluye todos los campos representados como entrada
 - d) todo lo anterior
8. Cuando cierra un archivo, _____.
 - a) éste ya no está disponible para el programa
 - b) no puede reabrirse
 - c) se asocia con un identificador interno
 - d) deja de existir

Ejercicios

1. Vernon Hills Mail Order Company envía con frecuencia múltiples paquetes por pedido. Por cada pedido de los clientes genera una etiqueta para cada caja que se enviará por correo. Las etiquetas contienen el nombre y la dirección completa del cliente, junto con un número de caja de este modo: *caja 9 de 9*. Para un pedido que requiere tres cajas se generan tres etiquetas: *caja 1 de 3*, *caja 2 de 3* y *caja 3 de 3*. Diseñe una aplicación que lea los registros que contienen el título de un cliente (por ejemplo, *señorita*), nombre, apellido, dirección, ciudad, estado, código postal y número de cajas. La aplicación debe leer los registros hasta que se encuentre *eof* y generar suficientes etiquetas de envío para cada pedido.
 2. Cupid Matchmaking Service mantiene dos archivos, uno para los clientes varones y otro para las mujeres. Cada archivo contiene una identificación del cliente, apellido, nombre y dirección; y se han ordenado de acuerdo con la identificación del cliente. Diseñe la lógica para un programa que una los dos archivos en uno que contenga una lista de todos los clientes, manteniendo el orden por número de identificación.
 3. Laramie Park District tiene archivos de las personas que participan en sus programas de verano e invierno este año; cada uno se ha ordenado de acuerdo con el número de identificación del participante y contiene campos adicionales para el nombre, apellido, edad y clase recibida (por ejemplo, *Natación para principiantes*).
 - a) Diseñe la lógica de un programa que una los archivos para los programas de verano e invierno para crear una lista del nombre y apellido de todos los participantes del año.
 - b) Modifique el programa de modo que, si un participante tiene más de un registro, dé salida al nombre del mismo una sola vez.
 - c) Modifique el programa de modo que si un participante tiene más de un registro, dé salida al nombre una sola vez, pero también puede dar salida a un conteo del número total de clases que ha tomado.

4. El grupo Apgar Medical lleva un archivo de pacientes para cada médico en el consultorio; cada registro contiene nombre y apellido, dirección de la casa y año de nacimiento del paciente. Los registros están clasificados en orden ascendente por año de nacimiento. Dos médicos, el doctor Best y el doctor Cushing, han formado una sociedad. Diseñe la lógica que produzca una lista combinada de sus pacientes en orden ascendente por año de nacimiento.
5. Martin Weight Loss Clinic lleva dos archivos de pacientes, uno para los clientes varones y otro para las mujeres. Cada registro contiene el nombre de un paciente y la pérdida de peso total actual en libras. Cada archivo está en orden descendente según la pérdida de peso. Diseñe la lógica que una ambos para producir uno combinado ordenados por pérdida de peso.
6. a) Curl Up and Dye Beauty Salon mantiene un archivo maestro que contiene un registro para cada cliente. Los campos en el archivo maestro incluyen número de identificación del cliente, nombre, apellido y cantidad total gastada este año. Cada semana se genera un archivo de transacción; contiene un número de identificación del cliente, el servicio recibido (digamos, *Manicura*) y el precio pagado. Cada archivo se ordena de acuerdo con el número de identificación. Diseñe la lógica para un programa que relacione los registros de los archivos maestro y de transacción y actualice el total pagado por cada cliente sumando el precio de la semana actual pagado al total acumulado. No todos los clientes compran servicios cada semana. La salida es el archivo maestro actualizado y un informe de errores que lista cualquier registro de transacción para el que no exista un registro maestro.
b) Modifique el programa para dar salida a un cupón para un corte de cabello gratis cada vez que un cliente rebasa \$750.00 en servicios. Al cupón, que contiene el nombre del cliente y un mensaje de felicitación apropiado, se le da salida durante la ejecución del programa de actualización cuando un cliente rebasa el total de \$750.00.

7. a) Timely Talent Temporary Help Agency mantiene un archivo maestro de empleados que contiene su número de identificación, apellido, nombre, dirección y tarifa por hora para cada trabajador temporal. El archivo se ha ordenado de acuerdo con el número de identificación. Cada semana se crea un archivo de transacción con un número de empleo, dirección, nombre del cliente, identificación del empleado y horas trabajadas en cada empleo ocupado por trabajadores de Timely Talent. El archivo de transacción también se ordena de acuerdo con la identificación del empleado. Diseñe la lógica para un programa que relacione los registros de los archivos maestro y de transacción, y dé salida a una línea para cada transacción, indicando el número de empleo, número de identificación del empleado, horas trabajadas, tarifa por horas y pago bruto. Suponga que cada trabajador temporal labora cuando mucho en un empleo por semana; dé salida a una línea por cada trabajador que ha laborado esa semana.
- b) Modifique el programa de la agencia de ayuda de modo que cualquier trabajador temporal pueda laborar en cualquier cantidad de empleos independientes en una semana. Imprima una línea para cada empleo esa semana.
- c) Modifique el programa de la agencia de ayuda de modo que acumule el pago total del trabajador para todos los empleos en una semana y dé salida a una línea por trabajador.



Encuentre los errores

8. Sus archivos descargables para el capítulo 7 incluyen DEBUG07-01.txt, DEBUG07-02.txt y DEBUG07-03.txt. Cada archivo comienza con algunos comentarios que describen el problema. Los comentarios son líneas que comienzan con dos diagonales (//). Después de los comentarios, cada archivo contiene seudocódigo que tiene uno o más errores que usted debe encontrar y corregir. (NOTA: estos archivos se encuentran disponibles sólo para la versión original en inglés.)



Zona de juegos

9. La International Rock Paper Scissors Society celebra campeonatos regionales y nacionales. Cada región celebra una competencia semifinal en la que los concursantes participan en 500 juegos de “Piedra, papel o tijeras”. Los 20 jugadores principales en cada región son invitados a las finales nacionales. Suponga que se le proporcionan archivos para las regiones Este, Medio Oeste y Oeste; cada uno contiene los siguientes campos para los primeros 20 competidores: apellido, nombre y número de juegos ganados. Los registros en cada archivo están clasificados en orden alfabético. Una los tres archivos para crear uno solo de los 60 competidores principales que participarán en el campeonato nacional.
10. En la sección “Zona de juegos” del capítulo 5 diseñó un juego de adivinanzas en el que la aplicación genera un número aleatorio y el jugador trata de adivinarlo. Después de cada adivinanza, desplegó un mensaje indicando si la adivinanza del jugador era correcta, demasiado alta o demasiado baja. Cuando el jugador al fin adivinaba el número correcto, desplegaba una puntuación que representaba un conteo del número de adivinanzas requeridas. Modifique el juego de modo que cuando comience, el jugador introduzca su nombre. Después de que un jugador participa en el juego exactamente cinco veces, guarde la mejor puntuación (más baja) de los cinco juegos en un archivo. Si el nombre del jugador ya existe en el archivo, actualice el registro con la nueva puntuación más baja; si no existe, cree para él un registro nuevo. Luego de que el archivo se actualice, despliegue todas las mejores puntuaciones almacenadas en el archivo.



Para discusión

303

11. Suponga que un departamento de policía lo contrata para escribir un programa que relacione los registros de arresto con los judiciales que detallen el último resultado o veredicto para cada caso. Se le ha dado acceso a los archivos actuales de modo que le sea posible probar el programa. Su amigo trabaja en el departamento de personal de una compañía grande y debe verificar los antecedentes de los empleados potenciales. (Los solicitantes de empleo firman un formato autorizando la verificación.) Los registros de policía están abiertos al público y su amigo podría buscarlos en el juzgado, pero le tomaría muchas horas por semana. Por conveniencia, ¿debería usted proporcionarle los resultados de cualesquier registros de arresto de los solicitantes de empleo?

12. Suponga que una clínica lo contrata para relacionar un archivo de las visitas de los pacientes al consultorio con sus registros maestros para imprimir varios informes. Mientras trabaja con los datos confidenciales, observa el nombre de la novia de un amigo. ¿Debería decir a éste que su novia busca tratamiento médico? ¿El tipo de tratamiento afecta su respuesta?

APÉNDICE A

Comprensión de los sistemas de numeración y los códigos de computadora

El sistema de numeración que usted conoce mejor es el **sistema de numeración decimal**, basado en 10 dígitos, 0 a 9. Los matemáticos llaman a los números de este sistema números de **base 10**. Cuando usa el sistema decimal, ningún otro símbolo está disponible; si desea expresar un valor más grande que 9 debe usar dígitos múltiples del mismo conjunto de 10, colocándolos en columnas.

Cuando usted usa el sistema decimal analiza un número de múltiples columnas asignando mentalmente los valores de lugar a cada columna. El valor de la columna de la extrema derecha es 1, el de la siguiente columna a la izquierda es 10, la siguiente es 100, y así sucesivamente; los valores de una columna se multiplican por 10 conforme se mueven hacia la izquierda. No hay límite para el número de columnas que puede usar; sólo las añade hacia la izquierda conforme necesita expresar valores más altos. Por ejemplo, la figura A-1 muestra cómo se representa el valor 305 en el sistema decimal. Sólo suma el valor del dígito en cada columna después de que se ha multiplicado por el valor de su columna.

Valor de columna	100	10	1
3	0	5	
3 * 100 = 300			
0 * 10 = 0			
5 * 1 = 5			
			305

Figura A-1 Representación de 305 en el sistema decimal

El **sistema de numeración binario** funciona igual que el decimal, excepto que usa sólo dos dígitos, 0 y 1. Los matemáticos los llaman números de **base 2**. Cuando usted usa el sistema binario debe utilizar columnas múltiples si desea expresar un valor mayor que 1 debido a que no hay un solo símbolo disponible que represente cualquier valor distinto de 0 o 1. Sin embargo, en lugar de que cada columna a la izquierda sea 10 veces mayor que la anterior, cada columna nueva en el sistema binario sólo es dos veces el valor de la previa. Por ejemplo, la figura A-2 muestra cómo se representan los números 9 y 305 en el sistema binario. Note que tanto en el binario como en el decimal es perfectamente aceptable, y a menudo necesario,

crear números con 0 en una o más columnas. Como con el sistema decimal, no hay límite al número de columnas que se usan en un número binario: usted puede usar tantas como se requieran para expresar un valor.

306

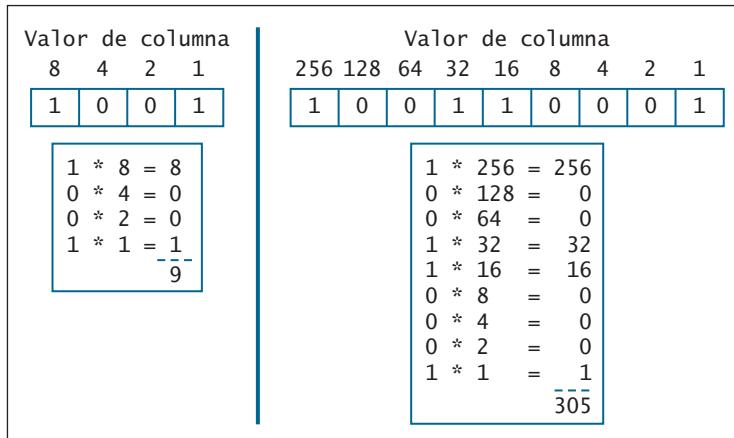


Figura A-2 Representación de los valores decimales 9 y 305 en el sistema binario

Una computadora almacena cada pieza de datos que usa como un conjunto de 0 y 1. Cada 0 o 1 se conoce como un **bit**, que es una abreviatura para *binary digit* (dígito binario). Todas las computadoras usan 0 y 1 porque todos los valores se almacenan como señales electrónicas que están ya sea encendidas o apagadas. Este sistema de dos estados se representa con más facilidad usando sólo dos dígitos.

Las computadoras usan un conjunto de dígitos binarios para representar los caracteres almacenados. Si las computadoras usaran sólo un dígito binario entonces sólo podrían representarse dos caracteres diferentes, debido a que el único bit sólo podría ser 0 o 1. Si las computadoras únicamente usaran dos dígitos, entonces sólo podrían representarse cuatro caracteres: los cuatro códigos 00, 01, 10 y 11, que en valores decimales son 0, 1, 2 y 3, respectivamente. Muchas computadoras usan conjuntos de ocho dígitos binarios para representar cada carácter que almacenan, debido a que esto proporciona 256 combinaciones diferentes. Un conjunto de ocho bits es un **byte**. Una combinación de un byte puede representar una *A*, otra una *B*, otras más *a* y *b*, y así sucesivamente. Doscientas cincuenta y seis combinaciones son suficientes de modo que cada letra mayúscula, letra minúscula, dígito y signo de puntuación que se usan en inglés tenga su propio código; incluso un espacio tiene un código. Por ejemplo, en el sistema que se conoce como **American Standard Code for Information Interchange (ASCII; Código Estándar Estadounidense para el Intercambio de Información)**, 01000001 representa el carácter *A*. El número binario 01000001 tiene un valor decimal de 65, pero este valor numérico no es importante para los usuarios de computadoras ordinarias; simplemente es un código que representa la *A*.

El ASCII no es el único código de computadora, pero es típico y se usa en la mayoría de las computadoras personales. El **Extended Binary Coded Decimal Interchange Code**, o **EBCDIC (Código ampliado de intercambio decimal codificado en binario)**, es un código de ocho bits que se usa en computadoras mainframe de IBM. En estas computadoras el principio es el mismo, cada carácter se almacena en un byte como una serie de dígitos binarios. Sin embargo,

los valores reales que se usan son diferentes. Por ejemplo, en EBCDIC, una *A* es 11000001, o 193. Otro código de lenguajes como Java y C# es **Unicode**; con éste se usan 16 bits para representar cada carácter. El carácter *A* en Unicode tiene el mismo valor decimal que la *A* en ASCII, 65, pero se almacena como 0000000010000001. Usar dos bytes proporciona muchas más combinaciones posibles que usar sólo ocho bits: 65,536 para ser exactos. Con Unicode hay disponibles suficientes códigos para representar todas las letras en inglés y los dígitos, al igual que caracteres de muchos alfabetos internacionales.

Los usuarios de las computadoras rara vez piensan en los códigos numéricos detrás de las letras, los números y los signos de puntuación que introducen desde sus teclados o ven desplegados en un monitor. Sin embargo, observan la consecuencia de los valores detrás de las letras cuando ven datos clasificados en orden alfabético. Cuando usted clasifica una lista de nombres, *Andrea* va antes que *Bruno* y *Carolina* va después de *Bruno* porque el código numérico para *A* es menor que el código para *B*, y el de *C* es mayor que el de *B* sin importar que use ASCII, EBCDIC o Unicode.

El cuadro A-1 muestra los valores decimal y binario detrás de los caracteres que más se usan en el conjunto de caracteres ASCII: las letras, los números y los signos de puntuación que usted puede introducir desde su teclado con sólo oprimir una tecla (otros valores que no se muestran en el cuadro A-1 también tienen propósitos específicos. Por ejemplo, cuando usted despliega el carácter que contiene el valor decimal 7 no aparece nada en la pantalla pero suena una campana. Los programadores usan con frecuencia este carácter cuando desean alertar a un usuario de un error o alguna otra condición extraña).



Cada número binario en el cuadro A-1 se muestra con dos grupos de cuatro dígitos; esta convención hace más fácil leer los números de ocho dígitos. Cuatro dígitos, o medio byte, son un **nibble**.

