

Jesse Liberty y David B. Horvath

SAMS

## Aprenda

las técnicas y conceptos  
para dominar C++ para  
Linux en sólo 21 días

## Aplique

sus conocimientos en la  
vida real

Aprendiendo®

# C++ para Linux®

Prentice  
Hall

en 21 Días

# Aprendiendo C++ para Linux en 21 Días

00143182



<b>DÍA 1</b>	Familiarícese con la forma en que trabajan en conjunto los componentes de un programa en C++. Entérese de lo que es una función y de lo que hace. Conozca algunas de las opciones disponibles para los compiladores de GNU.	<b>DÍA 2</b>	Entérese de lo que son las referencias, cuál es la diferencia en comparación con los apuntadores y cómo crearlas y utilizarlas. Además, descubra sus limitaciones y cómo utilizarlas para pasar variables hacia y desde las funciones.	<b>DÍA 3</b>	Declare y defina variables y constantes. Asigne valores a las variables y manipule esos valores. Escriba el valor de una variable en la pantalla.	<b>DÍA 4</b>	Aprenda lo relacionado con las instrucciones, los bloques y las expresiones. Ramifique su código con base en las condiciones. Conozca el contexto de un programa en C++, y cómo actuar en consecuencia.	<b>DÍA 5</b>	Aprenda a crear funciones, pasar variables hacia y desde las funciones, a combinar funciones en bibliotecas, y a crear proyectos (por medio de make).	<b>DÍA 6</b>	Defina una nueva clase y cree objetos de esa clase. Averigüe lo que son las funciones miembro y los datos miembro. Conozca los constructores y cómo utilizarlos.	<b>DÍA 7</b>	Aprenda q
<b>DÍA 8</b>	A qué son los flujos y cómo desearlos y cómo desearlos y cómo utilizarlos, aprenda qué es el espacio libre de almacenamiento y cómo manejar la memoria.	<b>DÍA 9</b>	Entérese de lo que son las referencias, cuál es la diferencia en comparación con los apuntadores y cómo crearlas y utilizarlas. Además, descubra sus limitaciones y cómo utilizarlas para pasar variables hacia y desde las funciones.	<b>DÍA 10</b>	Aprenda a sobre cargar funciones miembro y operadores, y a escribir funciones para soportar clases con variables asignadas en forma dinámica.	<b>DÍA 11</b>	Conozca qué es la herencia, cómo derivar una clase a partir de otra, y qué es el acceso protegido y cómo utilizarlo. Además, averigüe qué son las funciones virtuales.	<b>DÍA 12</b>	Conozca los arreglos, cómo declararlos, cómo procesar cadenas de caracteres en ellos; descubra la relación existente entre ellos y los apuntadores y utilice las funciones de cadenas de la biblioteca.	<b>DÍA 13</b>	Aprenda qué es la herencia múltiple y cómo utilizarla. Además, conozca la herencia virtual, los tipos de datos abstractos y las funciones virtuales puras.	<b>DÍA 14</b>	Conozca las variables y las funciones miembro estáticas. Cree y manipule apuntadores a funciones (y a funciones miembro), junto con arreglos de esos apuntadores.
<b>DÍA 15</b>	Entérese de lo que es la delegación y la delegación y cómo modelar completamente una cláusula de otra y siga cómo utilizar la herencia privada.	<b>DÍA 16</b>	Aprenda qué son los flujos y cómo se utilizan. Maneje la entrada y la salida y escriba a y lea desde archivos por medio de flujos.	<b>DÍA 17</b>	Conozca cómo se resuelven por nombre las clases y funciones. Cree y utilice un espacio de nombres, y aprenda a utilizar el espacio de nombres estándar conocido como std.	<b>DÍA 18</b>	Aprenda a utilizar el UML (Lenguaje de Modelado Unificado) para documentar y realizar el análisis y diseño orientados a objetos para comprender su problema y crear una solución.	<b>DÍA 19</b>	Conozca qué son las plantillas y cómo utilizarlas. Cree plantillas de clases y de funciones. Averigüe qué es la Biblioteca de Plantillas Estándar y cómo utilizarla.	<b>DÍA 20</b>	Descubra cómo se utilizan las excepciones, las cuestiones que provocan y cómo se utilizan en el manejo de errores. Conozca lo que es un depurador y cómo utilizar el depurador gdb de GNU.	<b>DÍA 21</b>	Entérese de lo que es la compilación condicional, la escritura de macros usando el preprocesador y cómo se utilizan los preprocesadores para depurar programas. Manipule bits individuales y utilicelos como banderas.
<b>DÍA 22</b>	Conozca más acerca del entorno de programación de Linux: el depurador, las bibliotecas, la edición, el control de la compilación, el control del código fuente y cómo obtener más información.	<b>DÍA 23</b>	Aprenda sobre la programación de shells en Linux: qué son los shells, cuáles shells están disponibles, cómo redireccionar la salida, configurar cuentas y crear secuencias de comandos de shell sencillas.	<b>DÍA 24</b>	Familiarícese con la programación de sistemas en Linux: cómo programar en C++ usando llamadas al sistema para interactuar con el sistema operativo. También se habla sobre los procesos y subprocesos.	<b>DÍA 25</b>	Conozca qué son las comunicaciones entre procesos en Linux: escribir programas que se comunicen con otros programas (procesos). Aquí se incluyen las tuberías con nombre e IPC de System V.	<b>DÍA 26</b>	Esta lección le presenta las herramientas disponibles en C++ y Linux para el desarrollo de GUIs (interfaces gráficas de usuario), que son interfaces más amigables para el usuario.				
<b>EM. ADICIONAL</b>	Esta semana consiste en cinco lecciones (bueno, no es una semana completa de calendario). Estas lecciones son las siguientes.												

Jesse Liberty  
David B. Horvath



Aprendiendo

# C++ para Linux

## en 21 Días

TRADUCCIÓN:

Alfonso Vidal Romero Elizondo  
Ingeniero en Electrónica y Comunicaciones

REVISIÓN TÉCNICA:

Hugo Jiménez Pérez  
Matemático



MÉXICO • ARGENTINA • BRASIL • COLOMBIA • COSTA RICA • CHILE  
ESPAÑA • GUATEMALA • PERÚ • PUERTO RICO • VENEZUELA

Datos de catalogación bibliográfica

**LIBERTY, JESSE Y B. HORVATH, DAVID**  
**Aprendiendo C++ para Linux en 21 Días**

PEARSON EDUCACIÓN, México, 2001

ISBN: 970-26-0012-X

Área: Computación

Formato: 18.5 x 23.5 cm

Páginas: 1144

**EDICIÓN EN ESPAÑOL**

**EDITOR DE DIVISIÓN COMPUTACIÓN:** ÓSCAR MADRIGAL MUÑIZ  
**SUPERVISOR DE TRADUCCIÓN:** ANTONIO NÚÑEZ RAMOS  
**SUPERVISOR DE PRODUCCIÓN:** RODRIGO ROMERO VILLALOBOS

**APRENDIENDO C++ PARA LINUX EN 21 DÍAS**

Versión en español de la obra titulada *SAMS Teach Yourself C++ for Linux in 21 Days*, de Jesse Liberty y David B. Horvath, publicada originalmente en inglés por SAMS, una división de Macmillan Computer Publishing, 201 W. 103<sup>rd</sup> Street, Indianapolis, Indiana, 46290, EUA.

Esta edición en español es la única autorizada.

Authorized translation from the English language edition published by Macmillan Computer Publishing.  
Copyright © 2000

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

**Primera Edición**

D.R. © 2000 por Pearson Educación de México, S.A. de C.V.  
Calle 4 No. 25-2do. Piso  
Fracc. Industrial Alce Blanco  
53370 Naucalpan de Juárez, Edo. de México

Cámara Nacional de la Industria Editorial Mexicana Registro No. 1031

Reservados todos los derechos. Ni la totalidad ni parte de esta publicación pueden reproducirse, registrarse o transmitirse, por un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea electrónico, mecánico, fotoquímico, magnético o electroóptico, por fotocopia, grabación o cualquier otro, sin permiso previo por escrito del editor.

El préstamo, alquiler o cualquier otra forma de cesión de uso de este ejemplar requerirá también la autorización del editor o de sus representantes.

ISBN: 970-26-0012-X de la versión en español  
ISBN: 0-672-31895-4 de la versión en inglés

Impreso en México, Printed in Mexico

1 2 3 4 5 6 7 8 9 0 - XX 03 02 10



**EDITOR ASOCIADO**

Michael Stephens

**EDITOR DE ADQUISICIONES**

Carol Ackerman

**EDITOR DE DESARROLLO**

Robyn Thomas

**EDITOR A ADMINISTRATIVA**

Charlotte Clapp

**EDITOR DE PROYECTO**

Christina Smith

**CORRECTORA DE ESTILO**

Margaret Berson

**ÍNDICE**

Eric Schroeder

**CORRECTORA DE PRUEBAS**

Candice Hightower

**REVISORES TÉCNICOS**

Javad Abdollahi

Sean Coughlin

Rory Bray

**COORDINADOR DEL EQUIPO**

Pamalee Nelson

**DESARROLLO DE MEDIOS**

Jason Haines

**DISEÑO DE INTERIORES**

Gary Adair

**DISEÑO DE PORTADA**

Aren Howell

**REDACTOR**

Eric Borgert

**PRODUCCIÓN**

Brandon Allen

Cheryl Lynch

# Resumen de contenido

## Introducción

### Semana 1 De un vistazo

- Día 1 Comencemos
- 2 Los componentes de un programa de C++
- 3 Variables y constantes
- 4 Expresiones e instrucciones
- 5 Funciones
- 6 Clases base
- 7 Más flujo de programa

### Semana 1 Repaso

### Semana 2 De un vistazo

- Día 8 Apuntadores
- 9 Referencias
- 10 Funciones avanzadas
- 11 Herencia
- 12 Arreglos, cadenas tipo C y listas enlazadas
- 13 Polimorfismo
- 14 Clases y funciones especiales

### Semana 2 Repaso

### Semana 3 De un vistazo

- Día 15 Herencia avanzada
- 16 Flujos
- 17 Espacios de nombres
- 18 Análisis y diseño orientados a objetos
- 19 Plantillas
- 20 Excepciones y manejo de errores
- 21 Qué sigue

Introducción	1
<b>Semana 1 De un vistazo</b>	<b>5</b>
Día 1 Comencemos	29
2 Los componentes de un programa de C++	43
3 Variables y constantes	67
4 Expresiones e instrucciones	99
5 Funciones	145
6 Clases base	161
7 Más flujo de programa	215
<b>Semana 1 Repaso</b>	<b>223</b>
<b>Semana 2 De un vistazo</b>	<b>225</b>
Día 8 Apuntadores	259
9 Referencias	293
10 Funciones avanzadas	333
11 Herencia	367
12 Arreglos, cadenas tipo C y listas enlazadas	413
13 Polimorfismo	455
<b>Semana 2 Repaso</b>	<b>487</b>
<b>Semana 3 De un vistazo</b>	<b>499</b>
Día 15 Herencia avanzada	501
16 Flujos	557
17 Espacios de nombres	599
18 Análisis y diseño orientados a objetos	619
19 Plantillas	661
20 Excepciones y manejo de errores	713
21 Qué sigue	747

<b>Semana 3 Repaso</b>	<b>791</b>
<b>Semana 4 De un vistazo</b>	<b>805</b>
Día 22 El entorno de programación de Linux	807
23 Programación shell	835
24 Programación de sistemas	853
25 Comunicación entre procesos	873
26 Programación de la GUI	895
<b>Semana 4 Repaso</b>	<b>943</b>
Apéndice A Precedencia de operadores	945
B Palabras reservadas de C++	947
C Números binarios, octales, hexadecimales y una tabla de valores ASCII	949
D Respuestas a los cuestionarios y ejercicios	965
Índice	1067

# Contenido

Introducción	1
<b>Semana 1 De un vistazo</b>	<b>5</b>
<b>DÍA 1 Comencemos</b>	<b>7</b>
Qué es GNU .....	7
Cómo obtener las herramientas GNU .....	8
Una breve historia acerca de C++ .....	9
Programas .....	10
Solución de problemas .....	11
Programación procedural, estructurada y orientada a objetos .....	11
C++ y la programación orientada a objetos .....	13
Cómo evolucionó C++ .....	14
¿Primero debo aprender C? .....	15
C++ y Java .....	15
El estándar ANSI .....	15
Prepárese para programar .....	16
El entorno de desarrollo GNU/Linux .....	17
Cómo compilar el código fuente .....	18
Cómo crear un archivo ejecutable con el enlazador .....	19
El ciclo de desarrollo .....	19
"¡Hola, mundo!", su primer programa de C++ .....	20
Uso del compilador g++ .....	23
Construcción del proyecto ¡Hola, mundo! .....	23
Errores de compilación .....	24
Resumen .....	25
Preguntas y respuestas .....	25
Taller .....	26
Cuestionario .....	26
Ejercicios .....	26
<b>DÍA 2 Los componentes de un programa de C++</b>	<b>29</b>
Un programa sencillo .....	29
Un vistazo breve a cout .....	32
Comentarios .....	34
Tipos de comentarios .....	34
Uso de comentarios .....	34
Un consejo final sobre los comentarios .....	35
Funciones .....	36
Uso de funciones .....	37

Más acerca del compilador GNU .....	39
Opciones del compilador GCC .....	40
Tips sobre el compilador GNU .....	40
Resumen .....	41
Preguntas y respuestas .....	41
Taller .....	42
Cuestionario .....	42
Ejercicios .....	42
<b>DÍA 3 Variables y constantes</b> .....	<b>43</b>
Qué es una variable .....	43
Cómo reservar memoria .....	44
Cómo determinar el tamaño de los enteros y otros tipos de datos .....	45
Uso de enteros con signo y sin signo .....	46
Tipos de variables fundamentales .....	47
Definición de una variable .....	48
Sensibilidad al uso de mayúsculas .....	49
Palabras reservadas .....	50
Cómo crear más de una variable a la vez .....	51
Cómo asignar valores a las variables .....	51
Uso de <code>typedef</code> .....	53
Cuándo utilizar <code>short</code> y cuándo utilizar <code>long</code> .....	54
Cómo sobregirar el valor de un entero sin signo .....	54
Cómo sobregirar el valor de un entero con signo .....	55
Uso de variables de tipo carácter .....	56
Los caracteres como números .....	57
Caracteres de impresión especiales .....	58
Uso de constantes .....	58
Constantes literales .....	58
Constantes simbólicas .....	59
Uso de constantes enumeradas .....	60
Resumen .....	63
Preguntas y respuestas .....	63
Taller .....	65
Cuestionario .....	65
Ejercicios .....	65
<b>DÍA 4 Expresiones e instrucciones</b> .....	<b>67</b>
Instrucciones .....	67
Espacio en blanco .....	68
Bloques de instrucciones e instrucciones compuestas .....	68
Expresiones .....	69
Operadores .....	71

Operador de asignación .....	71
Operadores matemáticos .....	71
División de enteros y el operador de módulo .....	72
Cómo combinar los operadores de asignación y matemáticos .....	74
Incremento y decremento .....	75
Prefijo y posfijo .....	75
Precedencia de operadores .....	77
Paréntesis anidados .....	78
La naturaleza de la verdad .....	79
Operadores relacionales .....	80
La instrucción <code>if</code> .....	81
Estilos de sangría .....	84
<code>else</code> .....	85
Instrucciones <code>if</code> avanzadas .....	86
Llaves en instrucciones <code>if</code> complejas .....	88
Operadores lógicos .....	91
Operador lógico <code>AND</code> .....	91
Operador lógico <code>OR</code> .....	91
Operador lógico <code>NOT</code> .....	92
Evaluación de corto circuito .....	92
Precedencia relacional .....	92
Más sobre falso y verdadero .....	93
Operador condicional (ternario) .....	94
Resumen .....	95
Preguntas y respuestas .....	96
Taller .....	96
Cuestionario .....	96
Ejercicios .....	97
<b>DÍA 5 Funciones</b> .....	<b>99</b>
Qué es una función .....	99
Valores de retorno, parámetros y argumentos .....	100
Declaración y definición de funciones .....	101
Declaración de una función .....	101
Uso de los prototipos de funciones .....	102
Definición de una función .....	104
Ejecución de una función .....	105
Variables locales .....	106
Variables globales .....	108
Variables globales: una advertencia .....	109
Más acerca de las variables locales .....	110
Instrucciones de una función .....	111

Más acerca de los argumentos de funciones .....	112
Uso de funciones como parámetros para otras funciones .....	112
Los parámetros son variables locales .....	113
Más acerca de los valores de retorno .....	114
Parámetros predeterminados .....	116
Sobrecarga de funciones .....	119
Temas especiales sobre funciones .....	122
Funciones en línea .....	122
Recursión .....	124
Cómo trabajan las funciones: un vistazo a su interior .....	129
Niveles de abstracción .....	129
Partición de la RAM .....	129
La pila y las funciones .....	132
Programas de archivos fuente múltiples (bibliotecas de funciones creadas por el programador) .....	133
Cómo crear y utilizar bibliotecas de funciones con g++ .....	134
Creación de archivos de proyecto (para make) .....	135
Funciones de bibliotecas estándar de C++ (libg++) .....	137
Funciones matemáticas .....	137
Funciones de caracteres y de cadenas de caracteres. ....	138
Funciones generales .....	139
Mucho más .....	140
Resumen .....	141
Preguntas y respuestas .....	141
Taller .....	142
Cuestionario .....	142
Ejercicios .....	143
<b>Día 6 Clases base</b> .....	<b>145</b>
Creación de nuevos tipos .....	145
¿Por qué crear un nuevo tipo? .....	146
Introducción a las clases y miembros .....	146
Declaración de una clase .....	147
Unas palabras sobre las convenciones de denominación .....	147
Declaración de un objeto .....	148
Comparación de clases y objetos .....	148
Cómo acceder a los miembros de las clases .....	149
Asignar a objetos, no a clases .....	149
Si no lo declara, su clase no lo tendrá .....	150
Definición del alcance público en comparación con la del privado .....	151
Debe hacer que los datos miembro sean privados .....	153
Distinción entre privacidad y seguridad .....	155

Implementación de los métodos de una clase .....	156
Comprensión de los constructores y destructores .....	159
Constructores y destructores predeterminados .....	160
Uso del constructor predeterminado .....	160
Uso de funciones miembro <code>const</code> .....	163
Distinción entre interfaz e implementación .....	164
Dónde colocar declaraciones de clases y definiciones de métodos .....	167
Aplicación de la implementación en línea .....	169
Uso de clases con otras clases como datos miembro .....	171
Uso de estructuras .....	175
Por qué dos palabras reservadas hacen lo mismo .....	176
Resumen .....	176
Preguntas y respuestas .....	177
Taller .....	178
Cuestionario .....	178
Ejercicios .....	178
<b>DÍA 7 Más flujo de programa</b>	<b>181</b>
Uso de los ciclos .....	181
Las raíces del uso de ciclos: la instrucción <code>goto</code> .....	181
Por qué se evita el uso de <code>goto</code> .....	183
Ciclos <code>while</code> .....	183
Instrucciones <code>while</code> más complicadas .....	185
Uso de <code>continue</code> y <code>break</code> en ciclos .....	186
Ciclos <code>while</code> ( <code>true</code> ) .....	189
Limitaciones del ciclo <code>while</code> .....	191
Ciclos <code>do...while</code> .....	192
Ciclos <code>for</code> .....	194
Ciclos <code>for</code> avanzados .....	196
Ciclos <code>for</code> vacíos .....	199
Ciclos anidados .....	200
Los ciclos <code>for</code> y su alcance .....	202
Resumen de los ciclos .....	203
Instrucciones <code>switch</code> .....	205
Uso de una instrucción <code>switch</code> con un menú .....	208
Resumen .....	212
Preguntas y respuestas .....	212
Taller .....	212
Cuestionario .....	213
Ejercicios .....	213

<b>Semana 2 De un vistazo</b>	<b>223</b>
<b>Día 8 Apuntadores</b>	<b>225</b>
<i>¿Qué es un apuntador?</i> .....	225
Cómo guardar la dirección en un apuntador .....	228
Elección de nombres de apuntadores .....	229
Uso del operador de indirección .....	229
Apuntadores, direcciones y variables .....	230
Manipulación de datos mediante el uso de apuntadores .....	231
Cómo examinar una dirección .....	232
<i>¿Por qué utilizar apuntadores?</i> .....	234
La pila y el heap .....	234
new .....	236
delete .....	236
Fugas de memoria .....	239
Objetos en el heap .....	239
Creación de objetos en el heap .....	240
Eliminación de objetos .....	240
Acceso a los datos miembro .....	241
Datos miembro en el heap .....	242
Apuntadores especiales y otras cuestiones .....	245
El apuntador <code>this</code> .....	246
Apuntadores perdidos, descontrolados o ambulantes .....	247
Apuntadores <code>const</code> .....	250
Aritmética de apuntadores .....	253
Resumen .....	256
Preguntas y respuestas .....	257
Taller .....	257
Cuestionario .....	257
Ejercicios .....	258
<b>Día 9 Referencias</b>	<b>259</b>
<i>¿Qué es una referencia?</i> .....	259
Uso del operador de dirección ( <code>&amp;</code> ) en referencias .....	261
Las referencias no se pueden reasignar .....	262
<i>¿Qué se puede referenciar?</i> .....	264
Uso de apuntadores nulos y referencias nulas .....	266
Paso de argumentos de funciones por referencia .....	266
Cómo hacer que <code>intercambiar()</code> funcione con apuntadores .....	268
Cómo implementar a <code>intercambiar()</code> con referencias .....	269
Comprensión de los encabezados y prototipos de funciones .....	271
Regreso de varios valores por medio de apuntadores .....	272
Regreso de valores por referencia .....	274

Cómo pasar parámetros por referencia para tener eficiencia .....	275
Paso de un apuntador <code>const</code> .....	278
Referencias como alternativa para los apuntadores .....	281
Cuándo utilizar referencias y cuándo utilizar apuntadores .....	283
Cómo mezclar referencias y apuntadores .....	284
¡No regrese una referencia a un objeto que esté fuera de alcance! .....	285
Cómo regresar una referencia a un objeto en el heap .....	287
¿A quién pertenece el apuntador? .....	289
Resumen .....	290
Preguntas y respuestas .....	290
Taller .....	291
Cuestionario .....	291
Ejercicios .....	291
<b>DÍA 10 Funciones avanzadas</b> .....	<b>293</b>
Sobrecarga de funciones miembro .....	293
Uso de valores predeterminados .....	296
Cómo elegir entre valores predeterminados y funciones sobrecargadas .....	298
Constructores predeterminados .....	298
Sobrecarga de constructores .....	299
Inicialización de objetos .....	300
Uso del constructor de copia .....	302
Sobrecarga de operadores .....	306
Cómo escribir una función de incremento .....	307
Sobrecarga del operador de prefijo .....	308
Cómo regresar tipos en funciones con operadores sobrecargados .....	310
Cómo regresar objetos temporales sin nombre .....	311
Uso del apuntador <code>this</code> .....	313
Sobrecarga del operador de posfijo .....	314
Cuáles son las diferencias entre prefijo y posfijo .....	314
Uso del operador de suma .....	317
Cómo sobrecargar a <code>operator+</code> .....	319
Cuestiones adicionales relacionadas con la sobrecarga de operadores .....	320
Limitaciones de la sobrecarga de operadores .....	321
Qué se debe sobrecargar .....	321
Uso del operador de asignación .....	321
Operadores de conversión .....	324
Cómo crear sus propios operadores de conversión .....	327
Resumen .....	329
Preguntas y respuestas .....	329
Taller .....	330
Cuestionario .....	330
Ejercicios .....	331

<b>DÍA 11 Herencia</b>	<b>333</b>
Qué es la herencia .....	333
Herencia y derivación .....	334
Cómo crear clases que representen animales .....	335
La sintaxis de la derivación .....	335
Comparación entre privado y protegido .....	337
Constructores y destructores .....	340
Paso de argumentos a los constructores base .....	342
Redefinición de funciones .....	346
Cómo ocultar el método de la clase base .....	348
Cómo llamar al método base .....	350
Métodos virtuales .....	352
Cómo trabajan las funciones virtuales .....	356
No puede llegar allá desde aquí .....	357
Partición de datos .....	357
Destructores virtuales .....	360
Constructores virtuales de copia .....	360
El costo de los métodos virtuales .....	363
Resumen .....	364
Preguntas y respuestas .....	364
Taller .....	365
Cuestionario .....	365
Ejercicios .....	366
<b>DÍA 12 Arreglos, cadenas tipo C y listas enlazadas</b>	<b>367</b>
Qué es un arreglo .....	367
Cómo acceder a los elementos de un arreglo .....	368
Cómo escribir mas allá del fin de un arreglo .....	369
Inicialización de arreglos .....	373
Declaración de arreglos .....	374
Arreglos de objetos .....	375
Trabajo con arreglos multidimensionales .....	377
Una palabra sobre los arreglos y la memoria .....	379
Uso del heap para solucionar problemas relacionados con la memoria .....	380
Arreglos de apuntadores .....	380
Declaración de arreglos en el heap .....	381
Uso de un apuntador a un arreglo en comparación con un arreglo de apuntadores .....	382
Uso de apuntadores con nombres de arreglos .....	382
Eliminación de arreglos en el heap .....	384
Qué son los arreglos de tipo char .....	385
Uso de funciones para cadenas .....	387
Uso de cadenas y apuntadores .....	389
Clases de cadenas .....	391