Número decimal	Número binario	Carácter ASCII
32	0010 0000	Espacio
33	0010 0001	! Signo de admiración de cierre
34	0010 0010	“ Comillas
35	0010 0011	# Signo de número, también llamado almohadilla o numeral
36	0010 0100	\$ Signo de dólar
37	0010 0101	% Signo de porcentaje
38	0010 0110	& et o ampersand
39	0010 0111	’ Apóstrofo, o comilla simple
40	0010 1000	(Paréntesis izquierdo
41	0010 1001) Paréntesis derecho
42	0010 1010	* Asterisco

Cuadro A-1 Valores decimales y binarios para los caracteres ASCII comunes (continúa)

Número decimal	Número binario	Carácter ASCII
43	0010 1011	+ Signo de adición
44	0010 1100	, Coma
45	0010 1101	- Guión o signo de resta
46	0010 1110	. Punto
47	0010 1111	/ Diagonal
48	0011 0000	0
49	0011 0001	1
50	0011 0010	2
51	0011 0011	3
52	0011 0100	4
53	0011 0101	5
54	0011 0110	6
55	0011 0111	7
56	0011 1000	8
57	0011 1001	9
58	0011 1010	: Dos puntos
59	0011 1011	; Punto y coma
60	0011 1100	< Signo menor que
61	0011 1101	= Signo igual
62	0011 1110	> Signo mayor que
63	0011 1111	? Signo de interrogación
64	0100 0000	@ Arroba
65	0100 0001	A
66	0100 0010	B
67	0100 0011	C
68	0100 0100	D
69	0100 0101	E
70	0100 0110	F

Cuadro A-1 Valores decimales y binarios para caracteres ASCII comunes (continuación)

Número decimal	Número binario	Carácter ASCII
71	0100 0111	G
72	0100 1000	H
73	0100 1001	I
74	0100 1010	J
75	0100 1011	K
76	0100 1100	L
77	0100 1101	M
78	0100 1110	N
79	0100 1111	O
80	0101 0000	P
81	0101 0001	Q
82	0101 0010	R
83	0101 0011	S
84	0101 0100	T
85	0101 0101	U
86	0101 0110	V
87	0101 0111	W
88	0101 1000	X
89	0101 1001	Y
90	0101 1010	Z
91	0101 1011	[Corchete izquierdo o de apertura
92	0101 1100	\ Diagonal invertida
93	0101 1101] Corchete derecho o de cierre
94	0101 1110	^ Acento circunflejo
95	0101 1111	_ Subrayado o guión bajo
96	0110 0000	` Acento grave
97	0110 0001	a
98	0110 0010	b

Cuadro A-1 Valores decimales y binarios para caracteres ASCII comunes (continuación)

Número decimal	Número binario	Carácter ASCII
99	0110 0011	c
100	0110 0100	d
101	0110 0101	e
102	0110 0110	f
103	0110 0111	g
104	0110 1000	h
105	0110 1001	i
106	0110 1010	j
107	0110 1011	k
108	0110 1100	l
109	0110 1101	m
110	0110 1110	n
111	0110 1111	o
112	0111 0000	p
113	0111 0001	q
114	0111 0010	r
115	0111 0011	s
116	0111 0100	t
117	0111 0101	u
118	0111 0110	v
119	0111 0111	w
120	0111 1000	x
121	0111 1001	y
122	0111 1010	z
123	0111 1011	{ Llave izquierda o de apertura
124	0111 1100	Línea vertical
125	0111 1101	} Llave derecha o de cierre
126	0111 1110	~ Tilde

Cuadro A-1 Valores decimales y binarios para caracteres ASCII comunes (continuación)

El sistema hexadecimal

El **sistema de numeración hexadecimal** es el sistema de **base 16**; usa 16 dígitos. Como se muestra en el cuadro A-2, los dígitos son 0 a 9 y A a F. Los profesionales de la computación usan con frecuencia el sistema hexadecimal para expresar direcciones e instrucciones cuando están almacenadas en la memoria de la computadora debido a que el hexadecimal proporciona expresiones abreviadas convenientes para los grupos de valores binarios. En el cuadro A-2 cada valor hexadecimal representa una de las 16 combinaciones posibles de los valores binarios de cuatro dígitos. Por consiguiente, en lugar de referenciar el contenido de la memoria como un valor binario de 16 dígitos, por ejemplo, los programadores pueden usar un valor hexadecimal de 4 dígitos.

Valor decimal	Valor hexadecimal	Valor binario (que se muestra usando cuatro dígitos)
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Cuadro A-2 Valores en los sistemas decimal y hexadecimal

En el sistema hexadecimal, cada columna es 16 veces el valor de la columna a su derecha. Por consiguiente, los valores de columna de derecha a izquierda son 1, 16, 256, 4096, y así sucesivamente. La figura A-3 muestra cómo se expresan 78, 171 y 305 en hexadecimal.

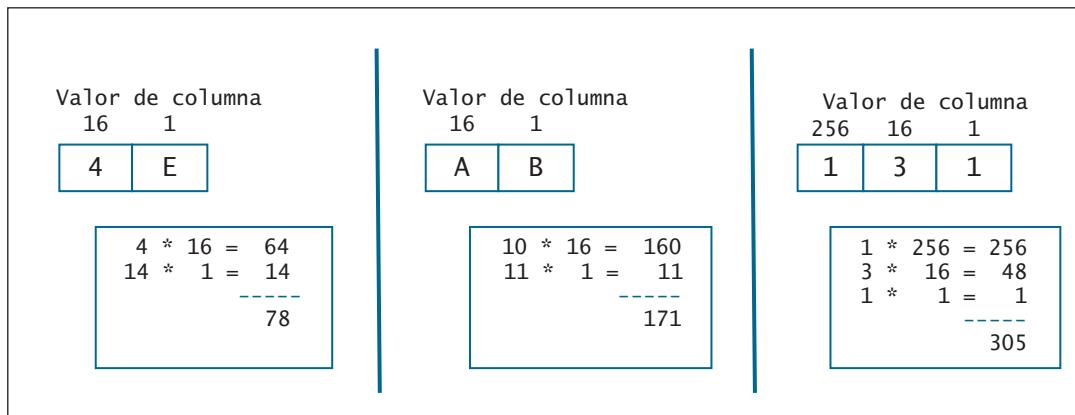


Figura A-3 Representación de los valores decimales 78, 171 y 305 en el sistema hexadecimal

Medición del almacenamiento

En los sistemas de cómputo, tanto la memoria interna como el almacenamiento externo se miden en bits y bytes. Ocho bits hacen un byte, y un byte con frecuencia contiene un solo carácter (en ASCII o EBCDIC) o medio carácter (en Unicode). Debido a que un byte es una unidad de almacenamiento muy pequeña, el tamaño de la memoria y los archivos con frecuencia se expresa en miles o millones de bytes. El cuadro A-3 describe algunos términos que se usan de manera común para la medición del almacenamiento.

Término	Abreviatura	Número de bytes usando sistema binario	Número de bytes usando sistema decimal	Ejemplo
Kilobyte	KB o kB	1024	mil	Este apéndice ocupa alrededor de 85 kB en un disco duro.
Megabyte	MB	1,048,576 (1024 × 1024 kilobytes)	un millón	Un megabyte puede contener un libro promedio en formato de texto. Un disquete de 3.5 pulgadas que usted quizá usó hace algunos años contenía 1.44 megabytes.
Gigabyte	GB	1,073,741,824 (1,024 megabytes)	un billón	El disco duro en una computadora laptop nueva tiene al menos 250 gigabytes. Un DVD_R puede contener alrededor de 5 gigabytes.
Terabyte	TB	1024 gigabytes	un trillón	Algunos discos duros son de 1 terabyte. La Biblioteca del Congreso entera ocupaba alrededor de 300 terabytes cuando se publicó este libro.
Petabyte	PB	1024 terabytes	un cuatrillón	El sitio Web de Google procesa alrededor de 24 petabytes por día.
Exabyte	EB	1024 petabytes	un quintillón	Una expresión común afirma que todas las palabras que han sido expresadas por los humanos alguna vez podrían ser almacenadas en forma de texto en 5 exabytes.
Zettabyte	ZB	1024 exabytes	un sextillón	Una expresión popular afirma que todas las palabras que han sido expresadas por los humanos alguna vez podrían ser almacenadas en forma de audio en 42 zettabytes.
Yottabyte	YB	1024 zettabytes	un septillón (un 1 seguido por 36 ceros)	El espacio combinado en todos los discos duros en el mundo es menos de 1 yottabyte.

Cuadro A-3 Términos que se usan de manera común para el almacenamiento en computadoras

En el sistema métrico, *kilo* significa 1000. Sin embargo, en el cuadro A-3 notará que un kilobyte consta de 1024 bytes. La discrepancia ocurre debido a que todo lo que se almacena en una computadora se basa en el sistema binario, así que se usan múltiplos de dos en la mayoría de las mediciones. Si usted multiplica 2 por sí mismo 10 veces, el resultado es 1024, lo que es un poco más de 1000. Del mismo modo, un gigabyte es 1,073,741,824 bytes, que es poco más de mil millones.

La confusión surge debido a que muchos fabricantes de discos duros usan el sistema decimal en lugar del sistema binario para describir el almacenamiento. Por ejemplo, si usted compra un disco duro que contiene 10 gigabytes, son exactamente 10,000 millones de bytes. Sin embargo, en el sistema binario, 10 GB es 10,737,418,240 bytes, de modo que cuando compruebe la capacidad de su disco duro, su computadora informará que no tiene 10 GB, sino sólo 9.31 GB.

Términos clave

El **sistema de numeración decimal** es el sistema de numeración basado en 10 dígitos y en el que los valores de columna son múltiplos de 10.

Base 10 describe los números creados usando el sistema de numeración decimal.

El **sistema de numeración binario** se basa en 2 dígitos; en él los valores de columna son múltiplos de 2.

Base 2 describe los números creados usando el sistema de numeración binario.

Un **bit** es un dígito binario; es una unidad de almacenamiento igual a un octavo de byte.

Un **byte** es una medida de almacenamiento igual a ocho bits.

El **American Standard Code for Information Interchange (ASCII; Código Estándar Estadounidense para el Intercambio de Información)** es un esquema de codificación de caracteres de ocho bits que se usa en muchas computadoras personales.

El **Extended Binary Coded Decimal Interchange Code (EBCDIC; Código ampliado de intercambio decimal codificado en binario)** es un esquema de codificación de caracteres de ocho bits que se usa en muchas computadoras grandes.

Unicode es un esquema de codificación de caracteres de 16 bits.

Un **nibble** es una medida de almacenamiento igual a cuatro bits, o medio byte.

El **sistema de numeración hexadecimal** es el sistema de numeración basado en 16 dígitos y en el cual los valores de columna son múltiplos de 16.

Base 16 describe los números creados usando el sistema de numeración hexadecimal.

APÉNDICE

B

Símbolos de diagrama de flujo

Este apéndice contiene los símbolos de diagrama de flujo que se usan en este libro.

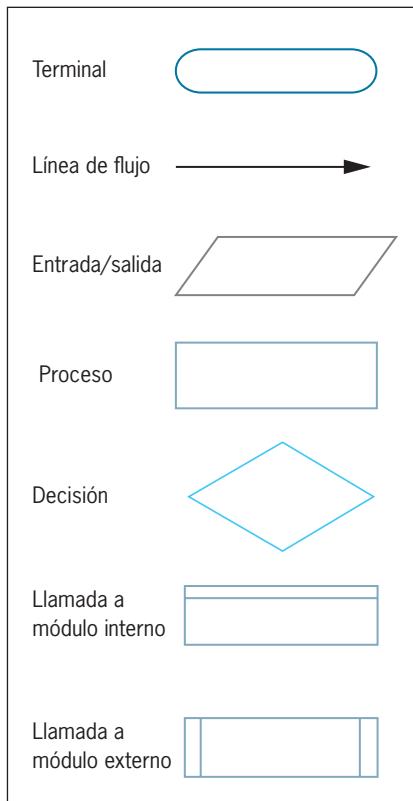


Figura B-1 Símbolos de diagrama de flujo

APÉNDICE

C

Estructuras

Este apéndice contiene diagramas de las estructuras permitidas en la programación estructurada. Aunque todos los problemas lógicos pueden resolverse usando las tres que son fundamentales, las adicionales son convenientes en algunas situaciones. Cada estructura tiene un punto de entrada y uno de salida. En estos puntos, es posible apilarlas y anidarlas.

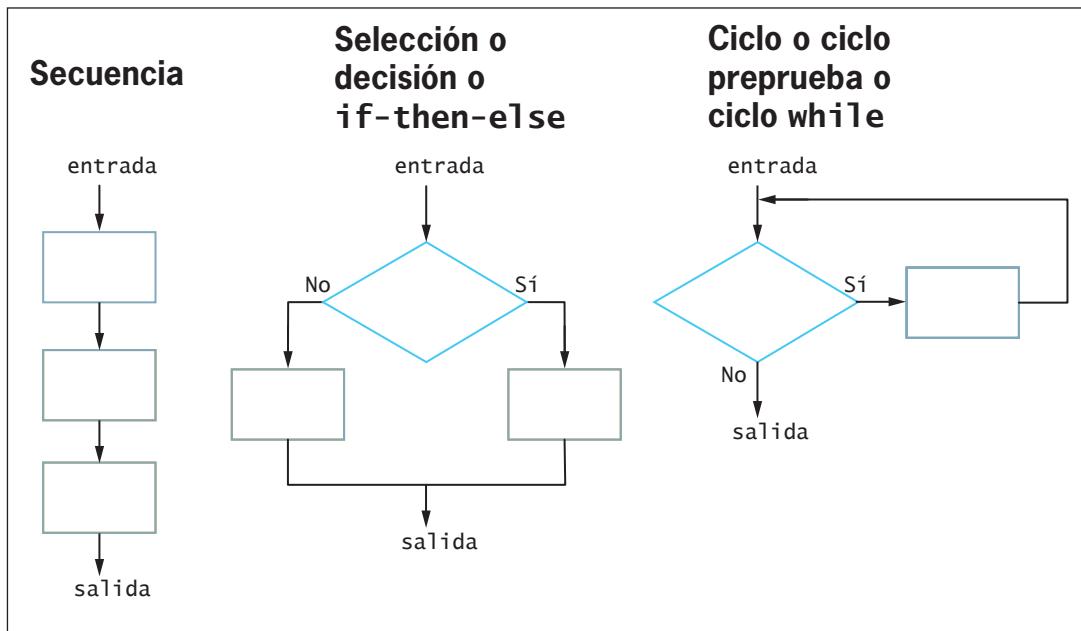
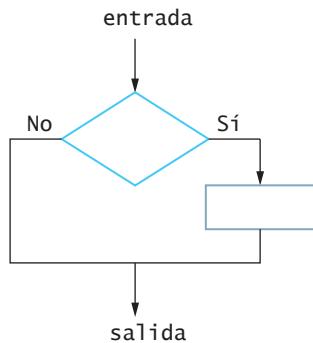


Figura C-1 Tres estructuras fundamentales

Decisión de alternativa única o if-then



Case

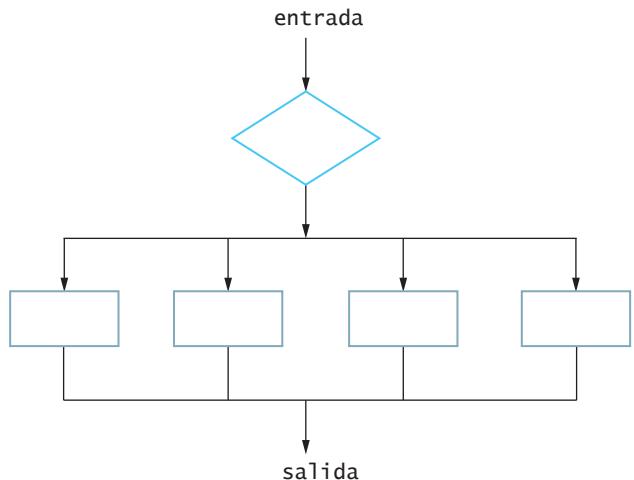


Figura C-2 Estructuras de selección adicionales

Ciclo posprueba o ciclo do-while

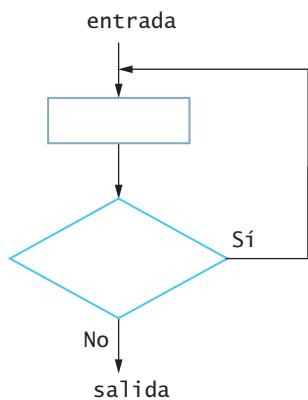


Figura C-3 Estructura de ciclo adicional

Resolución de problemas de estructuración difíciles

En el capítulo 3 usted aprendió que puede resolver cualquier problema lógico usando sólo las tres estructuras estándar: secuencia, selección y ciclo. Modificar un programa no estructurado para que se adhiera a las reglas estructuradas a menudo es algo sencillo. Sin embargo, en ocasiones estructurar un programa más complicado puede ser desafiante. Aun así, sin importar cuán complicado, extenso o mal estructurado sea un problema, las mismas tareas pueden lograrse *siempre* de una manera estructurada.