Listas enlazadas y otras estructuras .....	398
Análisis de un caso de prueba de listas enlazadas .....	399
Delegación de responsabilidad .....	399
Los componentes de una lista enlazada .....	400
¿Qué ha aprendido? .....	409
Uso de clases de arreglos en lugar de arreglos integrados .....	409
Resumen .....	410
Preguntas y respuestas .....	410
Taller .....	411
Cuestionario .....	411
Ejercicios .....	412
<b>DÍA 13 Polimorfismo</b>	<b>413</b>
Problemas con herencia simple .....	413
Filtración ascendente .....	416
Conversión descendente .....	416
Cómo agregar objetos a dos listas .....	419
Herencia múltiple .....	420
Las partes de un objeto con herencia múltiple .....	423
Constructores en objetos con herencia múltiple .....	424
Resolución de ambigüedad .....	426
Herencia de una clase base compartida .....	427
Herencia virtual .....	431
Problemas con la herencia múltiple .....	435
Mezclas y clases de capacidad .....	436
Tipos de datos abstractos .....	436
Funciones virtuales puras .....	440
Implementación de funciones virtuales puras .....	441
Jerarquías de abstracción complejas .....	445
¿Qué tipos son abstractos? .....	449
El patrón observador .....	449
Unas palabras sobre la herencia múltiple, los tipos de datos abstractos y Java .....	450
Resumen .....	451
Preguntas y respuestas .....	451
Taller .....	452
Cuestionario .....	452
Ejercicios .....	453
<b>DÍA 14 Clases y funciones especiales</b>	<b>455</b>
Datos miembro estáticos .....	455
Funciones miembro estáticas .....	461
Apuntadores a funciones .....	463
Por qué utilizar apuntadores a funciones .....	467
Uso de arreglos de apuntadores a funciones .....	470

Paso de apuntadores a funciones hacia otras funciones .....	472
Uso de <code>typedef</code> con apuntadores a funciones .....	475
Apuntadores a funciones miembro .....	477
Arreglos de apuntadores a funciones miembro .....	480
Resumen .....	483
Preguntas y respuestas .....	483
Taller .....	484
Cuestionario .....	484
Ejercicios .....	484
<b>Semana 2 Repaso</b>	<b>487</b>
<b>Semana 3 De un vistazo</b>	<b>499</b>
<b>DÍA 15 Herencia avanzada</b>	<b>501</b>
Contención .....	501
Cómo tener acceso a miembros de una clase contenida .....	508
Cómo filtrar el acceso a los miembros contenidos .....	508
El costo de la contención .....	508
Cómo copiar por valor .....	511
Implementación con base en la herencia/contención en comparación con la delegación .....	515
Delegación .....	516
Herencia privada .....	525
Clases amigas .....	534
Funciones amigas .....	544
Funciones amigas y sobrecarga de operadores .....	544
Sobrecarga del operador de inserción .....	549
Resumen .....	553
Preguntas y respuestas .....	554
Taller .....	554
Cuestionario .....	554
Ejercicios .....	555
<b>DÍA 16 Flujos</b>	<b>557</b>
Panorama general sobre los flujos .....	557
Encapsulación .....	558
Almacenamiento en búfer .....	558
Flujos y búferes .....	560
Objetos de E/S estándar .....	561
Redirección .....	561
Entrada por medio de <code>cin</code> .....	562
Cadenas .....	564
Problemas con cadenas .....	564
<code>operator&gt;&gt;</code> regresa una referencia a un objeto <code>istream</code> .....	567

Otras funciones miembro de <code>cin</code> .....	567
Entrada de un solo carácter .....	567
Entrada de cadenas desde el dispositivo de entrada estándar .....	570
<code>cin.ignore()</code> para limpieza de la entrada .....	573
<code>peek()</code> y <code>putback()</code> .....	574
Salida con <code>cout</code> .....	575
Limpieza de la salida .....	575
Funciones relacionadas .....	575
Manipuladores, indicadores e instrucciones para dar formato .....	577
Uso de <code>cout.width()</code> .....	577
Cómo establecer los caracteres de llenado .....	578
Cómo establecer indicadores de <code>iostream</code> .....	579
Flujos en comparación con la función <code>printf()</code> .....	581
Entrada y salida de archivos .....	585
Uso de <code>ofstream</code> .....	585
Apertura de archivos para entrada y salida .....	585
Archivos binarios en comparación con archivos de texto .....	589
Procesamiento de la línea de comandos .....	592
Resumen .....	595
Preguntas y respuestas .....	596
Taller .....	597
Cuestionario .....	597
Ejercicios .....	597
<b>DÍA 17 Espacios de nombres</b>	<b>599</b>
Comencemos con los espacios	
de nombres .....	599
Cómo se resuelven por medio del nombre las funciones y las clases .....	600
Creación de un espacio de nombres .....	604
Declaración y definición de tipos .....	605
Cómo definir funciones fuera de un espacio de nombres .....	605
Cómo agregar nuevos miembros .....	606
Cómo anidar espacios de nombres .....	606
Uso de un espacio de nombres .....	607
Presentación de la palabra reservada <code>using</code> .....	609
La directiva <code>using</code> .....	609
La declaración <code>using</code> .....	611
Uso del alias de un espacio de nombres .....	613
Uso del espacio de nombres sin nombre .....	613
Uso del espacio de nombres estándar <code>std</code> .....	614
Resumen .....	615
Preguntas y respuestas .....	616
Taller .....	617
Cuestionario .....	617
Ejercicios .....	617

<b>Día 18 Análisis y diseño orientados a objetos</b>	<b>619</b>
¿Es C++ un lenguaje orientado a objetos?	619
Qué son los modelos .....	620
Diseño de software: el lenguaje de modelado .....	621
Diseño de software: el proceso .....	622
Conceptualización: la visión .....	625
Análisis de los requerimientos .....	625
Casos de uso .....	626
Análisis de la aplicación .....	637
Análisis de los sistemas .....	637
Documentos de planeación .....	638
Visualizaciones .....	639
Artefactos .....	639
Diseño .....	640
Qué son las clases .....	640
Transformaciones .....	642
Modelo estático .....	643
Modelo dinámico .....	653
No se apresure a llegar al código .....	657
Resumen .....	658
Preguntas y respuestas .....	658
Taller .....	659
Cuestionario .....	659
Ejercicios .....	659
<b>Día 19 Plantillas</b>	<b>661</b>
Qué son las plantillas .....	661
Tipos parametrizados .....	662
Cómo crear una instancia a partir de una plantilla .....	662
Definición de una plantilla .....	662
Uso del nombre .....	664
Implementación de la plantilla .....	665
Funciones de plantillas .....	668
Plantillas y funciones amigas .....	669
Clases y funciones amigas que no son de plantilla .....	669
Clases y funciones amigas de plantilla general .....	673
Uso de elementos de plantilla .....	677
Funciones especializadas .....	681
Miembros estáticos y plantillas .....	687
La Biblioteca Estándar de Plantillas .....	691
Contenedores .....	692
Contenedores de secuencia .....	692
Contenedores asociativos .....	701

Pilas .....	705
Colas .....	706
Clases de algoritmos .....	706
Algoritmos de secuencia no mutante .....	707
Algoritmos de secuencia mutante .....	708
Resumen .....	709
Preguntas y respuestas .....	710
Taller .....	711
Cuestionario .....	711
Ejercicios .....	711
<b>DÍA 20 Excepciones y manejo de errores</b>	<b>713</b>
Bugs y corrupción de código .....	714
Excepciones .....	715
Unas palabras acerca de la corrupción del código .....	715
Excepciones .....	716
Cómo se utilizan las excepciones .....	716
Uso de los bloques <code>try</code> y <code>catch</code> .....	721
Cómo atrapar excepciones .....	722
Más de una especificación <code>catch</code> .....	722
Jerarquías de excepciones .....	725
Acceso a los datos de excepciones mediante la denominación de objetos de excepciones .....	728
Uso de excepciones y plantillas .....	735
Cómo activar excepciones sin errores .....	738
Cómo tratar con los errores y la depuración .....	739
Uso de <code>gdb</code> o depurador GNU .....	740
Uso de los puntos de interrupción .....	742
Uso de los puntos de observación .....	742
Examen y modificación del estado de la memoria .....	742
Desensamblaje .....	742
Resumen .....	743
Preguntas y respuestas .....	743
Taller .....	744
Cuestionario .....	744
Ejercicios .....	745
<b>DÍA 21 Qué sigue</b>	<b>747</b>
El preprocesador y el compilador .....	748
Cómo ver el formato del archivo intermedio .....	748
La directiva de preprocesador <code>#define</code> .....	748
Uso de <code>#define</code> como alternativa para constantes .....	749
Uso de <code>#define</code> para probar el código .....	749
Uso de la directiva de precompilador <code>#else</code> .....	749

Inclusión y guardias de inclusión .....	751
Funciones de macros .....	752
<i>¿Para qué son todos esos paréntesis?</i> .....	753
Macros en comparación con funciones y plantillas .....	754
Funciones en línea .....	755
Manipulación de cadenas .....	756
Uso de cadenas con la directiva <code>#define</code> .....	757
Concatenación .....	757
Macros predefinidas .....	758
<b>ASSERT()</b> .....	758
Depuración con <b>ASSERT()</b> .....	760
<b>ASSERT()</b> en comparación con las excepciones .....	760
Efectos secundarios .....	761
Constantes de clases .....	762
Impresión de valores interinos .....	767
Niveles de depuración .....	768
Manipulación de bits .....	775
Operador AND .....	775
Operador OR .....	776
Operador OR exclusivo .....	776
El operador de complemento .....	776
Cómo encender bits .....	776
Cómo apagar bits .....	776
Cómo invertir los bits .....	777
Campos de bits .....	777
Estilo de codificación .....	780
Uso de sangrías .....	781
Llaves .....	781
Líneas largas .....	781
Instrucciones <code>switch</code> .....	781
Texto del programa .....	782
Nombres de identificadores .....	783
Ortografía y uso de mayúsculas en los nombres .....	783
Comentarios .....	784
Acceso .....	784
Definiciones de clases .....	785
Archivos de encabezado .....	785
<b>ASSERT()</b> .....	785
<code>const</code> .....	786
Los siguientes pasos .....	786
Dónde obtener ayuda y orientación .....	786
Revistas .....	786

---

Manténgase en contacto .....	787
Su próximo paso .....	787
Resumen .....	787
Preguntas y respuestas .....	788
Taller .....	789
Cuestionario .....	789
Ejercicios .....	790
<b>Semana 3 Repaso</b>	<b>791</b>
<b>Semana 4 De un vistazo</b>	<b>805</b>
<b>DÍA 22 El entorno de programación de Linux</b>	<b>807</b>
Filosofía e historia .....	808
POSIX .....	808
El sistema X Windows .....	808
Uso de los editores de Linux .....	809
ed, ex, vi y las variantes de vi .....	809
emacs de GNU .....	812
ctags y etags .....	815
Lenguajes .....	816
gcc y g++ .....	816
Lenguajes de secuencias de comandos (perl, sed, awk) .....	818
ELF .....	818
Bibliotecas compartidas .....	819
Construcción o creación .....	820
make .....	820
Depuración .....	823
gdb .....	823
xxgdb .....	825
Sesión de ejemplo de depuración con gdb .....	826
Control de versiones .....	827
RCS .....	828
Documentación .....	830
Páginas del manual .....	830
info .....	831
HOWTOs y FAQs .....	832
Resumen .....	832
Preguntas y respuestas .....	833
Taller .....	834
Cuestionario .....	834
Ejercicios .....	834

<b>Día 23 Programación shell</b>	<b>835</b>
Qué es un shell .....	836
Shells disponibles en Linux .....	836
Operación de los shells y conceptos básicos de sintaxis .....	836
Características del shell .....	837
Redirección de E/S .....	838
Tuberías .....	839
Variables .....	840
Variables utilizadas por el shell .....	840
Variables establecidas por el shell .....	841
Procesamiento en segundo plano, suspensión y control de procesos .....	841
Completabilidad de comandos .....	842
Sustitución de comandos .....	843
Sustitución mediante comodines .....	843
Sustitución mediante cadenas .....	844
Sustitución mediante la salida de un comando .....	844
Sustitución mediante variables .....	844
Historial y edición de comandos .....	845
Creación de alias de comandos .....	845
Secuencias de comandos de los shells .....	846
Variables .....	846
Estructuras de control .....	847
Archivo(s) de inicio de shell .....	849
Resumen .....	850
Preguntas y respuestas .....	850
Taller .....	851
Cuestionario .....	851
Ejercicios .....	851
<b>Día 24 Programación de sistemas</b>	<b>853</b>
Procesos .....	853
Creación y terminación de procesos .....	854
Control de procesos .....	856
El sistema de archivos /proc .....	858
Estado y prioridad del proceso .....	859
Algoritmos de administración de procesos .....	860
Subprocesos .....	861
Subprocesamiento simple .....	862
Subprocesamiento múltiple .....	862
Creación y terminación de subprocesos .....	862
Administración .....	864
Sincronización .....	864