Considere el segmento de diagrama de flujo en la figura D-1. ¿Está estructurado?

No, no lo está. Para aclarar el segmento de diagrama de flujo haciéndolo estructurado, usted puede usar el método “espagueti”. Desenrede cada ruta del diagrama de flujo como si lo hiciera con las hebras de espagueti en un tazón. El objetivo es crear un nuevo segmento que ejecute exactamente las mismas tareas que el primero, pero usando sólo las tres estructuras: secuencia, selección y ciclo.

Para comenzar a desenredar el segmento de diagrama de flujo no estructurado, empiece con la decisión etiquetada A, que se muestra en la figura D-2. Este paso debe representar el comienzo de una selección o un ciclo, debido a que una secuencia no contendría una decisión.

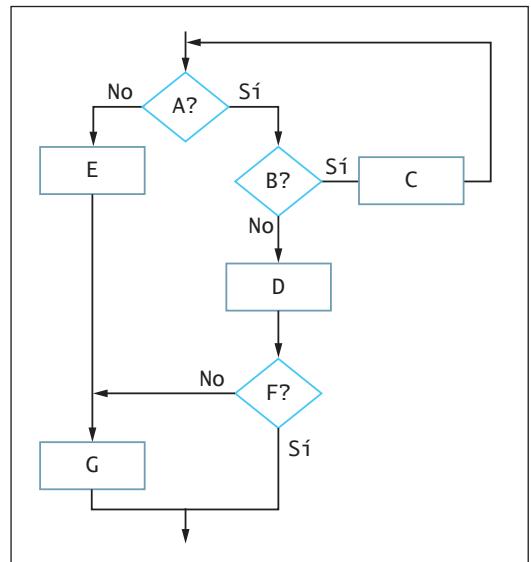


Figura D-1 Segmento de diagrama de flujo no estructurado

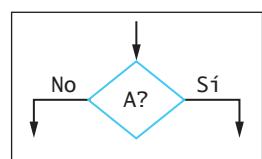


Figura D-2 Estructuración, paso 1

Si usted sigue la lógica en el lado del *No*, o izquierdo, de la pregunta en el diagrama de flujo original, puede seguir la rama izquierda de la decisión. Encontrará el proceso E, seguido por G, y después el final, como se muestra en la figura D-3. Compare las acciones *No* después de la decisión A en el primer diagrama de flujo (figura D-1) con las acciones posteriores a la decisión A en la figura D-3; son idénticas.

Ahora continúe del lado derecho, o de *Sí*, de la decisión A en la figura D-1. Cuando siga la línea de flujo, encontrará un símbolo de decisión etiquetado como B. Siga el lado izquierdo de B y a continuación surge un proceso D. Véase la figura D-4.

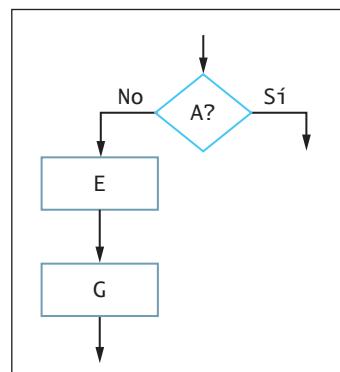


Figura D-3 Estructuración, paso 2

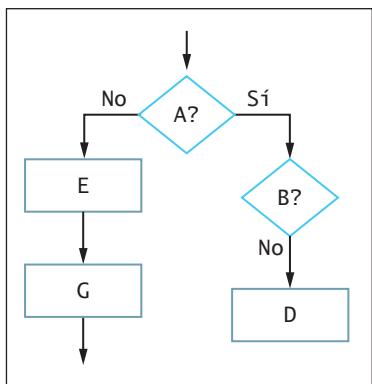
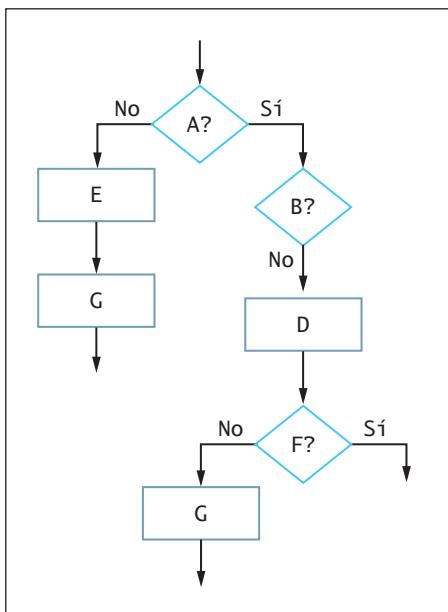
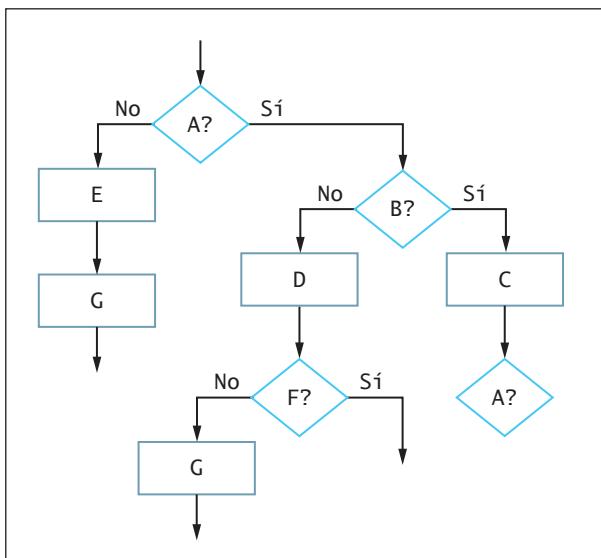
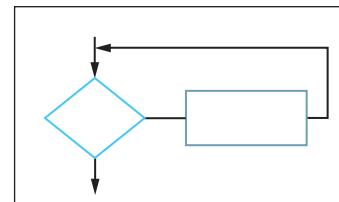


Figura D-4 Estructuración, paso 3

Después del paso D en el diagrama original, se encuentra una decisión etiquetada como F. Siga su lado izquierdo, o *No*, y obtenga el proceso G, y luego el final. Cuando sigue el lado derecho, o *Sí*, de F en el diagrama de flujo original, simplemente llega al final, como se muestra en la figura D-5. Note en ésta que el proceso G ahora aparece en dos ubicaciones. Cuando modifique los diagramas de flujo no estructurados de modo que se vuelvan estructurados, con frecuencia debe repetir pasos para eliminar las líneas cruzadas y la lógica espagueti que es difícil de seguir.

**Figura D-5** Estructuración, paso 4

El problema más grande en la estructuración del segmento de diagrama de flujo original de la figura D-1 sigue el lado derecho, o *Sí*, de la decisión B. Cuando la respuesta a B es *Sí*, usted encuentra el proceso C, como se muestra en las figuras D-1 y D-6. La estructura que comienza con la decisión C parece un ciclo porque regresa a la decisión A. Sin embargo, un ciclo estructurado debe tener la apariencia que se muestra en la figura D-7; una pregunta seguida por una estructura, regresando de inmediato a la pregunta. En la figura D-1, si la ruta que viene de C regresara directamente a B, no habría problema; sería un ciclo estructurado simple. Sin embargo, tal como está, la pregunta A debe repetirse. La técnica espagueti requiere que, si las líneas de lógica están enmarañadas, se repitan los pasos en cuestión. Así, usted repite una decisión A después de C, como se observa en la figura D-6.

**Figura D-6** Estructuración, paso 5**Figura D-7** Un ciclo estructurado

En el segmento de diagrama de flujo original en la figura D-1, cuando A es *Sí*, siempre sigue la pregunta B. Así, en la figura D-8, después de que A es *Sí* y B es *Sí*, se ejecuta el paso C, y A se pregunta de nuevo; cuando A es *Sí*, B se repite. En el original, cuando B es *Sí*, se ejecuta C, así en la figura D-8, en el lado derecho de B, C se repite. Después de C, ocurre A. En el lado derecho de A, ocurre B. En el lado derecho de B, ocurre C. Después de C, A debería ocurrir de nuevo, y así sucesivamente. Usted pronto se dará cuenta de que, al seguir los pasos en el mismo orden que en el segmento de diagrama de flujo original, siempre los repetirá.

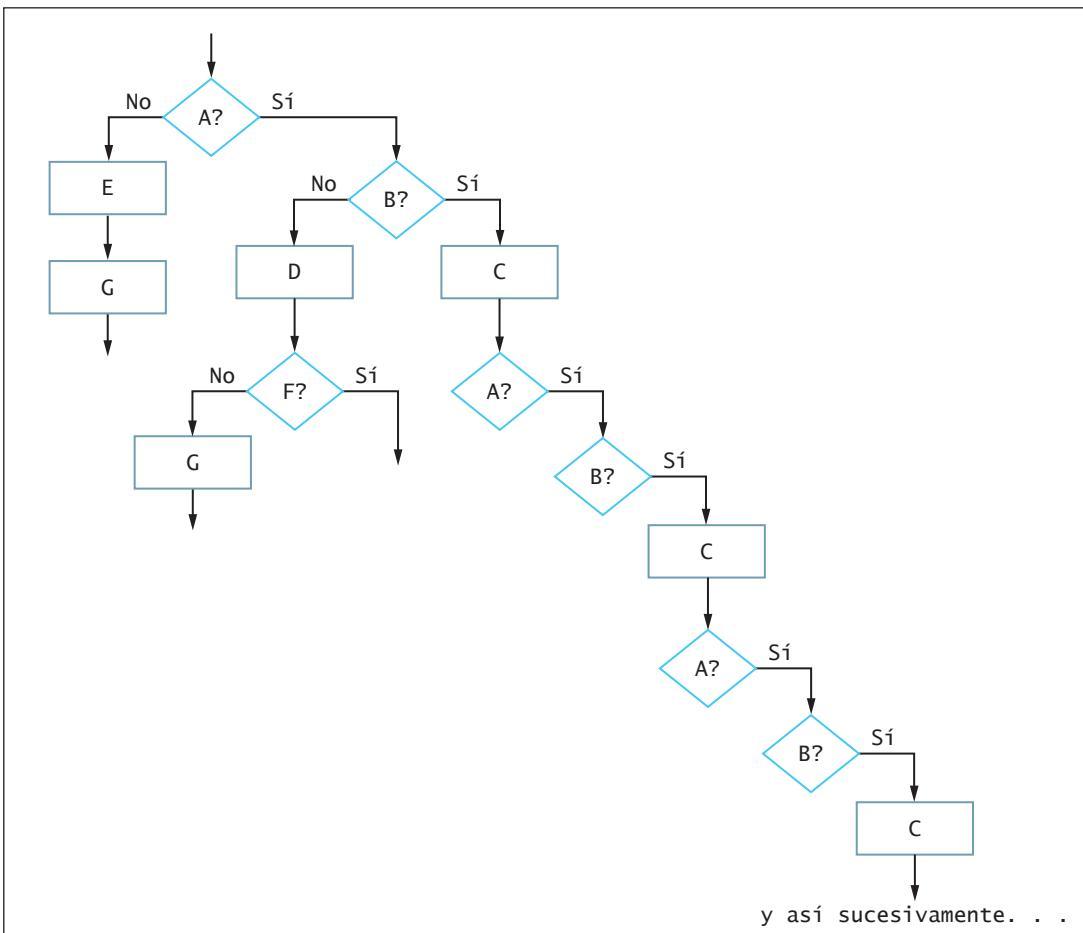


Figura D-8 Estructuración, paso 6

Si continúa con la figura D-8, nunca podrá terminar el diagrama de flujo; cada C siempre es seguida por otra A, B y C. En ocasiones, para estructurar un segmento de programa, tiene que agregar una variable bandera extra para salir de un lío infinito. Una bandera es una variable que usted establece para indicar un estado verdadero o falso. Por lo común, una variable se llama bandera cuando su único propósito es decirle si ha ocurrido un evento. Puede crear una variable bandera nombrada `shouldRepeat` y establecer su valor en `Sí` o `No`, dependiendo de si es apropiado repetir la decisión A. Cuando A es `No`, la bandera `shouldRepeat` debería establecerse en `No` debido a que, en esta situación, usted nunca deseará repetir de nuevo la pregunta A. Véase la figura D-9.

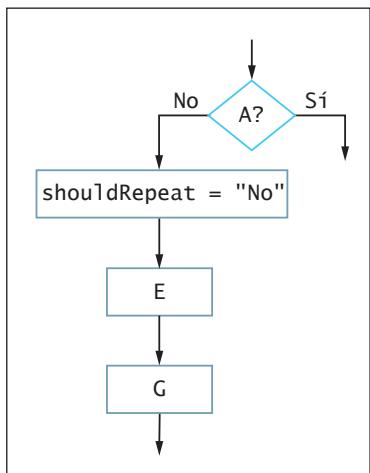


Figura D-9 Agregar una bandera al diagrama de flujo

Del mismo modo, después de que A es *Sí*, pero cuando B es *No*, usted nunca deseará repetir de nuevo la pregunta A. La figura D-10 muestra que usted establece `shouldRepeat` en *No* cuando la respuesta a B es *No*. Luego continúa con D y la decisión F que ejecuta G cuando F es *No*.