---

Resumen .....	871
Preguntas y respuestas .....	871
Taller .....	871
Cuestionario .....	872
Ejercicios .....	872
<b>DÍA 25 Comunicación entre procesos</b>	<b>873</b>
Antecedentes .....	873
Tuberías .....	874
<popen .....<="" pclose="" td="" y=""><td>877</td></popen>	877
Tuberías con nombre (FIFOs) .....	877
Comunicación entre procesos de System V .....	879
Creación de claves .....	879
Estructura de permisos de IPC .....	881
Comandos ipcs e ipcrm .....	881
Colas de mensajes .....	882
Semáforos .....	886
Memoria compartida .....	890
IPC .....	892
Resumen .....	893
Preguntas y respuestas .....	893
Taller .....	893
Cuestionario .....	893
Ejercicios .....	894
<b>DÍA 26 Programación de la GUI</b>	<b>895</b>
El escritorio de Linux .....	896
Qué es GNOME .....	899
Cómo obtener GNOME y otros recursos de GNOME .....	900
Cómo obtener GTK++ y otros recursos de GTK++ .....	901
Qué es KDE .....	901
Cómo obtener KDE y otros recursos de KDE .....	902
Programación con C++ en el escritorio de Linux .....	903
Fundamentos de la programación en GNOME .....	904
Cómo envolver a GTK++ con wxWindows .....	909
Creación de su primera aplicación de wxWindows: “¡Hola, mundo!” .....	910
Cómo agregar botones a su propia clase de ventana de wxWindows .....	913
Cómo agregar un menú a su propia clase de ventana wxWindows .....	920
Creación de aplicaciones wxWindows más complejas: wxStudio .....	923
Cómo obtener wxWindows y otros recursos de wxWindows .....	924
Fundamentos de la programación de KDE .....	924
Creación de su primera aplicación KDE: “Hello World” .....	924
Creación de su propia clase de ventana de KDE .....	926

Cómo agregar botones a su propia clase de ventana de KDE	928
Interacción de objetos por medio de señales y ranuras	931
Cómo agregar un menú a su propia clase de ventana de KDE	935
Creación de aplicaciones KDE más complejas: KDevelop	937
Resumen .....	939
Preguntas y respuestas .....	939
Taller .....	941
Cuestionario .....	941
Ejercicios .....	941
<b>Semana 4 Repaso</b>	<b>943</b>
<b>Apéndice A Precedencia de operadores</b>	<b>945</b>
<b>Apéndice B Palabras reservadas de C++</b>	<b>947</b>
<b>Apéndice C Números binarios, octales, hexadecimales y una tabla de valores ASCII</b>	<b>949</b>
Más allá de la base 10 .....	950
Un vistazo a las bases .....	951
Números binarios .....	952
Por qué base 2 .....	953
Bits, Bytes, y Nibbles .....	953
Qué es un KB .....	954
Números binarios .....	954
Números octales .....	954
Por qué octal .....	955
Números hexadecimales .....	955
ASCII .....	958
<b>Apéndice D Respuestas a los cuestionarios y ejercicios</b>	<b>965</b>
Día 1 .....	965
Cuestionario .....	965
Ejercicios .....	966
Día 2 .....	966
Cuestionario .....	966
Ejercicios .....	967
Día 3 .....	967
Cuestionario .....	967
Ejercicios .....	968
Día 4 .....	969
Cuestionario .....	969
Ejercicios .....	970

---

Día 5 .....	971
Cuestionario .....	971
Ejercicios .....	972
Día 6 .....	975
Cuestionario .....	975
Ejercicios .....	976
Día 7 .....	978
Cuestionario .....	978
Ejercicios .....	979
Día 8 .....	981
Cuestionario .....	981
Ejercicios .....	981
Día 9 .....	983
Cuestionario .....	983
Ejercicios .....	983
Día 10 .....	986
Cuestionario .....	986
Ejercicio .....	987
Día 11 .....	992
Cuestionario .....	992
Ejercicios .....	992
Día 12 .....	994
Cuestionario .....	994
Ejercicios .....	994
Día 13 .....	995
Cuestionario .....	995
Ejercicios .....	996
Día 14 .....	997
Cuestionario .....	997
Ejercicios .....	998
Día 15 .....	1004
Cuestionario .....	1004
Ejercicios .....	1005
Día 16 .....	1009
Cuestionario .....	1009
Ejercicios .....	1010
Día 17 .....	1012
Cuestionario .....	1012
Ejercicios .....	1013
Día 18 .....	1013
Cuestionario .....	1013
Ejercicios .....	1014

Día 19 .....	1018
Cuestionario .....	1018
Ejercicios .....	1019
Día 20 .....	1025
Cuestionario .....	1025
Ejercicios .....	1026
Día 21 .....	1031
Cuestionario .....	1031
Ejercicios .....	1032
Día 22 .....	1035
Cuestionario .....	1035
Ejercicios .....	1035
Día 23 .....	1037
Cuestionario .....	1037
Ejercicio .....	1038
Día 24 .....	1038
Cuestionario .....	1038
Ejercicios .....	1039
Día 25 .....	1040
Cuestionario .....	1040
Ejercicios .....	1041
Día 26 .....	1049
Cuestionario .....	1049
Ejercicios .....	1050
<b>Índice</b>	<b>1067</b>

# Acerca de los autores

## Autores principales

**Jesse Liberty** es autor de *C++ para principiantes*, así como de varios libros más sobre C++ y desarrollo de aplicaciones Web y programación orientada a objetos. Jesse es presidente de Liberty Associates, Inc., en donde ofrece programación personalizada, entrenamiento, enseñanza y consultoría. Fue vicepresidente de transferencias electrónicas de Citibank e Ingeniero de software distinguido en AT&T. También se desempeña como editor de la serie de libros *Programming From Scratch*, de Que. Ofrece apoyo para sus libros en <http://www.LibertyAssociates.com>.

**David B. Horvath**, CCP, es Consultor en jefe del área de Filadelfia, en Pensilvania, Estados Unidos. Ha sido consultor por más de 14 años y también es profesor adjunto de medio tiempo en universidades locales, en donde enseña temas que incluyen la programación en C/C++, UNIX y técnicas de bases de datos. Tiene una maestría en Dinámica organizacional de la Universidad de Pensilvania (y sigue tomando clases). Ha ofrecido seminarios y talleres a sociedades profesionales y empresas en todo el mundo.

David es autor de *UNIX for the Mainframe* (Prentice Hall/PTR), autor contribuyente de *UNIX Unleashed*, segunda edición (con crédito en portada), *Red Hat Linux*, segunda edición, *Using UNIX*, segunda edición (Que), *UNIX Unleashed*, tercera edición, *Learn Shell Programming in 24 Hours*, *Red Hat Linux 6 Unleashed*, *Linux Unleashed*, cuarta edición, y ha escrito varios artículos en revistas.

Cuando no se encuentra escribiendo en el teclado de su computadora, puede encontrársele trabajando en el jardín de su casa, tomando un baño de agua caliente en la bañera, o involucrado en varias actividades que molestan a las personas. Tiene más de 12 años de casado y tiene varios perros y gatos (y parece que la cantidad de ellos sigue en aumento).

Para preguntas relacionadas con este libro, puede contactarse con David en [cpplinux@cobs.com](mailto:cpplinux@cobs.com), y su página Web es <http://www.cobs.com/>. ¡No envíe correo basura, por favor!

## Autores contribuyentes

**Paul Cevoli** es ingeniero de software y vive en Boston, MA, en donde ha ejercido esta profesión durante los últimos 11 años. Se especializa en software de sistema, controladores de dispositivos y componentes internos de sistemas operativos. Paul trabaja actualmente en el grupo de software de sistema de una compañía de aplicaciones de Internet en el área metropolitana de Boston.

**Jonathan Parry-McCulloch** es consultor técnico para una institución financiera importante en la ciudad de Londres, en donde ofrece su experiencia en criptografía para sistemas automatizados para comercio de instrumentos financieros a plazo. También escribe aplicaciones GUI multiplataforma en C++, principalmente por diversión, y es un fanático de Linux declarado. Tiene un título de primera clase en Ingeniería electrónica y está complacido de que no tenga relevancia en relación con su trabajo.

Puede contactarse con Jonathan en [jm@ant1pope.org](mailto:jm@ant1pope.org), si tiene preguntas sobre las partes en las que él contribuyó para este libro, o sobre cualquier otra cosa que se le ocurra. Una advertencia: él realmente odia el correo basura, así que no lo envíe.

**Hal Moroff** ha diseñado sistemas incrustados durante 20 años. Originario de la Costa Este, trabajó varios años en Europa y Japón, y ahora es programador de tiempo completo en Silicon Valley, California. Escribe con regularidad en una columna de *Linux Magazine*, y lee la fuente de información siempre que puede.

# Dedicatoria

*Este libro está dedicado a la memoria viviente de David Levine.*

—Jesse

*Me gustaría dedicar este libro a todos los buenos maestros que he tenido a través de los años. Aquellos que me enseñaron que las preguntas no son estúpidas, aunque parezcan ser simples; que hay más por aprender que lo que está escrito en el libro; que la mejor forma de aprender algo que requiera pensarse, es haciendo —lo que algunas personas llaman juego, a menudo es aprendizaje. ¡Esos maestros ya saben quiénes son!*

*También quiero dedicar este libro a mis estudiantes. Me han hecho algunas preguntas asombrosas. Pensé que sabía computación y programación antes de que empezara a enseñar; al tener que contestar sus preguntas, ¡me han obligado a aprender aún más sobre esos temas!*

—David

# Reconocimientos

Quisiera agradecer a todas las personas que hicieron posible esta edición y la anterior de este libro. En primer lugar se encuentran Stacey, Robin y Rachel Liberty. También debo agradecer a los editores de SAMS, incluyendo a Carol Ackerman, Tracy Dunkelberger, Holly Allender, Sean Dixon, Chris Denny, Brad Jones, Robyn Thomas, Christina Smith y Margaret Berson.

Muchos lectores contribuyeron con sugerencias y mejoras para este libro y les agradezco a todos ellos. Por último, y de nuevo, agradezco a la señora Kalish, quien en 1965 enseñó a mi clase de sexto grado cómo hacer aritmética binaria, cuando ni ella ni nosotros sabíamos por qué o para qué.

—Jesse

Quiero agradecer a todas esas excelentes personas de Sams (Macmillan Computer Publishing) que ayudaron a que este libro pudiera realizarse. Carol Ackerman (Editora de adquisiciones) vino a mí con la idea de escribir este libro, Christina Smith fue Editora de proyectos y guió al grupo durante el proceso de edición, Margaret Berson realizó la edición de la copia (revisión de gramática, sintaxis, claridad, etcétera), Robyn Thomas fue editor de desarrollo (contenido y estructura), y Javad Abdollahi y Sean Coughlin realizaron la mayor parte de la edición técnica, junto con Rory Bray. Sin ellos, este libro no sería tan bueno como lo es ahora.

Definitivamente debo agradecer a mi coautor, Jesse Liberty. Cuando Carol y yo empezamos a discutir sobre este proyecto, pedí una muestra de un libro de la serie *Aprendiendo en 21 días*, de Sams. Ella me envió una copia de *Sams Teach Yourself C in 21 Days*. Despues de revisar ese libro, comenté a Carol que este libro sería mucho más sencillo si ya existiera un libro de C++ similar al libro de C. Resultó que ya había uno — el de Jesse. Así que la creación de este libro fue mucho más sencilla gracias a la información de C++ que ya se había escrito y que se tomó como base. Despues de todo, hay tantas maneras de explicar un ciclo `while` o el concepto de polimorfismo.

Los autores contribuyentes, Hal, Paul y Jon, han sido de enorme ayuda para cubrir algunos de los temas más específicos de Linux que van más allá del lenguaje C++ mismo. Quien merece atención especial es Jon Parry-McCulloch, que se encargó del capítulo sobre la GUI casi de inmediato y tuvo que batallar con las diferencias de las zonas horarias (el vive en Inglaterra). Jon es contribuyente en el grupo de discusión por correo electrónico en Internet llamado Culto del padre Darwin (Cult of Father Darwin) (el cual discute la manifestación de las teorías de Darwin en las acciones diarias y los errores de varias personas). No voy a decirle cómo encontrar este grupo porque tratamos de mantener la relación Señal a ruido alta (si está interesado, tendrá que encontrarlo usted mismo). Tuve la oportunidad de conocer a Jon y a otro miembro del grupo cuando asistí a una clase en Londres en 1999. Existe una gran diferencia entre hablar vía correo electrónico y hablar en persona.

Como con cualquier proyecto grande, alguien tiene que hacer un sacrificio. En mi caso la que sufre más debido al tiempo que se necesita invertir en un libro como éste es mi esposa Mary. Lo asombroso es que ella ni siquiera se queja (cuando menos no mucho).

Y por último, aunque no menos importante, se encuentran las personas de DCANet ([www.dca.net](http://www.dca.net)) que proporcionan alojamiento virtual de sitios Web para mi página Web, manejan mi dominio, mantienen los servidores de mi correo, y tienen una gran reserva de módem que significa llamadas locales para mí. Estas personas son muy responsivas; la línea nunca estuvo ocupada, y ¡Andrew White es un Administrador de sistemas sin igual! Ellos se encuentran en el área metropolitana de Filadelfia, y he enviado a muchas organizaciones para que ocupen sus servicios. No, no recibo ningún pago por esta propaganda (sin embargo, ellos me cuidan muy bien).

Estoy seguro que con tanto ajetreo pude haber olvidado a alguien. ¡Les aseguro que no fue intencional!

—David

## Pearson Educación de México

El personal de Pearson Educación de México está comprometido en presentarle lo mejor en material de consulta sobre computación. Cada libro de Pearson Educación de México es el resultado de meses de trabajo de nuestro personal, que investiga y refina la información que se ofrece.

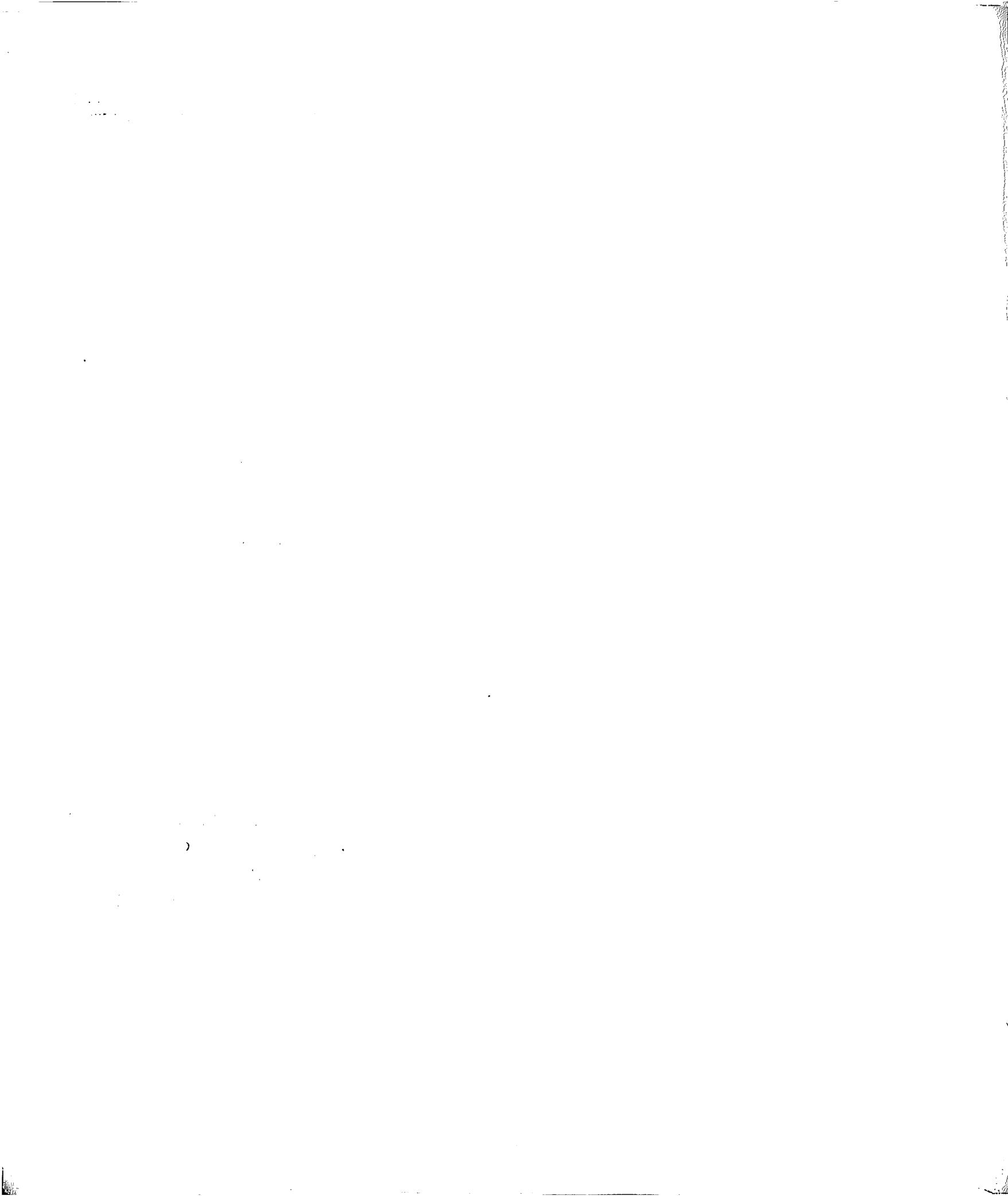
Como parte de este compromiso con usted, el lector de Pearson Educación de México lo invita a dar su opinión. Por favor háganos saber si disfruta este libro, si tiene alguna dificultad con la información y los ejemplos que se presentan, o si tiene alguna sugerencia para la próxima edición.

Sin embargo, recuerde que el personal de Pearson Educación de México no puede actuar como soporte técnico ni responder preguntas acerca de problemas relacionados con el software o el hardware.

Si usted tiene alguna pregunta o comentario acerca de cualquier libro de Pearson Educación de México, existen muchas formas de entrar en contacto con nosotros. Responderemos a todos los lectores que podamos. Su nombre, dirección y número telefónico jamás formarán parte de ninguna lista de correos ni serán usados para otro fin, más que el de ayudarnos a seguirle llevando los mejores libros posibles. Puede escribirnos a la siguiente dirección:

Pearson Educación de México  
Attn: Editorial División Computación  
Calle Cuatro No. 25, 2º Piso,  
Col. Fracc. Alce Blanco  
Naucalpan de Juárez, Edo. de México  
C.P. 53370.

Si lo prefiere, puede mandar un fax a Pearson Educación de México al (525) 5387-0811. También puede ponerse en contacto con Pearson Educación de México a través de nuestra página Web: <http://www.pearsonedlatino.com>



# Introducción

Este libro está diseñado para que usted aprenda a programar en C++ en el sistema operativo Linux. En sólo 21 días conocerá los aspectos básicos, como la administración de E/S, los ciclos y arreglos, la programación orientada a objetos, las plantillas y la creación de aplicaciones con C++, todo dentro de lecciones bien estructuradas y fáciles de entender. Para ilustrar los temas del día, en las lecciones se proporcionan listados de ejemplo, ejemplos de la salida y un análisis del código. Los ejemplos de sintaxis están explicados claramente para una útil referencia.

Para que se ponga al corriente con las características, herramientas y entorno específicos para Linux, ¡se incluye una semana adicional! Puede aprender C++ sin esa semana, pero sin duda le ayudará a mejorar sus habilidades específicas para Linux.

Para ayudarlo a desarrollar aún más sus habilidades, cada lección termina con un conjunto de preguntas y respuestas, un cuestionario y ejercicios. Puede comprobar su progreso revisando las respuestas a estas secciones, las cuales se proporcionan en el apéndice D, "Respuestas a los cuestionarios y ejercicios".

## Quién debe leer este libro

Con este libro, usted no necesita tener experiencia previa en programación para aprender C++. Con él empezará desde cero y aprenderá tanto el lenguaje como los conceptos relacionados con la programación en C++. A medida que avance en este entorno tan gratificante, descubrirá que los numerosos ejemplos de sintaxis y análisis de código son una guía excelente. No importa si es principiante o si tiene algo de experiencia en programación, la clara organización de este libro le facilitará y agilizará el aprendizaje de C++.

Este libro no le enseña cómo usar o instalar Linux, aunque se incluye una copia en el CD-ROM. Existen muchos otros buenos libros para ese propósito (como las series *Linux Unleashed*, de Sams).

C++ (al igual que su antecesor, el lenguaje C) es un lenguaje estándar, por lo que en esencia el lenguaje es el mismo tanto en Linux como en otras plataformas. En otras palabras, el lenguaje C++ que usted aprenda en este libro podrá aplicarlo en muchos sistemas y compiladores diferentes.

Hay dos compiladores distintos disponibles en Linux —gcc (Compilador C/C++ GNU) y egcs (Sistema Experimental del compilador GNU). El CD-ROM contiene egcs, mientras que las distribuciones de Linux anteriores (y la versión MS-DOS actual) utilizan gcc. Es difícil distinguir uno del otro desde la línea de comandos. Sin embargo, egcs tiene las características más avanzadas y actuales de C++.

Hay versiones disponibles de los compiladores GNU para la mayoría de las plataformas y sistemas operativos. Por esto, las habilidades que usted obtenga al leer este libro le serán útiles no sólo para Linux.

Debe estar ejecutando Linux para utilizar el compilador incluido.

## Convenciones

### Nota

Estos cuadros resaltan información que le ayuda a que su programación en C++ sea más eficiente y efectiva.

### Preguntas frecuentes

**FAQ:** ¿Qué hacen las FAQs?

**Respuesta:** Las preguntas frecuentes proporcionan mayores detalles acerca del uso del lenguaje, y aclaran las áreas potenciales de confusión.

### Precaución

Las precauciones dirigen su atención hacia los problemas o efectos secundarios que pueden ocurrir en situaciones específicas.

Estos cuadros proporcionan definiciones claras de términos esenciales.

### DEBE

**DEBE** utilizar los cuadros “Debe/No debe” para encontrar un resumen rápido de un principio fundamental de una lección.

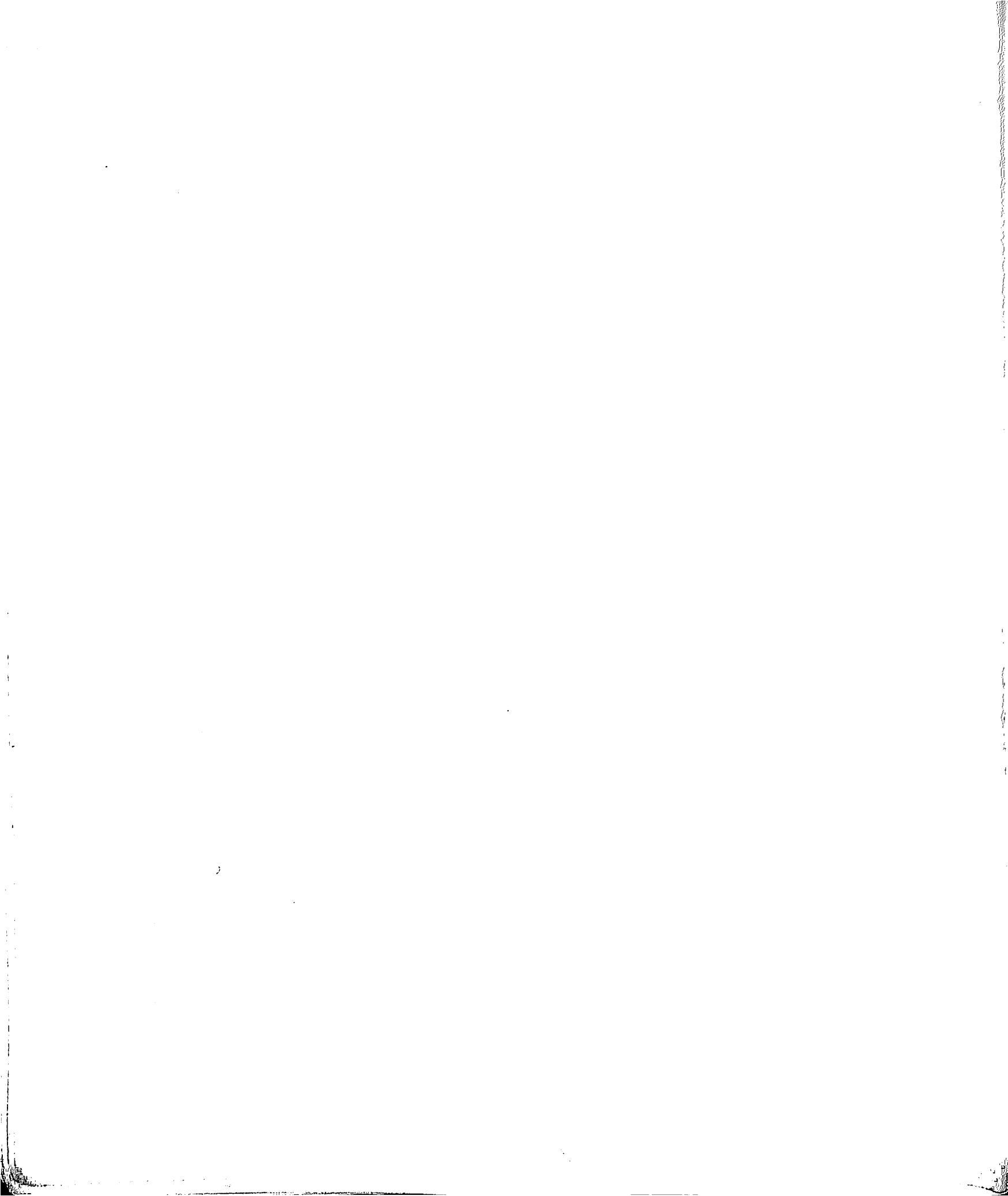
### NO DEBE

**NO DEBE** pasar por alto la útil información que se ofrece en estos cuadros.

Este libro utiliza varios tipos de letra para ayudarle a diferenciar entre el código de C++ y el español normal. El código de C++ está impreso en un tipo de letra especial conocido como monoespaciado. Los marcadores de posición —palabras o caracteres que se utilizan temporalmente para representar las palabras o caracteres reales que usted escribirá en el código— están impresos en *cursivas monoespaciadas*. Los términos nuevos o importantes están impresos en *cursivas*.

En los listados que se incluyen en este libro, cada línea de código real está numerada. Si en un listado ve una línea que no esté numerada, esto indica que la línea sin numeración es en realidad una continuación de la anterior línea de código numerada (algunas líneas de código son demasiado extensas para la anchura del libro). También verá un carácter de continuación de línea, como éste ➔. En este caso, debe escribir las dos líneas como una sola; no las divida.