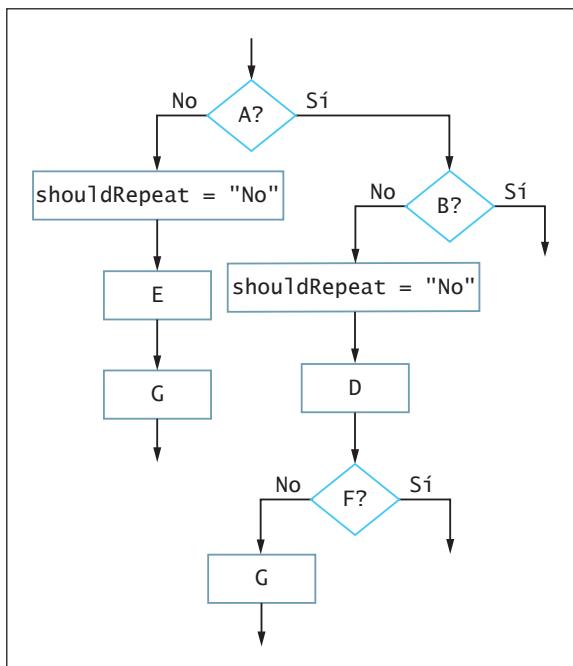


Figura D-10 Agregar una bandera a una segunda ruta en el diagrama de flujo

Sin embargo, en el segmento de diagrama de flujo original en la figura D-1, cuando el resultado de la decisión B es *Sí*, usted *desea* repetir A. Así, cuando B es *Sí*, ejecuta el proceso para C y establece la bandera `shouldRepeat` igual a *Sí*, como se muestra en la figura D-11.

324

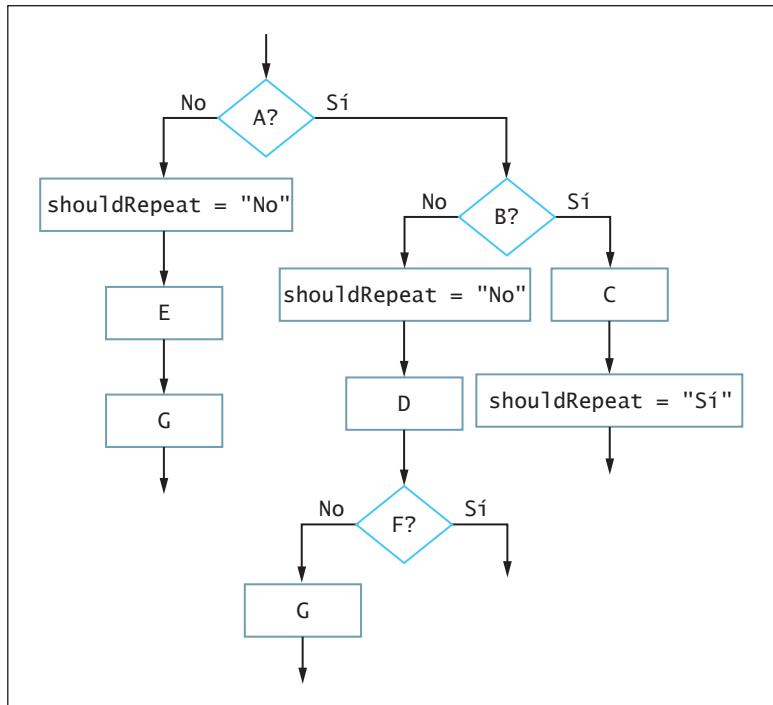


Figura D-11 Agregar una bandera a una tercera ruta en el diagrama de flujo

Ahora todas las rutas del diagrama de flujo pueden unirse en la parte inferior con una pregunta final: ¿`shouldRepeat` es igual a *Sí*? Si no lo es, sale; pero si lo es, extienda la línea de flujo para regresar a repetir la pregunta A. Véase la figura D-12. Tome un momento para verificar que los pasos que ejecutaría siguiendo la figura D-12 son los mismos que haría siguiendo D-1.

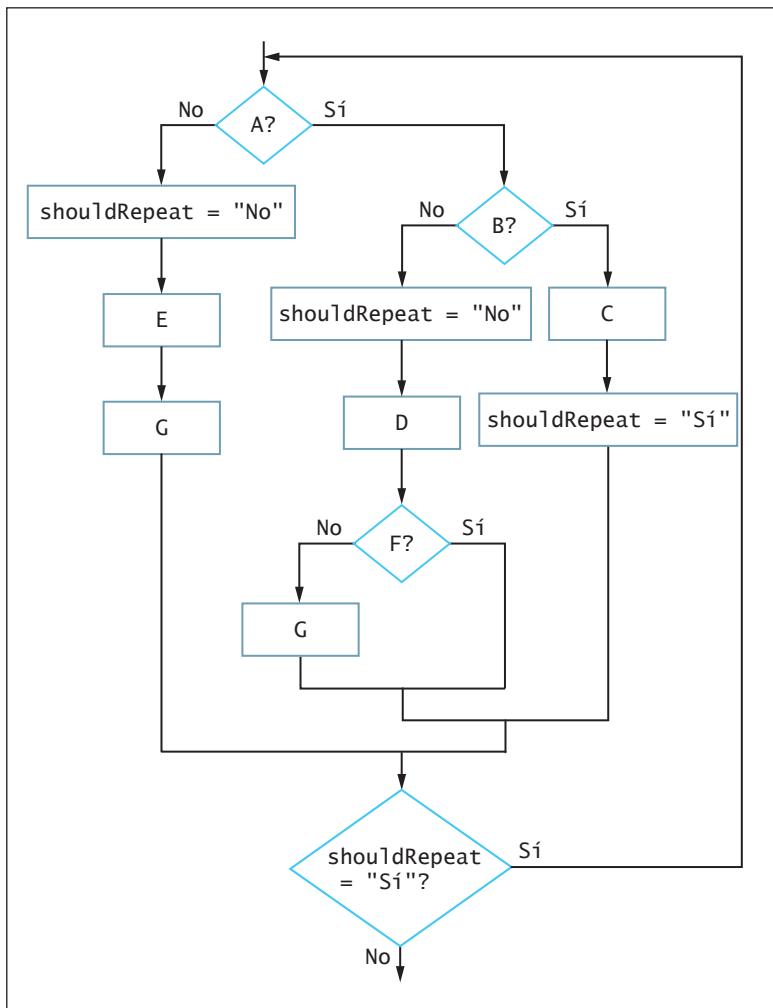


Figura D-12 Atar los cabos sueltos

- Cuando A es *No*, E y G siempre se ejecutan.
- Cuando A es *Sí* y B es *No*, D y la decisión F siempre se ejecutan.
- Cuando A es *Sí* y B es *Sí*, C siempre se ejecuta y A se repite.



La figura D-12 contiene tres estructuras de selección anidadas. Observe cómo la decisión F comienza una estructura de selección completa cuyas rutas Sí y No se unen cuando la estructura termina. Esta estructura de selección F está dentro de una ruta de la estructura de decisión B; la decisión B comienza una estructura de selección completa cuyas rutas Sí y No se unen en la parte inferior. Del mismo modo, la estructura de selección B reside por entero dentro de una ruta de la estructura de selección A.

El segmento de diagrama de flujo en la figura D-12 se ejecuta de manera idéntica a la versión espagueti original en la figura D-1. Sin embargo, ¿está estructurado el nuevo segmento de diagrama de flujo? Hay tantos pasos en él que es difícil decirlo. Usted puede ver la estructura con más claridad si crea un módulo llamado `aThroughG()`. Si crea el módulo que se muestra en la figura D-13, entonces el segmento de diagrama de flujo original puede trazarse como en la figura D-14.

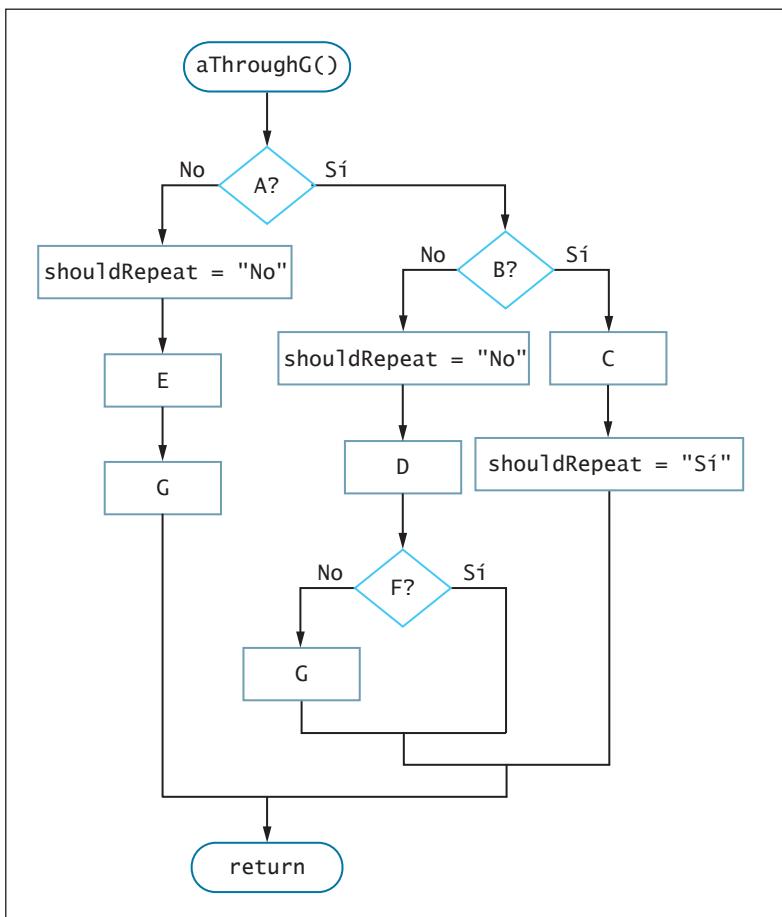


Figura D-13 El módulo `aThroughG()`

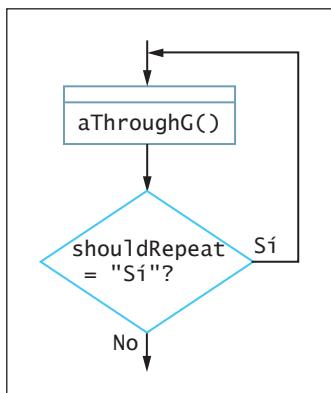


Figura D-14 Lógica en la figura D-12, que sustituye un módulo por los pasos A a G

Ahora usted puede ver que el segmento de diagrama de flujo completado en la figura D-14 es un ciclo `do-until`. Si prefiere usar un ciclo `while`, puede volver a trazar la figura D-14 para ejecutar una secuencia seguida por un ciclo `while`, como se muestra en la figura D-15.

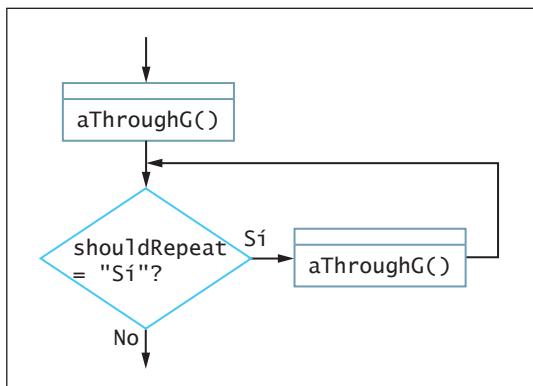


Figura D-15 Lógica en la figura D-14, que sustituye una secuencia y un ciclo `while` por el ciclo `do-until`

Ha requerido algún esfuerzo extra, incluyendo la repetición de pasos específicos y el uso de algunas variables bandera, pero todo problema lógico puede resolverse y es posible hacer que se ajuste a las reglas estructuradas usando las tres estructuras: secuencia, selección y ciclo.

APÉNDICE E

Creación de gráficas impresas

Un informe impreso es un tipo de salida muy común. Usted puede diseñarlo en una gráfica de espaciado de impresora, que también se llama gráfica de impresión o composición de impresión. Muchos programadores de hoy en día usan varias herramientas de software para diseñar su salida, pero usted también puede crear una gráfica impresa a mano. Este apéndice proporciona algunos detalles para crear a mano una gráfica de impresión. Incluso si usted nunca diseña salidas, podría ver gráficas impresas en la documentación de los programas existentes.

La figura E-1 muestra una gráfica de espaciado de impresora, que básicamente se ve como papel cuadriculado. La gráfica tiene muchos cuadros y en cada uno el diseñador coloca un carácter que se imprimirá. Las filas y las columnas por lo general están numeradas para referencia.

Figura E-1 Gráfica de espaciado de impresora

Por ejemplo, suponga que desea crear un informe impreso con las siguientes características:

- Un título impreso, INFORME DE INVENTARIO, que comienza a 11 espacios desde el borde izquierdo de la página y una línea abajo
- Encabezados de columna para NOMBRE DEL ARTÍCULO, PRECIO y CANTIDAD EN EXISTENCIA, dos líneas abajo del título y colocados sobre los elementos de datos reales que se están desplegando
- Datos variables que aparecen debajo de los encabezados de columna

El espaciado exacto y el uso de caracteres en mayúscula o minúscula en la gráfica impresa hacen una diferencia. Note que los datos constantes en la salida, los elementos que permanecen igual en cada ejecución del informe, no necesitan seguir las mismas reglas que los nombres variables en el programa. Dentro de un informe, las constantes como INFORME DE INVENTARIO y NOMBRE DEL ARTÍCULO pueden contener espacios. Estos encabezados ayudan a los lectores a entender la información que se presenta en el informe, no a que una computadora los interprete; no hay necesidad de correr juntos los nombres, como se hace cuando se eligen identificadores para las variables.

Un diseño de impresión en general muestra los datos variables que aparecerán en el informe. Por supuesto, es probable que los datos sean diferentes cada vez que el programa se ejecute. Por tanto, en lugar de escribir los nombres reales de los artículos y los precios, los usuarios y programadores con frecuencia usan X para representar los caracteres variables genéricos y 9 para representar los datos numéricos variables genéricos (algunos programadores usan X tanto para los datos de caracteres como para los numéricos). Cada línea que contenga X y 9 es una línea de detalle, o una línea que despliega los detalles de los datos. Las líneas de detalle por lo común aparecen muchas veces por página, en oposición a las líneas de encabezado que contienen el título y los encabezados de columna y en general aparecen sólo una vez por página.

Aun cuando un informe de inventario real podría contener al final cientos o miles de líneas de detalle, escribir dos o tres filas de X y 9 es suficiente para mostrar cómo aparecerán los datos. Por ejemplo, si un informe contiene los nombres de los empleados y sus salarios esos elementos de datos ocuparán las mismas posiciones impresas para dar salida línea por línea, ya sea que la salida contenga al final 10 empleados o 10,000. Algunas filas de X y 9 en posiciones idénticas son suficientes para establecer el patrón.

APÉNDICE



Dos variaciones de las estructuras básicas: case y do-while

Usted puede resolver cualquier problema lógico que encuentre usando sólo las tres estructuras: secuencia, selección y ciclo. Sin embargo, muchos lenguajes de programación permiten dos estructuras más: la estructura **case** y el ciclo **do-while**. Estas estructuras nunca se *necesitan* para resolver un problema; usted siempre puede usar una serie de selecciones en lugar de la estructura **case** y una secuencia más un ciclo **while** en lugar del ciclo **do-while**. Sin embargo, estas estructuras adicionales son convenientes en ocasiones. Los programadores consideran aceptables a todas las estructuras legales aceptables.

La estructura **case**

Puede usar la **estructura case** cuando hay varios valores distintos posibles para una sola variable y cada valor requiere una acción subsiguiente diferente. Suponga que trabaja en una escuela donde la colegiatura (*tuition*) varía por hora de crédito, dependiendo de si un estudiante es de primer ingreso, de segundo año, de tercero o de último. El diagrama de flujo y el pseudocódigo estructurados de la figura F-1 muestran una serie de decisiones que asignan diferentes valores a *tuition* dependiendo del valor de *year*.

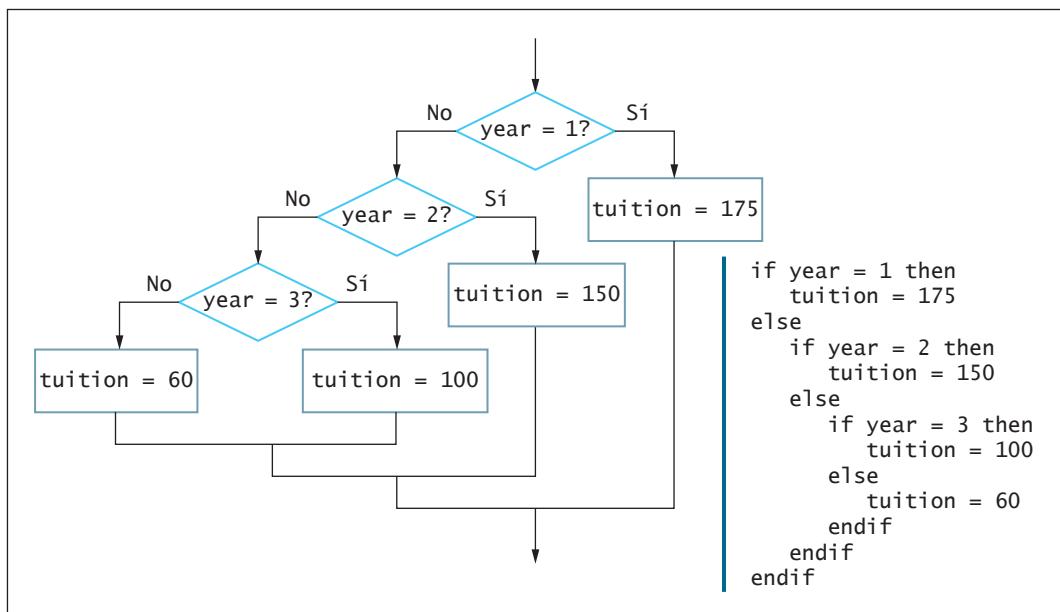


Figura F-1 Diagrama de flujo y seudocódigo de decisiones de colegiatura

La lógica que se muestra en la figura F-1 es correcta y completamente estructurada. La estructura de selección `year = 3?` está contenida en la estructura `year = 2?`, que se encuentra dentro de la estructura `year = 1?` (este ejemplo supone que si `year` no es 1, 2 o 3, el estudiante recibe la tarifa de colegiatura de último año).

Aun cuando los segmentos de programa en la figura F-1 son correctos y estructurados, muchos lenguajes de programación permiten usar una estructura `case`, como se muestra en la figura F-2. Cuando usted usa esta última compara una variable con una serie de valores de prueba, emprendiendo la acción apropiada cuando se encuentra una correspondencia. Muchas personas piensan que los programas que contienen la estructura `case` son más fáciles de leer que los que tienen una larga serie de decisiones, y la estructura `case` se permite porque los mismos resultados *podrían* lograrse con una serie de selecciones estructuradas (con lo que se estructura el programa). Es decir, si el primer programa está estructurado y el segundo refleja al primero punto por punto, entonces el segundo debe estar estructurado también.

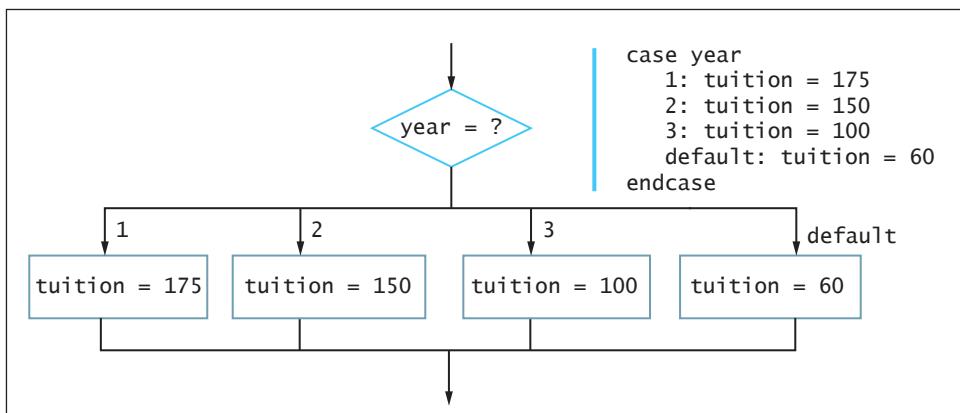


Figura F-2 Diagrama de flujo y seudocódigo de la estructura case que determina la colegiatura



El término *default* en la figura F-2 significa *si ninguno de los otros casos es verdadero*. Varios lenguajes de programación usan diferentes sintaxis para el caso *default*.

Usted usa la estructura *case* sólo cuando una serie de decisiones se basa en diferentes valores almacenados en una sola variable. Si se prueban múltiples variables, entonces debe usar una serie de decisiones.

Aun cuando un lenguaje de programación le permite usar la estructura *case*, debe entender que esta última es sólo una conveniencia que podría hacer que un diagrama de flujo, seudocódigo o código de programa real sean más fáciles de entender a primera vista. Cuando escribe una serie de decisiones usando la estructura *case*, la computadora aún toma una serie de decisiones individuales, del mismo modo que si hubiera usado muchas combinaciones *if-then-else*. En otras palabras, podría preferir ver el diagrama de la figura F-2 para entender las cuotas de colegiatura cobradas por una escuela, pero una computadora en realidad toma las decisiones como se muestra en la figura F-1, una a la vez. Cuando escribe sus propios programas, siempre es aceptable expresar un proceso de toma de decisiones complicado como una serie de selecciones individuales.

El ciclo do-while

Recuerde que un ciclo estructurado (al que también se llama ciclo *while*) se ve como en la figura F-3. Un ciclo de caso especial llamado *do-while* es como se observa en la figura F-4.

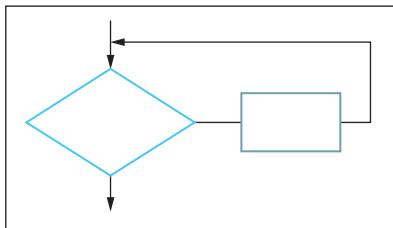


Figura F-3 El ciclo `while`, que es un ciclo preprueba

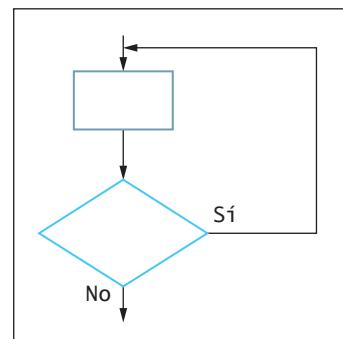


Figura F-4 Estructura de un ciclo `do-while`, que es un ciclo posprueba

Existe una diferencia importante entre estas dos estructuras. En un ciclo `while` usted hace una pregunta y, dependiendo de la respuesta, podría entrar o no en el ciclo para ejecutar su procedimiento. A la inversa, en un **ciclo do-while**, usted asegura que el procedimiento se ejecute al menos una vez; luego, dependiendo de la respuesta a la pregunta controladora, el ciclo puede ejecutarse o no en ocasiones adicionales.

Observe que la palabra *do* comienza el nombre del ciclo `do-while`. Esto debería recordarle que la acción que usted “hace” precede a probar la condición.

En un ciclo `while`, la pregunta que controla un ciclo se da al principio, o “encima”, del cuerpo del mismo. Un ciclo `while` es uno preprueba porque una condición se prueba antes de entrar en el ciclo de una sola vez. En un ciclo `do-while`, la pregunta que lo controla se da al final, o en la “parte inferior”, del cuerpo del ciclo. Los ciclos `do-while` son posprueba porque una condición se prueba después de que se ha ejecutado el cuerpo del mismo.

Se encuentran ejemplos de ciclos `do-while` todos los días. Por ejemplo:

do

 pagar una factura

while queden más facturas por pagar

Como otro ejemplo:

do

 lavar un plato

while queden más platos por lavar

En estos ejemplos, la actividad (pagar facturas o lavar platos) debe ocurrir al menos una vez. Con un ciclo `do-while`, usted hace la pregunta que determina si continuará sólo después de que la actividad se ha ejecutado al menos una vez.

Nunca se requiere que use un ciclo posprueba; usted puede duplicar la misma serie de acciones creando una secuencia seguida por un ciclo `while` preprueba estándar. Considere los diagramas de flujo y seudocódigo en la figura F-5.

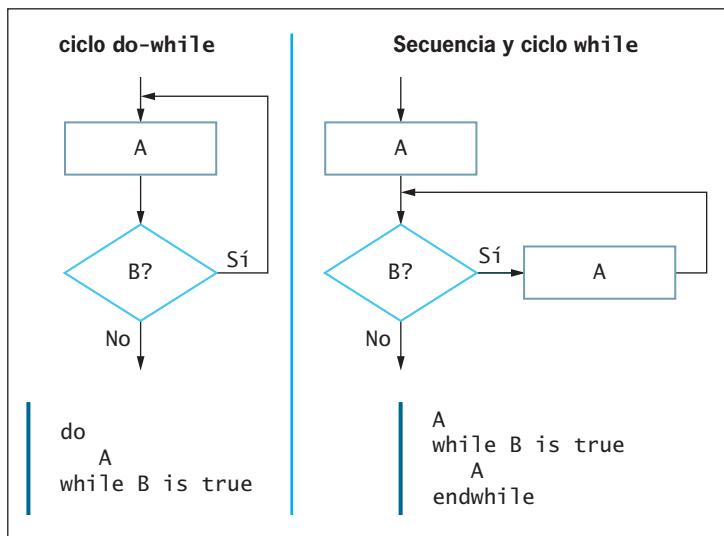


Figura F-5 Diagrama de flujo y seudocódigo para un ciclo do-while y un ciclo while que hace lo mismo

En el lado izquierdo de la figura F-5, A se ejecuta, y luego se pregunta B. Si B es sí, entonces se ejecuta A y B se pregunta de nuevo. En el lado derecho de la figura, A se ejecuta y luego se pregunta B. Si B es sí, entonces A se ejecuta y B se pregunta de nuevo. En otras palabras, ambos conjuntos de segmentos de diagramas de flujo y seudocódigos hacen exactamente lo mismo.

Debido a que los programadores entienden que cualquier ciclo posprueba (do-while) puede expresarse con una secuencia seguida de un ciclo while, la mayoría de los lenguajes permiten al menos una versión del ciclo posprueba por conveniencia.

Reconocimiento de las características compartidas por todos los ciclos estructurados

Cuando examine las figuras F-3 y F-4 notará que, en el ciclo while, la pregunta que controla el ciclo se coloca al principio de los pasos que se repiten. En el ciclo do-while, la pregunta que lo controla se coloca al final de la secuencia de pasos que se repiten.

Todos los ciclos estructurados, tanto de preprueba como de posprueba, comparten estas dos características:

- La pregunta que controla el ciclo debe proporcionar ya sea una entrada o una salida de la estructura que se repite.
- La pregunta que controla el ciclo proporciona la *única* entrada o salida de la estructura que se repite.

En otras palabras, hay exactamente un valor que controla el ciclo y proporciona ya sea la única entrada o la única salida del mismo.



Algunos lenguajes soportan un ciclo **do-until**, que es posprueba e itera hasta que la pregunta que controla el ciclo es falsa. El ciclo **do-until** sigue las reglas de uno estructurado.

Reconocimiento de ciclos no estructurados

La figura F-6 muestra un ciclo no estructurado. No es **while**, que comienza con una decisión y después de una acción regresa a la decisión. Tampoco es **do-while**, que inicia con una acción y termina con una decisión que podría repetir la acción. En cambio, comienza como un ciclo posprueba (uno **do-while**), con un proceso seguido por una decisión, pero una rama de la decisión no repite el proceso inicial. En cambio, ejecuta una acción nueva adicional antes de repetir el proceso inicial.

Si usted necesita usar la lógica que se muestra en la figura F-6, ejecutar una tarea, hacer una pregunta y quizá ejecutar una tarea adicional antes de regresar al primer proceso, entonces la forma de efectuar la lógica estructurada es repetir el proceso inicial dentro del ciclo al final del mismo. La figura F-7 muestra la misma lógica que la F-6, pero ahora es estructurada, con una secuencia de dos acciones que ocurren dentro del ciclo.

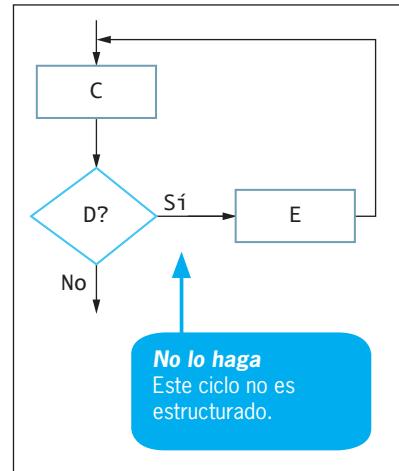


Figura F-6 Ciclo no estructurado

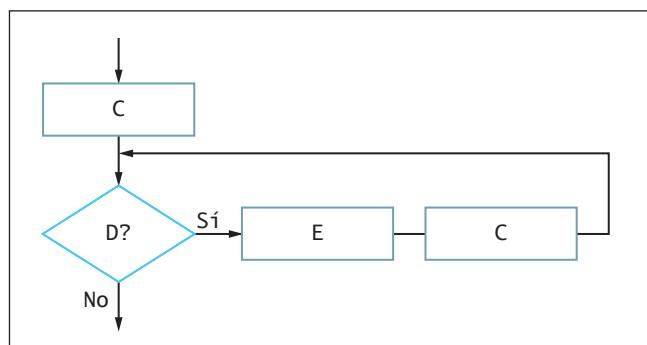


Figura F-7 Secuencia y ciclo estructurado que lleva a cabo las mismas tareas que la figura F-6



De manera especial cuando usted domina la lógica estructurada por primera vez, quizás prefiera usar sólo las tres estructuras básicas: secuencia, selección y ciclo **while**. Es posible resolver todos los problemas lógicos usando las y usted puede entender todos los ejemplos en este libro si utiliza sólo estas estructuras.

Términos clave

Una **estructura case** prueba una sola variable contra múltiples valores, proporcionando acciones separadas para cada ruta lógica.

336

Un **ciclo do-while** es de posprueba; el cuerpo se ejecuta antes de que se pruebe la variable de control del ciclo.

Un **ciclo do-until** es de posprueba; itera hasta que la pregunta que controla el ciclo es falsa.

Glosario

A

abrir un archivo Proceso de localizar un archivo en un dispositivo de almacenamiento, preparándolo físicamente para ser leído, y asociarlo con un identificador dentro de un programa.

abstracción El proceso de poner atención a las propiedades importantes mientras se ignoran los detalles no esenciales.

actualizar un archivo maestro Modificar los valores en un archivo maestro con base en registros de transacción.

acumulador Variable que se usa para recopilar o acumular valores.

algoritmo Secuencia de pasos necesarios para resolver cualquier problema.

ambiente de desarrollo integrado (IDE) Paquete de software que proporciona un editor, compilador y otras herramientas de programación.