Los listados también vienen incluidos en el CD-ROM con nombres de archivo que empiezan con 1st, seguido de un número de dos dígitos correspondiente a la lección, un guión corto y luego un número de 2 dígitos correspondiente al listado. Por ejemplo, el primer ejemplo del día 1 es 1st01-01.cxx



# SEMANA 1

## De un vistazo

Para empezar con la primera semana del aprendizaje de la programación en C++, necesitará unas cuantas cosas: un compilador, un editor y este libro. Si no cuenta con un compilador de C++ y un editor, de todas formas puede utilizar este libro, pero no obtendrá tanto provecho de él como lo haría si siguiera los ejercicios.

¡La mejor forma de aprender a programar es escribir programas! Al término de cada día encontrará un taller que contiene un cuestionario y algunos ejercicios. Asegúrese de contestar todas las preguntas, y de evaluar su trabajo tan objetivamente como le sea posible. Las lecciones posteriores amplían los conocimientos que usted obtiene en las primeras lecciones, por lo que debe asegurarse de entender el material completamente antes de avanzar.

## Una observación para los programadores de C

Tal vez el material de los primeros cinco días le sea familiar. Eche una ojeada a estos primeros días y haga los ejercicios para asegurarse de contar con los conocimientos necesarios para avanzar al día 6, "Clases base". Si conoce algo de C pero no lo ha utilizado en Linux, será mejor que lea detenidamente la introducción a los compiladores GNU que se da en las primeras dos lecciones.

## Objetivos

La primera semana cubre el material necesario para que usted empiece con la programación en general, y específicamente

1

2

3

4

5

6

7

**con C++.** En el día 1, "Comencemos", y en el día 2, "Los componentes de un programa de C++", se presentarán los conceptos básicos de la programación y el flujo de los programas. En el día 3, "Variables y constantes", aprenderá sobre estos elementos y como utilizar datos en sus programas. En el día 4, "Expresiones e instrucciones", aprenderá a ramificar los programas con base en los datos proporcionados y en las condiciones encontradas al ejecutar los programas. En el día 5, "Funciones", aprenderá lo que son las funciones y cómo se utilizan, y en el día 6, "Clases base", aprenderá acerca de las clases y los objetos. El día 7, "Más flujo de programa", le enseñará más cosas sobre el flujo del programa, y al finalizar la primera semana ya estará escribiendo verdaderos programas orientados a objetos.

# SEMANA 1

## DÍA 1

### Comencemos

¡Bienvenido a *Aprendiendo C++ para Linux en 21 Días!* Hoy iniciará el camino para convertirse en un hábil programador en C++. Aprenderá:

- Qué son los compiladores GNU y cómo se relacionan GNU y Linux
- Por qué C++ es el estándar predominante en el desarrollo de software
- Los pasos para desarrollar un programa de C++
- Los fundamentos sobre el uso de los compiladores GNU
- Cómo escribir, compilar y enlazar su primer programa funcional de C++

### Qué es GNU

El acrónimo GNU significa “GNU No es UNIX”. Es el nombre de una serie de paquetes de software útiles encontrados comúnmente en entornos UNIX que son distribuidos por el proyecto GNU en el MIT. Por lo general, los paquetes están disponibles sin costo en varios lugares en Internet (se cobran si usted quiere una copia en medio físico, como en disco flexible, cinta o CD-ROM). El desarrollo de los paquetes es un proceso cooperativo, y el trabajo es realizado por muchos voluntarios. Este esfuerzo es conducido principalmente por Richard M. Stallman (uno de los desarrolladores del editor EMACS).

**Linux es un sistema operativo muy parecido a UNIX (sin infringir las marcas registradas y los derechos reservados de las versiones comerciales de UNIX). Pero consiste básicamente en un kernel (el núcleo del sistema operativo).**

La mayoría de los comandos que se utilizan en Linux es en realidad parte del proyecto GNU patrocinado por la FSF (Fundación para el Software Libre). El 28% de una distribución típica de Linux es GNU, y sólo el 3% es verdaderamente Linux.

### Nota

Es verdad, el acrónimo GNU se refiere a sí mismo, pero fue creado de esa manera a propósito. Este tipo de designación tiene su historia en los productos relacionados con UNIX, pero no cubiertos por esta marca registrada. Un sistema operativo no comercial similar a UNIX fue XINU: "XINU No es UNIX".

De manera que, cuando piensa que está emitiendo comandos de Linux, en realidad está usando utilerías GNU. Las personas que trabajan en Linux no vieron una razón para duplicar el trabajo realizado por las personas de GNU. Ésta es parte de la filosofía de UNIX: reutilizar, no recrear.

Si analiza detenidamente la documentación incluida con su sistema Linux, debe encontrar alguna mención de GNU.

El proyecto GNU también incluye compiladores de C y C++ (junto con muchas otras herramientas, lenguajes y utilerías). El compilador de C se conoce como gcc, y el compilador de C++ se conoce como g++ (en Linux) o gxx (en otros sistemas operativos, como DOS, que no permiten el uso del signo de suma en los nombres de archivos).

Lo agradable acerca de las herramientas GNU es que las hay para muchas plataformas, no sólo para Linux. Así pues, puede obtener un compilador GNU para Linux, o para DOS, o para DEC (Compaq) Alpha, o para muchos otros sistemas.

Esto significa que puede aprender a utilizar un compilador y usarlo en muchos lugares. También puede tener un proyecto personal de C++ en Linux en el que trabaje por su cuenta cuando en realidad deba estar trabajando para su patrón, inclusive si sólo tiene acceso a un equipo basado en Microsoft Windows o DOS. Claro que, ¡usted nunca haría algo así!

## Cómo obtener las herramientas GNU

La manera más fácil de obtener el compilador GNU y otras herramientas es instalar la copia de Linux que viene en el CD-ROM que se incluye en este libro. Para obtener detalles sobre la instalación, tendrá que revisar las instrucciones que vienen en el CD-ROM, ya que este libro se enfoca en C++, no en Linux. Pearson Educación publica también algunos buenos libros sobre Linux (no estoy haciendo publicidad para esta compañía, ya que he contribuido en algunos capítulos en varios de esos libros).

Como lo mencioné anteriormente, también puede obtener el compilador para otras plataformas y sistemas operativos.

Si ya tiene Linux instalado, existe la posibilidad de que tenga también los compiladores GNU (pero tal vez tenga una versión antigua).

El mejor lugar para buscarlos (además del CD-ROM) es <http://www.gnu.org>. Allí usted puede descargar sin costo las herramientas y los compiladores GNU para varias plataformas. También puede obtener información relacionada con la compra del software de GNU en CD-ROM. De cualquier forma, debería considerar enviarles una donación. Después de todo, cuesta dinero mantener páginas Web e imprimir documentación. Yo ordené un juego completo de los archivos binarios y las herramientas para todas las plataformas, para compararlas. (Yo no obtengo dinero cuando usted compra o hace una donación a la Fundación para el Software Libre; simplemente creo en lo que están haciendo.)

Desarrollar código lleva tiempo. Y también lleva tiempo cortar, duplicar y entregar los CD-ROMs. Como resultado, existen diferentes versiones para diferentes plataformas, y diferentes versiones entre los medios de distribución. Puede descargar la versión más reciente o comprar una versión un poco menos reciente.

El CD-ROM que viene en este libro incluye la versión 2.9.5 de los compiladores GNU (la más reciente al momento de escribir este libro). La versión disponible en el CD-ROM de la FSF es la 2.7.2. En general, entre más nuevo sea el compilador, es mejor.

Los ejemplos que vienen en este libro se pueden compilar con ambas versiones, a menos que se indique lo contrario.

Las versiones más recientes del compilador son parte del egcs (Sistema Experimental del Compilador GNU). La gente de Cygnus hizo la mayor parte del trabajo para crear el nuevo compilador. Cygnus es la empresa oficial de mantenimiento para los compiladores GNU (bajo la supervisión de un comité de dirección). Cygnus Solutions fue adquirida por la empresa Red Hat, que tiene su propia distribución del sistema operativo Linux. Para obtener información de los compiladores GNU, visite el URL <http://www.redhat.com>.

## Una breve historia acerca de C++

Los lenguajes computacionales han experimentado una impresionante evolución desde que se construyeron las primeras computadoras electrónicas para ayudar en los cálculos sobre trayectorias de artillería, durante la segunda guerra mundial. En un principio, los programadores trabajaron con las instrucciones de computadora más primitivas: el lenguaje de máquina. Estas instrucciones se representaban con largas cadenas de unos y ceros. En poco tiempo se inventaron los ensambladores para asignar instrucciones de máquina a instrucciones nemotécnicas manejables y entendibles para los humanos, como ADD y MOV.

Con el tiempo surgieron los lenguajes de nivel más alto, como BASIC y COBOL. Estos lenguajes permitieron que los programadores trabajaran con algo que se aproximaba a palabras y oraciones, como Let I = 100 (Hacer que I = 100). Estas instrucciones se traducían a lenguaje de máquina por medio de los intérpretes y compiladores. Un *intérprete* traduce un programa a medida que lo lee, convirtiendo las instrucciones del programa, o código, directamente en acciones. Un *compilador* traduce el código a un formato intermedio. Este paso se conoce como compilar, y produce un archivo objeto. A continuación, el compilador invoca a un *enlazador*, el cual a su vez convierte uno o más archivos objeto en un programa ejecutable.

Debido a que los intérpretes leen el código en el orden en el que está escrito, y lo ejecutan al instante, los programadores pueden utilizarlos con facilidad. Los compiladores llevan a cabo los pasos adicionales de compilar y enlazar el código, lo cual es algo inconveniente. Sin embargo, los compiladores producen un programa que es muy veloz cada vez que se ejecuta, debido a que la tarea de traducir el código fuente a lenguaje de máquina, lo cual consume mucho tiempo, ya se ha realizado.

Otra ventaja de muchos lenguajes compilados, como C++, es que se puede distribuir el programa ejecutable a personas que no tengan el compilador. Con un lenguaje interpretado, es necesario tener el intérprete para ejecutar el programa.

Algunos lenguajes, como Visual Basic, llaman al intérprete biblioteca en tiempo de ejecución (runtime library). Java llama máquina virtual (VM) a su intérprete en tiempo de ejecución. Si ejecuta código de JavaScript, el navegador Web (como Internet Explorer o Netscape) proporciona la VM.

Durante muchos años, el principal objetivo de los programadores de computadoras fue escribir breves piezas de código que se ejecutaran rápidamente. El programa necesitaba ser pequeño, debido a que la memoria era costosa, y también necesitaba ser veloz, ya que el poder de procesamiento también era costoso. A medida que las computadoras se han vuelto más pequeñas, económicas y veloces, y a medida que el costo de la memoria se ha reducido, estas prioridades han cambiado. En la actualidad, el tiempo de un programador cuesta más que la mayoría de las computadoras utilizadas en las empresas. Un código bien escrito y fácil de mantener es imprescindible. “Fácil de mantener” significa que a medida que cambian los requerimientos de las empresas, se puede extender y mejorar el programa sin incurrir en costos demasiado altos.

## Programas

La palabra *programa* se utiliza de dos formas: para describir instrucciones individuales (o código fuente) creadas por el programador, y para describir una pieza completa de software ejecutable. Esta distinción puede ocasionar una enorme confusión, por lo que trataremos de distinguir entre el código fuente por una parte, y el ejecutable por la otra.

Un programa se puede definir ya sea como un conjunto de instrucciones creadas por un programador, o como una pieza ejecutable de software.

El código fuente se puede convertir en programa ejecutable de dos formas: los intérpretes convierten el código fuente en instrucciones para la computadora, y ésta ejecuta de inmediato esas instrucciones. De igual manera, los compiladores convierten el código fuente en un programa, el cual se puede ejecutar posteriormente. Aunque es más fácil trabajar con los intérpretes, la mayor parte de la programación se hace con compiladores, debido a que el código compilado se ejecuta mucho más rápido. C++ es un lenguaje compilado.

1

## Solución de problemas

Los problemas que los programadores deben resolver han estado cambiando. Hace 20 años se creaban programas para manejar grandes cantidades de información no procesada. Las personas que escribían el código y las personas que utilizaban el programa eran todos profesionales de la computación. En la actualidad, la cantidad de personas que utilizan computadoras es mucho mayor, y la mayoría conoce muy poco acerca de la forma en que trabajan las computadoras y los programas. Las computadoras son herramientas utilizadas por gente que está más interesada en resolver sus problemas de negocios que en batallar con ellas.

Irónicamente, los programas son cada vez más complejos para que sean más fáciles de usar para esta nueva generación. Han pasado ya los días en los que los usuarios escribían comandos crípticos en indicadores esotéricos, sólo para ver un flujo de información no procesada. Los programas de la actualidad utilizan “interfaces amigables para el usuario” complejas, que involucran múltiples ventanas, menús, cuadros de diálogo y la inmensidad de metáforas con las que nos hemos familiarizado. Los programas escritos para soportar este nuevo método son mucho más complejos que aquellos que fueron escritos hace sólo 10 años.

Con el desarrollo de Web, las computadoras han entrado a una nueva era de penetración de mercado; en la actualidad hay más personas que utilizan computadoras, y sus expectativas son muy altas. Durante los últimos años, los programas se han vuelto más grandes y más complejos, y se ha manifestado ya la necesidad de las técnicas de programación orientada a objetos para manejar esta complejidad.

A medida que han cambiado los requerimientos de programación, también han evolucionado las técnicas y los lenguajes utilizados para escribir programas. Aunque la historia completa es fascinante, este libro se enfoca en la transformación de la programación procedural a la programación orientada a objetos.

## Programación procedural, estructurada y orientada a objetos

Hasta hace poco, se pensaba que los programas eran una serie de procedimientos que actuaban sobre los datos. Un *procedimiento*, o función, es un conjunto de instrucciones específicas que se ejecutan una tras otra. Los datos estaban bastante separados de los procedimientos, y el truco de la programación era llevar el registro de cuáles funciones llamaban a cuáles otras funciones, y cuáles datos se cambiaban. Para que esta situación potencialmente confusa tuviera sentido, se creó la programación estructurada.

**La idea principal de la programación estructurada es tan simple como la idea de dividir y conquistar. Se puede pensar que un programa de computadora consiste en un conjunto de tareas. Cualquier tarea que sea demasiado compleja como para ser descrita, simplemente se divide en un conjunto de tareas más pequeñas, hasta que éstas sean lo suficientemente pequeñas e independientes como para entenderlas con facilidad.**

Como ejemplo, calcular el salario promedio de todos los empleados de una compañía es una tarea algo compleja. Sin embargo, se puede dividir en las siguientes subtareas

1. Averiguar el ingreso que obtiene cada persona.
2. Contar cuántas personas hay en la nómina.
3. Sumar todos los salarios.
4. Dividir el total entre el número de personas que hay en la nómina.

La suma de los salarios se puede dividir en los siguientes pasos:

1. Obtener el registro de cada empleado.
2. Obtener el salario.
3. Sumar el salario al total.
4. Obtener el registro del siguiente empleado.

A su vez, la obtención del registro de cada empleado se puede dividir en los siguientes pasos:

1. Abrir el archivo de empleados.
2. Ir al registro correcto.
3. Leer la información del disco.

La programación estructurada sigue siendo un método bastante exitoso para resolver problemas complejos. Sin embargo, a finales de los 80 se hicieron muy claras algunas deficiencias de la programación estructurada.

En primer lugar, es un deseo natural pensar en la información (por ejemplo, los registros de los empleados) y en lo que se puede hacer con la información (ordenar, editar, etcétera) como una sola idea. La programación procedural trabajaba en contra de esto, y separaba las estructuras de datos de las funciones que manipulaban esos datos.

En segundo lugar, los programadores tenían que reinventar constantemente nuevas soluciones para viejos problemas. Esto se conoce como “reinventar la rueda”, que es lo opuesto a la reutilización. La idea de la reutilización es crear componentes que tengan propiedades conocidas, y luego poder adaptarlos a su programa a medida que los necesita. Esto se inspira en el mundo del hardware: cuando un ingeniero necesita un transistor nuevo, por lo general no inventa uno; busca de entre todos los transistores existentes uno que funcione de la manera que necesita, o tal vez lo modifica. No existía una opción parecida para un ingeniero de software.

**TÉRMINO NUEVO** En la actualidad, la forma en la que utilizamos las computadoras, con menús, botones y ventanas, fomenta un método más interactivo y controlado por eventos para la programación de computadoras. *Controlado por eventos* significa que al ocurrir un evento (que el usuario haga clic en un botón o seleccione una opción de un menú) el programa debe responder. Los programas son cada vez más interactivos, y esto se ha convertido en algo importante a diseñar para ese tipo de funcionalidad.

Los programas antiguos obligaban al usuario a proceder paso por paso a través de una serie de pantallas. Los programas modernos controlados por eventos presentan todas las opciones al mismo tiempo y responden a las acciones de los usuarios.

**TÉRMINO NUEVO** La programación *orientada a objetos* trata de responder a esas necesidades, proporcionando técnicas para manejar la enorme complejidad, lograr la reutilización de componentes de software y acoplar los datos con las tareas que manipulan esos datos.

La esencia de la programación orientada a objetos es tratar a los datos y a los procedimientos que actúan sobre esos datos como un solo objeto (una entidad independiente con una identidad y ciertas características propias).

## C++ y la programación orientada a objetos

C++ soporta completamente la programación orientada a objetos, incluyendo los tres pilares del desarrollo orientado a objetos: encapsulación, herencia y polimorfismo.

### Encapsulación

Cuando un ingeniero necesita agregar una resistencia al dispositivo que está creando, por lo general no la construye partiendo desde cero. Busca entre todas las resistencias existentes, examina las bandas coloreadas que indican las propiedades y escoge la que necesita. La resistencia es una “caja negra” en lo que concierne al ingeniero; no le importa mucho cómo realiza su trabajo dicha caja mientras se ajuste a sus especificaciones; no necesita buscar dentro de la caja para utilizarla en su diseño.

La propiedad de ser una unidad independiente se conoce como *encapsulación*. Con esta propiedad podemos lograr el ocultamiento de los datos. El *ocultamiento de datos* es una característica altamente valorada con la que un usuario puede utilizar un objeto sin saber o preocuparse por la forma en que éste trabaja internamente. Así como usted puede utilizar un refrigerador sin saber cómo funciona el compresor, también puede utilizar un objeto bien diseñado sin conocer sus miembros de datos internos.

De la misma manera, cuando el ingeniero utiliza la resistencia, no necesita saber nada relacionado con el estado interno de ésta. Todas sus propiedades están encapsuladas dentro del objeto resistencia; no están distribuidas por todo el circuito. No es necesario entender cómo funciona la resistencia para utilizarla en forma efectiva. Sus datos están ocultos dentro de su cubierta protectora.

C++ soporta las propiedades de la encapsulación por medio de la creación de tipos definidos por el usuario, conocidos como clases. En el día 6, "Clases base", verá como crear clases. Después de crear una clase bien definida, ésta funciona como una entidad completamente encapsulada (se utiliza como una unidad completa). El funcionamiento interno de la clase debe estar oculto. Los usuarios de una clase bien definida no necesitan saber como funciona la clase; sólo necesitan saber cómo utilizarla.

## Herencia y reutilización

Cuando los ingenieros de Acme Motors quieren construir un nuevo auto, tienen dos opciones: pueden empezar desde cero, o pueden modificar un modelo existente. Tal vez su modelo Estrella sea casi perfecto, pero quieren agregarle un turbocargador y una transmisión de seis velocidades. El ingeniero en jefe preferiría no tener que empezar desde cero, sino decir: "Construyamos otro Estrella, pero agreguemosle estas capacidades adicionales. Llamaremos Quasar al nuevo modelo". Un Quasar es como un Estrella, pero mejorado con nuevas características.

C++ soporta la herencia. Se puede declarar un nuevo tipo que sea una extensión de un tipo existente. Se dice que esta nueva subclase se deriva del tipo existente, y algunas veces se conoce como tipo derivado. El Quasar se deriva del Estrella y por consecuencia hereda todas sus cualidades, pero se le pueden agregar más en caso de ser necesario. La herencia y su aplicación en C++ se tratan en el día 11, "Herencia", y en el día 15, "Herencia avanzada".

## Polimorfismo

El nuevo Quasar podría responder en forma distinta de un Estrella al oprimir el acelerador. El Quasar podría utilizar la inyección de combustible y un turbocargador, mientras que el Estrella simplemente alimentaría gasolina a su carburador. Sin embargo, un usuario no tiene que conocer estas diferencias. Simplemente puede "pisarle" y ocurrirá lo adecuado, dependiendo del auto que conduzca.

C++ soporta la idea de que distintos objetos hacen "lo adecuado", mediante lo que se conoce como *polimorfismo de funciones* y polimorfismo de clases. Poli significa muchos, y morfismo significa forma. Polimorfismo se refiere a que el mismo nombre toma muchas formas; este tema se trata en el día 10, "Funciones avanzadas", y en el día 13, "Polimorfismo".

# Cómo evolucionó C++

A medida que el análisis, el diseño y la programación orientados a objetos empezaron a tener popularidad, Bjarne Stroustrup tomó el lenguaje más popular para el desarrollo de software comercial, el lenguaje C, y lo extendió para proporcionar las características necesarias que facilitaran la programación orientada a objetos.

Aunque es cierto que C++ es un superconjunto de C, y que casi cualquier programa legítimo de C es un programa legítimo de C++, el salto de C a C++ es muy significativo. C++ se benefició de su relación con C durante muchos años debido a que los programadores de C podían utilizar C++ con facilidad. Sin embargo, para obtener el beneficio completo de C++, muchos programadores descubrieron que tenían que olvidarse de todo lo que habían aprendido y tenían que aprender una nueva forma de visualizar y resolver los problemas de programación.

1

## ¿Primero debo aprender C?

Esta pregunta surge inevitablemente: "Debido a que C++ es un superconjunto de C, ¿primero debería usted aprender C?" Stroustrup y la mayoría de los programadores de C++ están de acuerdo en que no sólo es innecesario aprender C primero, sino que también podría ser desventajoso.

En este libro no se da por hecho que usted tiene experiencia previa en programación. Sin embargo, si usted es programador de C, los primeros cinco capítulos de este libro son, en su mayor parte, un repaso, pero le indicarán también cómo utilizar los compiladores GNU. En el día 6 empezaremos con el verdadero trabajo del desarrollo de software orientado a objetos.

## C++ y Java

En la actualidad, C++ es el lenguaje predominante para el desarrollo de software comercial. Durante los últimos años Java ha desafiado ese dominio, pero el péndulo se balancea de nuevo a favor de C++, y muchos de los programadores que lo dejaron para utilizar Java están volviendo con C++. De cualquier forma, los dos lenguajes son tan similares que si se aprende uno es como si se aprendiera el 90% del otro.

## El estándar ANSI

El Comité de Estándares Acreditados, que opera bajo los procedimientos del ANSI (Instituto Estadounidense de Estándares Nacionales), ha creado un estándar internacional para C++.

El Estándar C++ se conoce ahora como Estándar ISO (Organización Internacional de Estándares), Estándar NCITS (Comité Estadounidense para Estándares de Tecnología de la Información), Estándar X3 (el antiguo nombre de NCITS) y Estándar ANSI/ISO. Este libro seguirá refiriéndose al estándar ANSI debido a que es el término más utilizado.

El estándar ANSI es un esfuerzo para asegurar que C++ sea portable (por ejemplo, garantizar que el código apegado al estándar ANSI que usted escriba para los compiladores GNU se compile sin errores en un compilador de cualquier otro fabricante, como Microsoft). Además, debido a que el código que se usa en este libro se apegue al estándar ANSI, se debe compilar sin errores en una Mac, en un equipo Windows o en un equipo Alpha.

Para la mayoría de los estudiantes de C++, el estándar ANSI será invisible. Este estándar ha sido estable desde hace tiempo, y los principales fabricantes lo manejan. Nos hemos asegurado de que todo el código que aparece en esta edición del libro se apegue al estándar ANSI.

## Prepárese para programar

C++ requiere, tal vez con más exigencia que otros lenguajes, que el programador diseñe el programa antes de escribirlo. Los problemas triviales, como los que se discuten en los primeros capítulos de este libro, no requieren de mucho diseño. Sin embargo, los problemas complejos, como los que tienen que enfrentar a diario los programadores profesionales, sí requieren del diseño, y entre más completo sea, existe una mayor probabilidad de que el programa resuelva, a tiempo y dentro del presupuesto, los problemas que debe resolver. Un buen diseño también ayuda a que un programa esté relativamente exento de errores y sea fácil de mantener. Se ha estimado que el 90% del costo del software es el costo combinado de la depuración y el mantenimiento. El que un buen diseño pueda reducir esos costos puede tener un impacto considerable en el costo total del proyecto.

Lo primero que debe preguntarse cuando se prepare para diseñar cualquier programa es: “¿Cuál es el problema que estoy tratando de resolver?” Todo programa debe tener un objetivo claro y bien definido, y usted descubrirá que incluso los programas más simples de este libro lo tienen.

La segunda pregunta que todo buen programador se hace es: “¿Se puede lograr esto sin recurrir a la escritura de software personalizado?” Reutilizar un viejo programa, usar pluma y papel (la manera antigua, original, manual y segura de hacer el trabajo) o comprar software de algún establecimiento son por lo general una mejor solución para un problema que escribir algo nuevo. El programador que pueda ofrecer estas alternativas nunca sufrirá por la escasez de trabajo; buscar soluciones menos costosas para los problemas de hoy siempre generará nuevas oportunidades más adelante.

Dando por hecho que entiende el problema y que requiere escribir un programa nuevo, ya está listo para empezar su diseño.

El proceso de entender completamente el problema (análisis) y de crear una solución (diseño) es el fundamento necesario para escribir una aplicación comercial de primera clase. Aunque, lógicamente, estos pasos vienen antes de la codificación (es decir, usted

debe entender el problema y diseñar la solución antes de implementarla), es mejor que aprenda la sintaxis fundamental y la semántica de C++ antes de aprender las técnicas de análisis y diseño formales.

1

## El entorno de desarrollo GNU/Linux

En este libro se da por hecho que usted utiliza los compiladores GNU en un entorno basado en texto, o una combinación similar. Es decir, su compilador debe tener un modo en el que usted pueda escribir directamente en la pantalla sin preocuparse por un entorno gráfico, como Windows o Macintosh. Este modo se conoce como modo de *consola* y es estándar para los compiladores GNU. Si trabaja en un entorno diferente, tendrá que buscar una opción, como *consola* o *easy window*, o revisar la documentación de su compilador. Si utiliza uno de los entornos gráficos en Linux, puede abrir una ventana de emulación de terminal (como xterm) para que pueda trabajar en modo de sólo texto.