American Standard Code for Information Interchange (ASCII: Código Estándar Estadounidense para el Intercambio de Información) Esquema de codificación de caracteres de ocho bits que se usa en muchas computadoras personales.

anidar estructuras Colocar una estructura dentro de otra.

apilar estructuras Unir las estructuras de un programa de un extremo a otro.

archivo Grupo de registros que van juntos por alguna razón lógica.

archivo de computadora Colección de datos almacenados en un dispositivo no volátil en un sistema de cómputo.

archivo de respaldo Copia que se conserva en caso de que los valores necesiten restaurarse a su estado original.

archivo de transacción Aquel que contiene datos temporales que se usan para actualizar un archivo maestro.

archivo hijo Copia de un archivo después de su revisión.

archivo maestro Aquel que contiene datos completos y relativamente permanentes.

archivo padre Copia de un archivo antes de su revisión.

archivo secuencial Archivo en el que los registros se almacenan uno detrás de otro en algún orden.

archivos binarios Aquellos que contienen datos que no han sido codificados como texto.

archivos de acceso aleatorio Aquellos que contienen registros que pueden localizarse en cualquier orden.

archivos de acceso directo Aquellos de acceso aleatorio.

archivos de acceso instantáneo Aquellos de acceso aleatorio en los que debe tenerse acceso a los registros de inmediato.

archivos de texto Aquellos que contienen datos que pueden leerse en un editor de texto.

arreglo Serie o lista de variables en la memoria de una computadora, todas ellas tienen el mismo nombre pero se distinguen con subíndices.

arreglos paralelos Dos o más arreglos en los que cada elemento en uno de ellos está asociado con el elemento en la misma posición relativa en el otro o los otros.

asociatividad de izquierda a derecha Describe operadores que evalúan primero la expresión a la izquierda.

asociatividad derecha y asociatividad de derecha a izquierda Describe operadores que evalúan primero la expresión de la derecha.

autodocumentación Describe un programa que contiene datos y nombres de módulos significativos que describen el propósito del programa.

B

bandera Variable establecida para indicar si ha ocurrido algún evento.

base 2 Describe los números creados usando el sistema de numeración binario.

base 10 Describe los números creados usando el sistema de numeración decimal.

base 16 Describe los números creados usando el sistema de numeración hexadecimal.

base de datos Contenedor lógico que contiene un grupo de archivos, con frecuencia llamados tablas, que juntos atienden las necesidades de información de una organización.

basura Describe el valor desconocido almacenado en una variable no asignada.

bit Dígito binario; unidad de almacenamiento igual a un octavo de byte.

bloque Grupo de declaraciones que se ejecutan como una sola unidad.

búsqueda binaria Aquella que empieza en medio de una lista ordenada y luego determina si debería continuar hacia arriba o hacia abajo para encontrar un valor objetivo.

búsqueda lineal Búsqueda a lo largo de una lista desde un extremo hasta el otro.

byte Una unidad de almacenamiento en computadora; puede contener cualquiera de 256 combinaciones de 0 y 1 que a menudo representan un carácter.

C

cadena Describe datos que no son numéricos.

caja de Pascal Convención para nombrar variables en la que la letra inicial está en mayúsculas, los nombres de variables con múltiples palabras se escriben juntos y cada palabra nueva dentro del nombre de la variable comienza con una letra mayúscula.

campo Elemento de datos individual, como lastName, streetAddress o annualSalary.

campo de control de interrupciones Variable que contiene el valor que señala una interrupción de procesamiento especial en un programa.

carácter Una letra, número o símbolo especial como A, 7 o \$.

carga adicional Todos los recursos y tiempos requeridos por una operación.

carpetas Unidades de organización en dispositivos de almacenamiento; cada una puede contener múltiples archivos al igual

que carpetas adicionales. Las carpetas son directorios gráficos.

caso nulo Rama de una decisión en la que no se emprende ninguna acción.

cerrar un archivo Acción que hace que un archivo ya no esté disponible para una aplicación.

ciclo Estructura que repite acciones mientras continúa una condición.

ciclo anidado Estructura de ciclo dentro de otra; son ciclos dentro de ciclos.

ciclo de desarrollo del programa Pasos que ocurren durante la vida de un programa, entre los que se encuentran: planeación, codificación, traducción, prueba, producción y mantenimiento del programa.

ciclo definido Aquel para el cual el número de repeticiones es un valor predeterminado.

ciclo do-until Ciclo después de la prueba que itera hasta que la pregunta que lo controla es falsa.

ciclo do-while Ciclo después de la prueba en el que el cuerpo se ejecuta antes que se pruebe la variable que lo controla.

ciclo externo Ciclo que contiene un ciclo anidado.

ciclo indefinido Aquel para el que no es posible predecir el número de ejecuciones cuando se escribe el programa.

ciclo infinito Flujo de lógica que se repite sin fin.

ciclo interno Cuando los ciclos están anidados, el ciclo que está contenido dentro del otro.

ciclo posprueba Ciclo que prueba su variable de control del ciclo después de cada iteración, lo que significa que el cuerpo del mismo se ejecuta al menos una vez.

ciclo preprueba Ciclo que prueba su variable de control del ciclo antes de cada

iteración, lo que significa que el cuerpo del mismo podría no ejecutarse nunca.

ciclo while o ciclo while...do Un ciclo en el que un proceso continúa mientras alguna condición sigue siendo verdadera.

clasificación Proceso de colocar registros en orden por el valor en un campo o campos específicos.

cláusula else Parte de una decisión que contiene la acción o acciones que se ejecutan sólo cuando la expresión booleana en la decisión es falsa.

cláusula if-then Parte de una decisión que mantiene la acción resultante cuando la expresión booleana en la decisión es verdadera.

codificar del programa Acto de escribir las declaraciones de un programa en un lenguaje de programación.

código de programa El conjunto de instrucciones que escribe un programador en un lenguaje de programación.

código espagueti Lógica de programa enredada y no estructurada.

código fuente Declaraciones legibles de un programa, escritas en un lenguaje de programación.

código objeto Aquel que se ha traducido a lenguaje de máquina.

comentario del programa Declaración no ejecutable que colocan los programadores dentro del código para explicar las declaraciones del programa en inglés. *Véase también* documentación interna.

compilador Software que traduce un lenguaje de alto nivel en lenguaje de máquina e identifica los errores de sintaxis. Un compilador es parecido a un intérprete; sin embargo, traduce todas las declaraciones en un programa antes de ejecutarlas.

comprobación de rango Comparación de una variable con una serie de valores

que marcan los extremos limitantes de los rangos.

condición compuesta Aquella construida cuando se requieren múltiples decisiones antes de determinar un resultado.

confiabilidad Característica de los programas modulares que aseguran que se ha probado un módulo y se ha demostrado que funciona en forma correcta.

constante de cadena (o constante de cadena literal) Grupo específico de caracteres encerrados dentro de comillas.

constante literal Valor numérico o de cadena literal.

constante nombrada Ubicación de memoria nombrada, similar a una variable, con excepción de que su valor nunca cambia durante la ejecución de un programa. De manera convencional, las constantes se nombran usando sólo letras mayúsculas.

constante numérica Valor numérico específico.

contador Cualquier variable numérica que se usa para contar el número de veces que un evento ha ocurrido.

contenedor Uno de una clase de objetos cuyo propósito principal es contener otros elementos, por ejemplo, una ventana.

control de interrupciones Desviación temporal en la lógica de un programa para el procesamiento de un grupo especial.

control de interrupciones de nivel único

Interrupción en la lógica de un programa con base en el valor de una sola variable.

conversión Conjunto de acciones que una organización debe emprender para cambiar al uso de un programa o sistema nuevo.

correr Acción de una computadora cuando lleva a cabo las declaraciones en un programa compilado o interpretado. También se le llama *ejecutar*.

cuerpo de ciclo Conjunto de acciones que ocurren dentro de un ciclo.

cuerpo del módulo Parte de un módulo que contiene todas sus instrucciones.

D

decisión AND Aquella en la que dos condiciones deben ser verdaderas para que tenga lugar una acción.

decisión anidada Decisión dentro de la cláusula *if-then* o *else* de otra decisión; también llamada *if anidada*.

decisión binaria Decisión sí o no; se le llama así porque hay dos resultados posibles.

decisión OR Aquella que contiene dos o más condiciones; si se cumple al menos una condición, la acción resultante tiene lugar.

declaración Exposición que nombra una variable y su tipo de datos.

declaración de asignación Aquella que almacena el resultado de cualquier cálculo ejecutado en su lado derecho en la ubicación nombrada en su lado izquierdo.

declaración de variables El proceso de nombrar variables del programa y asignarles un tipo.

declaración fin de estructura Aquella que designa el final de una estructura de seudocódigo.

declaración for Declaración que puede usarse para codificar ciclos definidos; también llamada *ciclo for*. La declaración contiene una variable de control de ciclo que se inicializa, evalúa y altera en forma automática.

declaración if en cascada Serie de declaraciones *if anidadas*.

declaración return del módulo Parte de un módulo que marca su fin e identifica el punto en el que el control regresa al programa o módulo que lo llamó.

decrementar Cambiar una variable al disminuirla por un valor constante, con frecuencia 1.

depuración Proceso de hallar y corregir errores en el programa.

diagrama de flujo Representación gráfica de los pasos lógicos que se requieren para resolver un problema.

diccionario de datos Lista de todos los nombres de variables que se usan en un programa, junto con su tipo, tamaño y descripción.

directorios Unidades de organización en dispositivos de almacenamiento; cada uno puede contener múltiples archivos al igual que directorios adicionales. En un sistema de interfaz gráfica, los directorios con frecuencia se llaman *carpetas*.

dispositivo de almacenamiento Aparato de hardware que contiene información para su recuperación posterior.

dispositivos de almacenamiento permanente Dispositivos de hardware que contienen datos no volátiles; algunos ejemplos incluyen discos duros, DVD, discos Zip, unidades USB y carretes de cinta magnética.

dispositivos de entrada y salida por defecto Dispositivos de hardware que no requieren abrirse; por lo general son el teclado y el monitor, respectivamente.

documentación Todo el material de apoyo que va con un programa.

documentación externa Todo el material externo que elaboran los programadores para respaldar un programa; contrasta con los *comentarios de programa*, que son la documentación interna del mismo.

documentación interna Aquella que se encuentra dentro de un programa. Véase *también* comentario del programa.

E

editor de texto Programa que se usa para crear archivos de texto simples; se parece a un procesador de palabras, pero sin tantas características.

ejecutar Hacer que una computadora use un programa escrito y compilado; también se le llama *correr (running)*.

elemento Variable individual de un arreglo.

en ámbito Característica de las variables y constantes declaradas dentro de un método que se aplican sólo dentro de ese método.

encabezado del módulo Parte de un módulo que incluye su identificador y quizá otra información identificadora necesaria.

encapsulamiento Acto de contener las instrucciones y datos de una tarea en el mismo método.

entero Un número entero.

entrada Describe el ingreso de elementos de datos en la memoria de la computadora usando dispositivos de hardware como teclados y ratones.

entrada anticipada o lectura anticipada Declaración que lee la primera entrada en el registro de datos antes de empezar un ciclo estructurado.

entrada con eco Acto de repetir la entrada de vuelta al usuario ya sea en un indicador subsiguiente o en una salida.

eof Marcador de final de datos en un archivo, abreviatura en inglés de *fin de archivo (end of file)*.

error de sintaxis Error en el lenguaje o la gramática.

error lógico El que ocurre cuando se ejecutan instrucciones incorrectas o cuando se ejecutan en un orden erróneo.

error semántico Aquel que ocurre cuando se usa una palabra correcta en un contexto incorrecto.

escribir en un archivo Acto de copiar datos de la RAM a un almacenamiento persistente.

estructura Unidad básica de lógica de programación; cada estructura es una secuencia, selección o ciclo.

estructura case Aquella que evalúa una sola variable contra múltiples valores, proporcionando acciones separadas para cada ruta lógica.

estructura de ciclo Aquella que repite acciones mientras una condición de prueba se mantiene verdadera.

estructura de decisión Estructura de programa en la que se hace una pregunta y, dependiendo de la respuesta, se emprende uno de dos cursos de acción. Luego, sin importar cuál ruta se siga, las rutas se unen y se ejecuta la siguiente tarea.

estructura de secuencia Estructura de programa que contiene pasos que se ejecutan en orden. Una secuencia puede contener cualquier cantidad de tareas, pero no hay posibilidad de desviarse y de saltarse cualquiera de las tareas.

estructura de selección Estructura de programa que contiene una pregunta y toma uno de dos cursos de acción dependiendo de la respuesta. Entonces, sin importar cuál ruta se siga, la lógica del programa continúa con la siguiente tarea.

evaluación de cortocircuito Característica lógica en la que las expresiones en cada parte de una expresión más grande se evalúan sólo en tanto es necesario para determinar el resultado final.

expresión booleana Aquella que representa sólo uno de dos estados, que se expresan por lo general como verdadero o falso.

expresión trivial Aquella que siempre evalúa al mismo valor.

Extended Binary Coded Decimal Interchange Code (EBCDIC: Código ampliado de intercambio decimal codificado en binario) Esquema de codificación de caracteres de ocho bits que se usa en muchas computadoras grandes.

F

fuerza de límites Término que describe el subíndice de una matriz que no está dentro del rango de subíndices aceptables.

G

gigabyte Mil millones de bytes.

GIGO Acrónimo en inglés para *garbage in, garbage out (entra basura, sale basura)*; significa que si la entrada es incorrecta, la salida no tiene valor.

global Describe variables que se conocen en todo el programa.

gráfica de jerarquía Diagrama que ilustra las relaciones de los módulos entre sí.

gráfica IPO Herramienta de desarrollo de programas que define las tareas de entrada, procesamiento y salida.

gráfica TOE Herramienta de desarrollo de programas que enlista tareas, objetos y eventos.

H

hacer declaraciones Proceso de nombrar las variables del programa y asignarles un tipo.

hardware El equipo de un sistema de cómputo.

I

IDE Acrónimo en inglés de Ambiente de Desarrollo Integrado (*Integrated Development Environment*), que es el ambiente de

desarrollo visual en algunos lenguajes de programación.

identificador Nombre del componente de un programa.

if-then Estructura similar a if-then-else, pero no es necesaria una acción alternativa o “else”.

incrementar Cambiar una variable agregándole un valor constante, con frecuencia 1.

indicador Mensaje que se muestra en el monitor pidiendo al usuario una respuesta.

información Datos procesados.

informe de control de interrupciones

Informe que enlista elementos en grupos. Con frecuencia, cada grupo es seguido por un subtotal.

informe sumario Aquel que sólo enumera totales, sin registros individuales detallados.

inicializar una variable Acto de asignar el primer valor a una variable, a menudo al mismo tiempo en que ésta se crea.

interfaz gráfica de usuario (GUI) Una interfaz de programa que usa pantallas para mostrar la salida del mismo y permite a los usuarios interactuar con éste en un ambiente gráfico.

iteración Otro nombre para una estructura de ciclo.

J

jerarquía de datos Representa la relación de bases de datos, archivos, registros, campos y caracteres.

K

kilobyte Aproximadamente 1000 bytes.

L

leer de un archivo Acto de copiar a la RAM los datos de un archivo que

se encuentra en un dispositivo de almacenamiento.

lenguaje binario Lenguaje de computación que se representa usando una serie de 0 y 1.

lenguaje de bajo nivel Lenguaje de programación que no está muy alejado del lenguaje de máquina, en contraposición con uno de alto nivel.

lenguaje de máquina Lenguaje de circuitería encendido/apagado de una computadora; de bajo nivel formado por 1 y 0 que la máquina entiende.

lenguaje de programación Lenguaje como Visual Basic, C#, C++, Java o COBOL, que se usa para escribir programas.

lenguaje de programación de alto nivel Lenguaje de programación en inglés, en contraposición con uno de bajo nivel.

lenguaje de programación interpretado Lenguaje como Python, Lua, Perl o PHP que se usa para escribir programas que se mecanografián directamente desde un teclado y se almacenan como texto y no como archivos ejecutables binarios. También se les llama *lenguajes de programación de scripting* o *lenguajes de script*.

línea de comandos La ubicación en una pantalla de computadora donde se mecanografián entradas para comunicarse con el sistema operativo de la máquina.

línea de flujo Flecha que conecta los pasos en un diagrama de flujo.

llamada a módulo Usar el nombre de un módulo para invocarlo, causando que se ejecute.

lógica Instrucciones que se dan a la computadora en una secuencia específica, sin omitir ninguna ni agregar superfluas.

lógica de línea principal Lógica general del programa principal de principio a fin.

lvalue Identificador de la dirección de memoria a la izquierda de un operador de asignación.

M

mantenimiento Todas las mejoras y correcciones hechas a un programa después de que está en producción.

megabyte Un millón de bytes.

memoria de acceso aleatorio (RAM) Almacenamiento interno temporal de la computadora.

memoria de computadora Almacenamiento interno temporal dentro de una computadora.

Microsoft Visual Studio IDE Paquete de software que contiene herramientas útiles para crear programas en Visual Basic, C++ y C#.

modularización Proceso de dividir un programa en módulos.

módulo Unidad pequeña de un programa que se usa con otros módulos para hacer un programa. Los programadores también se refieren a ellos como subrutinas, procedimientos, funciones y métodos.

N

nibble Medida de almacenamiento igual a cuatro bits o medio byte.

nivel del programa Nivel en el que se declaran las variables globales.

no volátil Término que describe el almacenamiento cuyo contenido se conserva cuando se pierde la energía.

notación de camello Convención para nombrar las variables en la que la letra inicial está en minúsculas, los nombres de variables con múltiples palabras se escriben juntos y cada palabra nueva dentro del nombre de la variable comienza con una letra mayúscula.

notación húngara Convención para la denominación de variables en la que un tipo de datos de una variable u otra información se almacena como parte de su nombre.

numérico Término que describe datos que consisten en números.

número mágico Constante numérica no identificada.

números reales Números de punto flotante.

O

operador condicional AND Símbolo que se usa para combinar decisiones de modo que dos o más condiciones deben ser verdaderas para que ocurra una acción. También se le llama *operador AND*.

operador condicional OR Símbolo que se usa para combinar decisiones cuando cualquier condición puede ser verdadera para que ocurra una acción. También se le llama *operador OR*.

operador de asignación Signo de igual; siempre requiere el nombre de una ubicación de memoria en su lado izquierdo.

operador de comparación relacional

Símbolo que expresa comparaciones lógicas (booleanas). Algunos ejemplos incluyen =, >, <, >=, <= y <>.

operador lógico NOT Símbolo que invierte el significado de una expresión booleana.

operador binario Aquel que requiere dos operandos, uno en cada lado.

orden ascendente Describe el ordenamiento de registros de menor a mayor, con base en un valor dentro de un campo.

orden de operaciones Describe las reglas de precedencia.

orden descendente Describe la ordenación de registros de mayor a menor, con base en un valor dentro de un campo.

P

palabras clave Conjunto limitado de palabras que se reserva en un lenguaje.

pila Ubicación de memoria en la que la computadora sigue la pista de la dirección de memoria correcta a la que debería regresar después de ejecutar un módulo.

poblar un arreglo Asignar valores a los elementos de un arreglo.

polimorfismo La capacidad de un método de actuar de manera apropiada dependiendo del contexto.

portátil Término que describe un módulo que puede reutilizarse con más facilidad en múltiples programas.

precedencia Cualidad de una operación que determina el orden en el que se evalúa.

procesamiento Organizar elementos de datos, comprobar su precisión o ejecutar operaciones matemáticas en ellos.

procesamiento por lotes Aquel que ejecuta las mismas tareas con muchos registros en secuencia.

programa de control de interrupciones

Aquel en el que un cambio en el valor de una variable inicia acciones especiales o causa que ocurra un procesamiento especial o extraño.

programa interactivo Aquel en el que un usuario hace solicitudes directas, en contraposición a otro en el que la entrada proviene de un archivo.

programa principal Aquel que se ejecuta de principio a fin y llama a otros módulos; también se le conoce como *método de programa principal*.

programación defensiva Técnica en la que los programadores tratan de prepararse para todos los posibles errores antes de que ocurran.

programación orientada hacia los objetos Técnica de programación que se enfoca en los objetos, o “cosas”, y describe sus atributos y comportamientos.

programación procedural Técnica de programación que se enfoca en los procedimientos que crean los programadores.

programación sin goto Nombre para describir la programación estructurada, porque los programadores estructurados no usan una declaración “go to”.

programar Acto de desarrollar y escribir programas.

programas Conjuntos de instrucciones para una computadora.

programas estructurados Aquellos que siguen las reglas de la lógica estructurada.

programas no estructurados Aquellos que *no* siguen las reglas de la lógica estructurada.

prueba de escritorio Proceso de recorrer una solución de programa en papel.

R

registro Grupo de campos que se encuentran almacenados juntos como una unidad debido a que contienen datos sobre una entidad única.

relación indirecta Describe la relación entre arreglos paralelos en los que un elemento en el primer arreglo no tiene acceso directo a su valor correspondiente en el segundo.

repetición Otro nombre para una estructura de ciclo.

reutilización Característica de los programas modulares que permite que se usen módulos individuales en una variedad de aplicaciones.

ruta Combinación de la unidad de disco de un archivo y la jerarquía completa de directorios en los que reside el archivo.

ruta sin salida Una ruta lógica que nunca puede ser recorrida.

S

salida Describe la operación de recuperar información de la memoria y enviarla a un dispositivo, como un monitor o una impresora, de modo que las personas puedan ver, interpretar y trabajar con los resultados.

seguridad de tipo Característica de los lenguajes de programación que impide asignar valores de un tipo de datos incorrecto.

selección de alternativa dual o alternativa dual if Estructura de selección que define una acción que se ejecutará cuando la condición probada es verdadera, y otra acción que se ejecutará cuando sea falsa.

selección if de alternativa única o selección de alternativa única Estructura de selección en la que se requiere una acción sólo para una rama de la decisión. Esta forma de la estructura de selección también se llama *if-then*, porque no se necesita una acción “else”.

seudocódigo Representación en inglés de los pasos lógicos que se requieren para resolver un problema.

símbolo de anotación Símbolo de los diagramas de flujo que contiene información que amplía lo que aparece en otro símbolo de dichos diagramas; con frecuencia se representa con un cuadro de tres lados y está conectado con el paso al que hace referencia por medio de una línea punteada.

símbolo de decisión Aquel que representa una decisión en un diagrama de flujo; tiene forma de diamante.

símbolo de entrada Símbolo que indica una operación de entrada y se representa en los diagramas de flujo como un paralelogramo.

símbolo de entrada/salida

Paralelogramo en los diagramas de flujo.

símbolo de procesamiento Aquel que se representa como un rectángulo en los diagramas de flujo.

símbolo de salida Aquel que indica una operación de salida y se representa como un paralelogramo en los diagramas de flujo.

símbolo I/O Símbolo de entrada/salida.

símbolo terminal Aquel que se usa en cada extremo de un diagrama de flujo; su forma es una pastilla. También se le llama *símbolo inicio/fin*.

sintaxis Reglas de un lenguaje.

sistema de cómputo Combinación de todos los componentes requeridos para procesar y almacenar datos usando una computadora.

sistema de numeración binario Sistema de numeración basado en dos dígitos; los valores de columna son múltiplos de 2.

sistema de numeración decimal El sistema de numeración basado en 10 dígitos; los valores de columna son múltiplos de 10.

sistema de numeración hexadecimal Sistema de numeración basado en 16 dígitos; los valores de las columnas son múltiplos de 16.

software Programas que dicen a la computadora qué hacer.

software de aplicación Programas que realizan una tarea para el usuario.

software de sistema Programas que manejan las operaciones de la computadora.

stub Método sin declaraciones que se usa como marcador.

subíndice Número que indica la posición de un elemento particular dentro de un arreglo.

T

tabla de verdad Diagrama que se usa en matemáticas y lógica para describir la verdad de una expresión completa con base en la verdad de sus partes.

tamaño del arreglo Número de elementos que puede contener un arreglo.

tarea de fin de trabajo Paso al final de un programa para terminar la aplicación.

tareas de administración Aquellas que deben ejecutarse al comienzo de un programa para prepararse para el resto del mismo.

tareas de ciclo detallado Los pasos que se repiten para cada conjunto de datos de entrada.

tiempo real Término que describe aplicaciones que requieren acceso inmediato a un registro mientras un cliente espera.

tipo de datos La característica de una variable que describe la clase de valores que puede contener una variable y los tipos de operaciones que pueden ejecutarse con ella.

tomar una decisión Probar un valor para determinar una ruta lógica.

U

Unicode Esquema de codificación de caracteres de 16 bits.

unidad central de procesamiento (CPU) Pieza de hardware que procesa datos.

unión de archivos Acto de combinar dos o más archivos mientras se mantiene el orden secuencial.

usuarios (o usuarios finales) Personas que trabajan con los programas de computadora y se benefician de ellos.

V

validación de datos Asegurar que los datos queden dentro de un rango aceptable.

valor centinela Valor que representa un punto de entrada o salida.

valor comodín Valor preseleccionado que detiene la ejecución de un programa.

valor de paso Número que se usa para incrementar una variable de control de ciclo en cada paso a lo largo de un ciclo.

valor de punto flotante Variable numérica fraccionaria que contiene un punto decimal.

valores alfanuméricos Conjunto de valores que incluyen caracteres alfabéticos, números y puntuación.

variable Ubicación de memoria nombrada de un tipo de datos específico, cuyo contenido puede variar o diferir con el tiempo.

variable de cadena Aquella que puede contener texto que incluya letras, dígitos y caracteres especiales como signos de puntuación.

variable de control de ciclo Aquella que determina si un ciclo continuará.

variable numérica Aquella que contiene valores numéricos.

variable temporal Variable activa que contiene resultados intermedios durante la ejecución de un programa.

visible Característica de los elementos de datos que significa que “pueden verse” sólo dentro del método en el cual se declaran.

volátil Característica de la memoria interna en la cual su contenido se pierde cada vez que la computadora se apaga.

Índice

Nota: Los números de página en **negritas** indican dónde se definen los términos clave.

Caracteres especiales

- > (operador mayor que), 126
- < (operador menor que), 126
- () (paréntesis), 52
- <> (operador no igual a), 126
- * (asterisco), 45
- + (signo de suma), 45
- (signo de resta), 47
- / (diagonal), 47
- = (signo igual), 47, 126
- >= (operador mayor o igual que), 126
- <= (operador menor o igual que), 126

A

- Abrir un archivo, **262**
- Abstracción, **49**
 - modularización, 49-50
- Actualizar un archivo maestro, **281**
- Acumuladores, **195**, 195-198
- Agotamiento IPV4, 13
- Algoritmos, **9**
- Almacenamiento
 - de datos en arreglos, 214-216
 - dispositivos de, **3**
 - medición del, 312-314
 - no volátil, **3**
 - volátil, **3**
- Ambiente de desarrollo integrado (IDE; integrated development environment), **23**, 23-24
- American Standard Code for Information Interchange (ASCII; Código Estándar Estadounidense para el Intercambio de Información), **306**, 307-310
- Anotación, símbolos de, **64**, 64-65
- Archivo(s), **260**. Véase también archivos de computadora

- binarios, **258**
- de acceso aleatorio, **290**, 290-291
- de acceso directo, **290**
- de acceso instantáneo, **290**
- de respaldo, **266**
- de transacción, **281**
- hijo, **266**, 282
- padre, **266**, 282
- Archivo(s) de computadora, **258**, 258-260
 - abrir, 262
 - archivos binarios, 258
 - archivos de texto, 258
 - cerrar, 264
 - de acceso aleatorio, 290-291
 - de respaldo, 266
 - declarar, 261-262
 - escribir datos, 264
 - hijo, 266, 282
 - leer datos, 262-264
 - organización de, 259
 - padre, 266, 282
 - procesamiento de archivos maestros y de transacción, 281-289
 - programa que ejecuta operaciones de archivo, 264-267
 - secuenciales. Véase archivos secuenciales
 - unión de, 273
- Archivo(s) maestro, **281**
 - actualiza el, 281
- Archivo(s) secuenciales, **267**, 267-281
 - lógica de control de interrupción, 268-272
 - unión de, 273-281
- Arreglo(s), **213-245**, **214**
 - almacenamiento de datos en, 214-216
 - búsqueda para una correspondencia de rango, 237-241

- búsqueda para una correspondencia exacta, 226-229
- constantes con, 224-225
- elementos del, 214-215
- paralelos, 230, 230-237
- permanencia dentro de los límites de los, 241-243
- poblar los, 215
- reemplazar decisiones anidadas, 216-224
- tamaño del, 214
- uso de un ciclo **for** para procesar, 244-245
- ASCII (American Standard Code for Information Interchange; Código Estándar Estadounidense para el Intercambio de Información), 306, 307-310
- Asignación
- declaraciones de, 42
 - operador de (=), 47
 - operador de, 42
- Asociatividad
- de derecha a izquierda, 42, 42-43
 - de izquierda a derecha, 47
 - derecha, 42
- Asterisco (*), operador de multiplicación, 47
- B**
- Base 2, 305
- Base 10, 305
- Base 16, 311
- Basura, 40
- Bit, 306
- Búsqueda
- arreglos paralelos, 230-237
 - lineal, 226
 - mejora de la eficiencia de la, 234-236
- Bytes, 258, 306
- C**
- Cadena
- constante de, 38
 - tipos de datos de, 38
 - variable de, 43-44
- Caja de Pascal, 41
- Campos, 260
- Caracteres, 260
- Carga adicional, 45
- Carpetas, 259
- Caso nulo, 88
- Cerrar un archivo, 264
- Ciclo(s), 18, 169-205
- anidados, 177, 177-182
- comprensión en la lógica de línea principal de un programa, 175-177
- contados (controlado por contador), 172
- cuerpo de, 88
- de lógica no estructurada, reconocimiento de, 335
- definidos, 172-173
- do-until**, 335
- do-while** (ciclo posprueba), 193, 317, 332-334, 333
- errores comunes en los, 183-191
- estructurados, características compartidas por, 334-335
- exterior, 177
- for**, 192-194
- for** para procesar arreglos, 244-245
 - indefinido, 173, 173-175
 - infinito, 18, 18-19
 - interior, 177
 - para acumular totales, 194-198
 - para validar datos, 198-200
 - posprueba (**do-while**), 193, 317, 332-334, 333
 - preprueba, 193
 - que pide entradas de nuevo, limitación de un, 200-202
 - validación de la sensatez y consistencia de los datos, 203-204
 - validar tipos de datos, 202-203
 - variable de control de, 171-177
 - ventajas de crear, 170-171
- Ciclo(s) de desarrollo del programa, 7, 7-14
- paso de codificación, 10
 - paso de entender el programa, 8-9
 - paso de mantenimiento, 13-14
 - paso de planeación de la lógica, 9-10
 - paso de producción, 13
 - paso de prueba, 12-13
 - paso de traducción del programa al lenguaje de máquina, 10-11
- Ciclo(s) definido, 172
- con un contador, 172-173
- Clasificación, 267
- Cláusula
- else**, 125
 - if-then**, 125
- Codificación de un programa, 3, 10
- Código
- espagueti, 84, 84-86
 - fuente, 3
 - objeto, 3
- Cohesión funcional, 55
- Compiladores, 4

- Condición compuesta, **129**
 Confiabilidad, **50**, 50-51
 Constante(s)
 como el tamaño de un arreglo, 224
 como subíndices de un arreglo, 225
 como valores de elemento del arreglo, 225
 de cadena literal, **38**
 declarar dentro de los módulos, 55-57
 literales, **38**
 nombradas, **44**, 44-45
 numérica, **38**
 numérica literal, **38**
 Contador, **173**
 ciclos controlados por, **172**
 Control de interrupción (o interrupciones), **268**
 de nivel único, 269
 Conversión, **13**
 Correspondencia exacta, búsqueda de un arreglo para una, 226-229
 Cortes de línea confusos, evitar, 68
 Cortocircuito, evaluación de, **135**
 CPU (unidad central de procesamiento), **2**
- D**
- Datos
 base de, **261**
 diccionario de, **67**
 elementos de, **2**
 jerarquía de, **260**, 260-261
 numéricos, tipo de, **38**
 tipos de, **39**, 40, 43-44
 visibles, elementos de, **57**
 Decisiones AND, **129**, 129-138
 anidación para eficiencia, 132-134
 decisión anidada, 129-132
 errores comunes, 136-138
 operador AND. Véase operador AND
 Decisiones anidadas (*if* anidado), **130**, 130-132
 reemplazar con arreglos, 216-224
 Decisiones OR, **138**, 138-147
 errores comunes, 143-147
 para eficiencia, 140-141
 Declaración(es), **39**, 39-41
 de fin de estructura, **87**
 dentro de los módulos, 55-57
 endif, **87**
 for (ciclo **for**), **192**, 192-194
 if en cascada, **130**, 130-132
 largas, variables temporales para clasificar las, 68-69
 precisas, diseño de, 68-69
 return del módulo, **51**
 Declarar un archivo, 261-262
 Decrementar, **173**
 Depuración, **13**
 Descomposición funcional. Véase modularización
 Desplegar, 266
 Diagonal (/), operador de división, **47**
 Diagramas de flujo, **14**
 módulos, 52-54
 símbolos de, 315
 trazo de, 16-17
 Directorios, **259**
 Dispositivos
 de almacenamiento permanentes, **258**
 de entrada predeterminados, 264
 de entrada y salida predeterminados, **264**
 de salida predeterminados, 264
 Documentación, **9**
 externa, 64
 interna, 64
- E**
- EBCDIC (Extended Binary Coded Decimal Interchange Code; Código ampliado de intercambio decimal codificado en binario), **306**
 Ejecutar un programa, **4**
 Elegancia, 189
 Elementos, **214**, 214-215
 del arreglo, constantes como valores de, 225
 En ámbito, **57**
 Entra basura, sale basura (GIGO; garbage in, garbage out), **200**
 Entrada, **2**
 con eco, **70**, 69-71
 símbolo de, **16**
 Entrada anticipada (lectura anticipada), **98**
 para estructurar un programa, 95-101
 Eof, **21**
 Errores lógicos, **5**
 Escribir en un archivo, **264**
 Estructura(s), 83-111, **86**, 316-317. Véase también
 programas estructurados
 anidadas, **90**, 90-92
 apiladas, **89**, 89-90
 case, 317, 330-332
 ciclo, 88-89, 96-97, 316
 ciclo posprueba (**do-while**), 193, 317, 332-334, **333**
 de decisión de alternativa única (**if-then**), 317
 de decisión. Véase estructura de selección
 de secuencia, 86-87, 316

- estructurar y modularizar la lógica no estructurada, 105-110
if-**t**hen (decisión de alternativa única), 317
if-**t**hen-**e**lse, 87, 316
razones para la, 101-102
reconocimiento de la, 102-105
resolución de problemas de estructuración difíciles, 318-327
selección (decisión). *Véase* estructura de selección
Estructura de ciclo, **88**, 88-89, 96-97
 estructura de selección comparada con, 96
Estructura de secuencia, **86**, 86-87
 diagrama, 316
Estructura de selección, **87**, 87-88, 122-125
 case, 317, **330**, 330-332
 decisión de alternativa única (**i**f-**t**hen), 317
 diagrama, 316
 estructura de ciclo comparada con, 96
Exabytes, 313
Expresiones booleanas, **122**. *Véase* también Decisiones AND; Decisiones OR
 operador lógico NOT, 147
Expresiones triviales, **126**
Extended Binary Coded Decimal Interchange Code (EBCDIC; Código ampliado de intercambio decimal codificado en binario), **306**
- F**
Forzar, **201**
Fuera de límites, **243**
Función(es). *Véase* modularización; módulo(s)
- G**
Gigabytes, **258**, 313
GIGO (garbage in, garbage out; entra basura, sale basura), **200**
Gráfica(s)
 de espaciado de impresora, 328-329
 de jerarquía, **61**, 61-63
 IPO, **9**
 TOE, **9**
GUI (interfaz gráfica del usuario), **24**
- H**
Hardware, **2**
- I**
IDE (integrated development environment; ambiente de desarrollo integrado), **23**, 23-24
Identificadores, **39**, 40
 elección de, 66-67
If de alternativa
 dual (selecciones de alternativa dual), **88**
 única (selecciones de alternativa única), **88**
Impresión, 266
Incrementar, **173**
Indicadores, **69**
 claros, escritura de, 69-70
Índice, **214**
Información, **3**
Informes sumarios, **198**
Inicialización de la variable de control de ciclo, descuidar la, 183-184
Iniciar la variable, **40**
Instrucciones, repetición de las, 17-18
Interfaz gráfica del usuario (GUI), **24**
Intérprete, **3**, 4
Interrupciones, control de campo de, **269**
 de nivel único, **269**
 informe de, **268**, 268-269
 programa de, **268**
Iteración, **88**. *Véase* también estructura
- K**
Kilobytes, **258**, 313
- L**
Leer desde un archivo, **262**, 262-264
Lenguaje
 binario, **4**
 de máquina, **3**
 de bajo nivel, **11**
 traducir programas a, 10-11
 de programación, **3**
 de alto nivel, **10**
 interpretados (lenguajes de programación de scripting o lenguajes de script), **4**
Línea(s)
 de comandos, **24**
 de flujo, **16**
Llamar a un módulo, **48**, 48-49, 52
Lógica, **5**, 5-7
 de línea principal, **51**
 comprensión del ciclo en la, 175-177
 no estructurada, estructuración y modularización de la, 105-110
 planeación de la, 9-10
Lovelace, Ada Byron, 26
- M**
Mantenimiento, **13**, 13-14
Megabytes, **258**, 313

- M**
- Memoria
 - arreglos que ocupan la, 214-215
 - de acceso aleatorio (RAM), 3
 - de la computadora, 3
 - Métodos. *Véase* modularización; módulo(s)
 - Microsoft Visual Studio IDE, 23
 - Modularización, 48-61
 - abstracción, 49-50
 - configuración más común para lógica de línea principal, 57-61
 - de la lógica no estructurada, 105-110
 - declaración de variables y constantes dentro de los módulos, 55-57
 - proceso, 51-61
 - reutilización del trabajo, 50-51
 - varios programadores, 50
 - Módulo(s), 48
 - cohesión funcional, 55
 - cuerpo del, 51
 - declaración `return` del, 51
 - diagramas de flujo, 52-54
 - encabezado del, 51
 - encapsulados, 55
 - encapsulamiento, 55
 - llamar a un, 48-49, 52
 - nombre del, 52
 - portátiles, 57
- N**
- Nivel de programa global, 57
 - Nombramiento de variables, 41-42
 - Nombre(s)
 - autodocumentados, 66
 - del módulo, 52
 - Notación
 - de camello, 41
 - húngara, 41
 - Números
 - mágicos, 44
 - reales, 38
- O**
- Operaciones aritméticas, 45-48
 - Operador(es), 47
 - binarios, 42
 - condicional AND, 134. *Véase* también operador AND
 - condicional OR, 141. *Véase* también operador OR
 - de división (/), 47
 - de multiplicación (*), 47
 - de resta (-), 47
 - de suma (+), 47
 - lógico NOT, 147
 - mayor o igual que (>=), 126
 - mayor que (>), 126
 - menor o igual que (<=), 126
 - menor que (<), 126
 - no igual a (<>), 126
 - Operador(es) AND, 134, 134-136
 - combinación con operador OR, precedencia en, 154-157
 - evaluación de cortocircuito, 135
 - tablas de verdad, 135
 - Operador(es) de comparación relacionales, 126, 126-129
 - error común con los, 129
 - lista, 126
 - Operador(es) OR, 141, 141-142
 - precedencia cuando se combinan con operador AND, 154-157
 - Orden
 - ascendente, 273
 - de operaciones, 46
 - descendente, 273
- P**
- Palabras clave, 41
 - Paréntesis, nombres de módulo, 52
 - Petabytes, 313
 - Pila, 55
 - Poblar el arreglo, 215
 - Precedencia, 154
 - combinar operadores AND y OR, 154-157
 - Procedimientos. *Véase* modularización; módulo(s)
 - Procesamiento, 2
 - por lotes, 290
 - símbolos de, 16
 - Programa(s), 2
 - características de un buen diseño de, 63-72
 - codificación del, 3, 10
 - código de, 3
 - comentarios del, 64, 64-66
 - correr un, 3, 4
 - interactivo, 290
 - mantenimiento del, 13-14
 - no estructurados, 84, 84-86
 - poner en producción, 13
 - principal, 51
 - prueba del, 12-13
 - traducir al lenguaje de máquina, 10-11
 - valor centinela para terminar un, 19-22

Programas estructurados, 84. *Véase* también
estructura(s)
características, 94
entrada anticipada para, 95-101

Programación, 2
ambientes de, 22-24
defensiva, 198
evolución de los modelos de, 25-27
lenguaje de, 3
orientada hacia los objetos, 26
procedimental, 26
sin goto, 101

Prueba
de escritorio, 10
de programas, 12-13
de valores, 20

R

Rango, búsqueda en un arreglo para una correspondencia de, 237-241
Rango, comprobación de, 148, 148-154
errores comunes en la, 150-154
Registros, 260
Reglas de precedencia, 46
Relación indirecta, 234
Repetición, 88. *Véase* también estructura en ciclo
Reutilización, 50
modularización, 50-51
Ruta(s), 259
inalcanzable, 150, 150-152
sin salida, 150, 150-152

S

Salida, 3
Seguridad del tipo, 44
Seudocódigo, 14
escritura en, 15
Signo de suma (+), operador de adición, 45
Signo igual (=)
operador de asignación, 47
operador de equivalencia, 126
Signo menos (-), operador de resta, 47
Símbolo en diagramas de flujo
de decisión, 20, 315
de entrada/salida (I/O), 16, 315
de línea de flujo, 315
de llamada a módulo externo, 315
de llamada a módulo interno, 315
de proceso, 315
de salida, 16
I/O (entrada/salida), 16, 315
terminal, 16, 315

Sintaxis, 3
errores de, 3
Sistemas de numeración, 305-312
binario, 305, 305-312
decimal, 305, 307-312
hexadecimal, 311, 311-312
Sistemas de cómputo, 2, 2-4
Software, 2
de aplicación, 2
de sistema, 2
Stub, 181
Subíndice, 214
constante como, 225
fuera de límites, 243
Subrutinas. *Véase* modularización; módulo(s)

T

Tablas, 261
de verdad, 135
Tamaño del arreglo, 214
Tareas
de administración, 57
de ciclo detallado, 58
de fin de trabajo, 58
Terabyte, 313
Texto
archivos de, 258
editor de, 22, 22-23
Tiempo real, aplicaciones en, 290
Tomar una decisión, 20
Totales, ciclo para acumular, 194-198
Traducir programas al lenguaje de máquina, 10-11
Turing, Alan, 26

U

Unicode, 306
Unidad central de procesamiento (CPU), 2
Unir archivos, 273
Usuario(s), 8
ambientes de, 24-25
finales, 8

V

Validación
datos. *Véase* validar datos
de la sensatez y consistencia de los datos, 203-204
de un tipo de datos, 202-203
Validar datos, 198
ciclo para, 198-200
Valor(es), 19

- alfanuméricos, **38**
centinela, 19-21, **21**
comodines, **21**
de paso, **192**
Variable(s), **6**, 38-44
asignación de valores a las, 42-43
de punto flotante, **38**
declaración de, 39-41
declaración dentro de los módulos, 55-
57
enteras, 38
inicializar, 40
nombramiento de, 41-42
numérica, **43**
punto flotante, 38
tipos de datos de las, 43-44
y constantes locales, **57**
Variable(s) de control de ciclo, **171**, 171-177
ciclo definido con un contador, 172-173
ciclo indefinido con un valor centinela, 173-
175
comparación errónea con, 186-187
comprensión del ciclo en la lógica de línea
principal de un programa, 175-177
decrementar, 173
descuidar la alteración de, 185
descuidar la inicialización de, 183-184
incluyendo dentro del ciclo declaraciones que
pertenezcan al exterior del mismo, 187-191
incrementar, 173
Variable(s) temporales, **68**
para clarificar las declaraciones largas, 68-69

W

- `While . . . do` (ciclo `while`), **88**, 88-89
diagrama, 316

Y

- Yottabyte, 313

Z

- Zettabyte, 313

Introducción

A la Programación Lógica y Diseño

7a. Ed. | JOYCE FARRELL

Prepare para el éxito a los programadores principiantes con la muy efectiva *Introducción a la Programación Lógica y Diseño*, de Farrell. Este texto popular adopta un enfoque único independiente del lenguaje de programación con un énfasis en las convenciones modernas. El estilo de redacción del libro, claro y conciso, elimina la jerga altamente técnica mientras presenta conceptos universales de programación y alienta un estilo de programación y pensamiento lógico sólidos. Las explicaciones revisadas y más definidas de esta edición incorporan diagramas de flujo, seudocódigo y diagramas para asegurar que incluso los lectores sin experiencia previa en programación entiendan por completo los conceptos de la programación y el diseño modernos.

CARACTERÍSTICAS DEL TEXTO

- Ejercicios adicionales basados en la elaboración de diagramas de flujo y seudocódigo en esta edición ayudan a los estudiantes a obtener una mejor comprensión del alcance de la programación moderna.
- Explicaciones revisadas minuciosamente proporcionan la guía más clara posible para los lectores que no tienen experiencia previa en programación.
- Una abundancia de oportunidades de práctica probadas, incluyendo Ejercicios de programación, Ejercicios de eliminación de errores y cuestionarios “Dos verdades y una mentira”, mantienen a los estudiantes interesados y aprendiendo en forma activa.

ACERCA DEL AUTOR



Joyce Farrell ha escrito una amplia variedad de libros de texto de programación exitosos reconocidos por su estilo de redacción directo y claro y su presentación efectiva. Es autora de varios textos de Course Technology, incluyendo versiones orientadas hacia los objetos y extendidas de este libro, al igual que *Java Programming*, *Microsoft Visual C++* y *Object-Oriented Programming Using C++*. Instructora muy respetada, Farrell impartió temas de sistemas de información de computadora por más de 20 años en colegios en Illinois y Wisconsin.



Visite nuestro sitio en <http://latinoamerica.cengage.com>

ISBN-13: 978-607481905-2

ISBN-10: 607481905-X



9 786074 819052