Con GNU puede utilizar EMACS, vi o el editor de texto que prefiera. Si utiliza un compilador distinto, tendrá distintas opciones: su compilador puede tener su propio editor integrado o puede utilizar un editor de texto comercial para producir archivos de texto. Lo que cuenta es que, sin importar en dónde escriba su programa, éste debe tener capacidad para guardar archivos de texto plano, sin comandos de procesamiento de palabras incrustados en el texto. Algunos ejemplos de editores adecuados son el Bloc de notas (Notepad) de Windows, el comando *edit* de DOS, Brief, Epsilon, EMACS y vi. Muchos procesadores comerciales de palabras, como WordPerfect, Word y otros más, incrustan caracteres especiales pero también ofrecen un método para guardar archivos de texto plano, por lo que debe asegurarse de que la manera en que guarda su archivo es la adecuada.

Los archivos que usted crea con su editor se llaman archivos de código fuente, y para C++ por lo general se utiliza la extensión .cpp, .cp o .c. En este libro pusimos la extensión .cxx a todos los archivos de código fuente, debido a que los compiladores GNU aceptan esto como código fuente de C++ en distintas plataformas. Si usted utiliza algo diferente, revise qué extensiones necesita su compilador.

### Nota

Los compiladores GNU sí consideran importante la extensión de archivo. Debe utilizar .cxx o .c++.

Muchos otros compiladores de C++ no consideran importante la extensión. Si no especifica la extensión, muchos utilizarán .cpp de forma predeterminada. Sin embargo, tenga cuidado; algunos compiladores tratan a los archivos .c como código de C y a los archivos .cpp como código de C++. Nuevamente, revise su documentación.

DEBE	NO DEBE
<b>DEBE</b> utilizar un editor de texto plano (como vi, EMACS o incluso el comando edit de DOS) para crear su código fuente, o utilizar el editor integrado que tenga su compilador.	<b>NO DEBE</b> utilizar un procesador de palabras que guarde caracteres de formato especiales. Si va a utilizar un procesador de palabras, guarde el archivo como texto ASCII.
<b>DEBE</b> guardar sus archivos con la extensión .cxx o .c++.	
<b>DEBE</b> revisar el manual del compilador GNU para averiguar cómo compilar y enlazar correctamente sus programas.	

## Cómo compilar el código fuente

Aunque el código fuente de su archivo es algo criptográfico, y aunque cualquiera que no sepa C++ tendrá problemas para entender su función, de todas formas es lo que llamamos un formato entendible para los humanos. Su archivo de código fuente no es un programa, y no es posible ejecutarlo.

Para convertir su código fuente en programa, debe utilizar un compilador. La manera más sencilla de invocar el compilador g++ es la siguiente:

```
g++ archivo.c++ -o archivo
```

O si se encuentra en una plataforma que no soporte el signo de suma (+) en un nombre de archivo, como MS-DOS, puede utilizar lo siguiente:

```
gxx archivo.cxx -o archivo.exe
```

En ambos casos debe reemplazar *archivo* con el nombre que usted seleccione para su programa. Se producirá un archivo ejecutable con el nombre *archivo* o *archivo.exe*. Si omite la opción *-o*, se producirá un archivo llamado *a.out* (Linux) o *a.exe* (MS-DOS), respectivamente.

Si utiliza un compilador diferente, tendrá que revisar su documentación para determinar la forma de invocarlo y cómo indicarle en dónde encontrar su código fuente (esto varía de un compilador a otro).

Después de compilar su código fuente, se produce un archivo objeto. Por lo general, este archivo se nombra con la extensión .o (Linux) o .obj (MS-DOS). Sin embargo, éste todavía no es un programa ejecutable. Para convertir esto en programa ejecutable, debe ejecutar su enlazador.

1

## Cómo crear un archivo ejecutable con el enlazador

Por lo general, los programas de C++ se crean al enlazar uno o más archivos objeto con una o más bibliotecas. Una *biblioteca* es una colección de archivos que se pueden enlazar y que se proporcionan con su compilador, se compran por separado, o usted puede crearlos y compilarlos. Todos los compiladores de C++ vienen con una biblioteca de funciones (o procedimientos) y clases útiles que usted puede incluir en su programa. Una *función* es un bloque de código que realiza un servicio, como agregar dos números o imprimir en pantalla. Una *clase* es una colección de datos y funciones relacionadas; hablaremos mucho sobre las clases, empezando en el día 5, “Funciones”.

Los pasos para crear un archivo ejecutable son los siguientes:

1. Crear un archivo de código fuente, con una extensión .c++ o .cxx.
2. Compilar el código fuente en un archivo con el formato de objeto.
3. Enlazar su archivo objeto con cualquier biblioteca necesaria para producir un programa ejecutable.

Los compiladores GNU invocan automáticamente al enlazador (conocido como `ld`) para producir el archivo ejecutable.

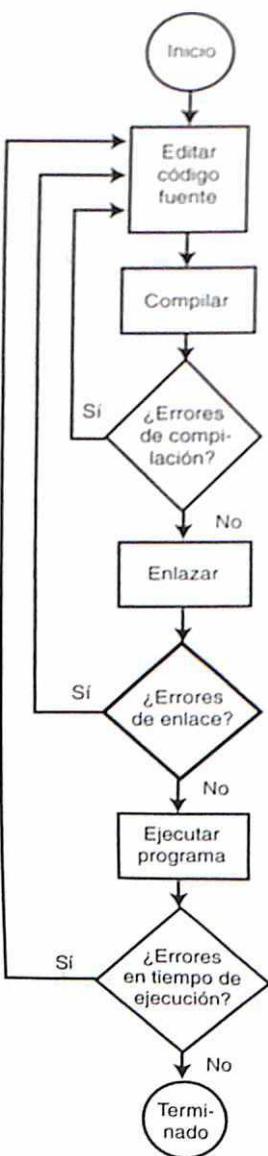
## El ciclo de desarrollo

Si todos los programas funcionaran la primera vez que se prueban, ese sería el ciclo completo de desarrollo: escribir el programa, compilar el código fuente, enlazar el programa y ejecutarlo. Desafortunadamente, casi cualquier programa, sin importar qué tan trivial sea, puede tener y tendrá errores, o *bugs*. Algunos errores ocasionarán que el compilador falle, otros, que el enlazador falle, y otros más aparecerán solamente cuando se ejecute el programa.

Cualquiera que sea el tipo de error que se encuentre, debe arreglarlo, y para eso necesita editar su código fuente, volver a compilar y enlazar, y luego volver a ejecutar el programa. Este ciclo se representa en la figura 1.1, que muestra en un diagrama los pasos del ciclo de desarrollo.

**FIGURA 1.1**

*Los pasos para el desarrollo de un programa de C++.*



## “¡Hola, mundo!”, su primer programa de C++

Los libros tradicionales de programación empiezan enseñándole cómo escribir las palabras “¡Hola, mundo!” en la pantalla, o una variación de esa instrucción. Esta consagrada tradición se sigue también en este libro.

Escriba el primer programa directamente en su editor, como se muestra en el listado 1.1. Después de escribirlo y estar seguro de que está correcto, guarde el archivo, compílelo, enlácelo y ejecútelo. Debe imprimir las palabras “¡Hola, mundo!” en la pantalla. No se preocupe mucho por la forma en que funciona; este ejemplo es sólo para que usted se familiarice con el ciclo de desarrollo. Todos los aspectos de este programa se tratarán en los siguientes dos días.

**Nota**

Todos los listados de este libro vienen en el CD-ROM para facilitarle las cosas. Tienen nombres de archivo en el formato `1stDD-NN.cxx`. `DD` es el número del día (01, 02, etcétera) y `NN` es el número del listado (01 para el primero del día, 02 para el segundo, y así sucesivamente).

El primer listado del libro (listado 1.1) es el archivo `1st01-01.cxx`.

No todos los archivos de listado se compilarán. Cuando éste sea el caso, se indicará en el texto.

1

**Precaución**

El siguiente listado, al igual que el resto de los listados de este libro, contiene números de línea a la izquierda. Estos números se utilizan sólo como referencia. No debe escribirlos. Por ejemplo, en la linea 1 del listado 1.1 deberá escribir:

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

**ENTRADA LISTADO 1.1 El programa "¡Hola, mundo!"**

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:     cout << "¡Hola, mundo!\n";
6:     return 0;
7: }
```

Utilice el archivo del listado que viene en el CD-ROM; si opta por escribir usted mismo este programa, asegúrese de escribirlo exactamente como se muestra. Ponga mucha atención en la puntuación. El símbolo `<<` de la línea 5 es el operador de inserción que, en los teclados que tienen la distribución de teclas en español, se obtiene al oprimir la tecla “`<`” que está a la izquierda de la “`Z`”. La línea 5 termina con un punto y coma (`;`). ¡No omita este símbolo!

Para compilar y enlazar este programa con el compilador GNU en Linux, debe escribir lo siguiente:

```
g++ 1st01-01.cxx -o 1st01-01
```

Si prefiere utilizar otro compilador, asegúrese de seguir correctamente las indicaciones. La mayoría de los compiladores enlaza automáticamente, pero de todas formas revise su documentación.

Si hay errores, examine cuidadosamente el código para ver en qué difiere del listado 1.1. Si ve un error en la línea 1, como `cannot find file iostream.h`, revise la documentación de su compilador para ver las indicaciones sobre la ruta de los archivos de encabezado (que se encuentran en el subdirectorio `include`) o las variables de entorno.

Si recibe un error que indique que no hay prototipo para `main` (de un compilador que no sea GNU), agregue la línea `int main();` antes de la línea 3. En todos los programas de este libro necesitará agregar esta línea antes del comienzo de la función `main`. La mayoría de los compiladores no requieren esta línea, pero algunos sí.

Si éste es el caso, su programa final debe lucir como el siguiente:

```
1: #include <iostream.h>
2: int main(); // la mayoria de los compiladores no necesitan esta linea
3: int main()
4: {
5:     cout << "¡Hola, mundo!\n";
6:     return 0;
7: }
```

Trate de ejecutar el programa `1st-01-01`; debe aparecer en su pantalla lo siguiente:

¡Hola, mundo!

Si es así, ¡felicidades! Acaba de escribir, compilar y ejecutar su primer programa de C++. Tal vez no parezca mucho, pero casi todos los programadores profesionales de C++ empiezan con este programa.

### Uso de las bibliotecas estándar

Para asegurar que los lectores que utilicen compiladores antiguos no tengan problemas con el código que viene en este libro, hemos utilizado los archivos de encabezado al estilo antiguo

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

en lugar de las nuevas bibliotecas estándar

```
# include <iostream>
```

Esto debe funcionar en todos los compiladores y tiene pocas desventajas. No obstante, si usted prefiere utilizar las nuevas bibliotecas estándar, sólo necesita cambiar el código a

```
# include <iostream>
```

y agregar la línea

```
using namespace std;
```

justo debajo de su lista de archivos de encabezado. El tema acerca del uso de namespace se detalla en el día 17, "Espacios de nombres".

Ya sea que utilice o no archivos de encabezado estándar, el código que viene en este libro se debe ejecutar sin necesidad de modificarlo. La principal diferencia entre las antiguas bibliotecas y la nueva biblioteca estándar es la biblioteca iostream (la cual se describe en el día 16, "Flujos"). Ni siquiera estos cambios afectan el código del libro; los cambios son sutiles, complejos y están más allá del alcance de una introducción elemental.

## Uso del compilador g++

Todos los programas de este libro se probaron con el compilador g++ de GNU, versión 2.9.5; muchos de ellos también se probaron con la versión 2.7.2. En teoría, debido a que es un código que se apega al estándar ANSI, todos los programas de este libro se deben ejecutar sin problemas en cualquier compilador compatible con ANSI, de cualquier fabricante.

En teoría, la teoría y la práctica son lo mismo. En la práctica, nunca es así.

Para que usted pueda poner manos a la obra, esta sección le presenta brevemente la forma de editar, compilar, enlazar y ejecutar un programa por medio del compilador GNU. Si utiliza un compilador distinto, los detalles de cada paso tal vez sean algo diferentes. Incluso si utiliza el compilador GNU versión 2.7.2 o 2.9.5, revise su documentación para averiguar cómo proceder a partir de aquí.

## Construcción del proyecto ¡Hola, mundo!

Para crear y probar el programa ¡Hola, mundo!, siga estos pasos:

1. Elija un editor y cree un archivo.
2. Escriba el código como se muestra en el listado 1.1 (o puede utilizar el editor para abrir el archivo `1st01-01.cxx` que se incluye en el CD-ROM).
3. Guarde el archivo y salga del editor.
4. Escriba el comando para compilación (`g++ 1st01-01.cxx -o 1st01-01` si utiliza Linux; para otros sistemas, consulte su documentación).
5. Escriba el nombre del archivo ejecutable para ejecutar el programa.

## Errores de compilación

Los errores en tiempo de compilación pueden ocurrir por muchas razones. Por lo general son el resultado de un error en la escritura o de cualquier otro pequeño error inadvertido. Los buenos compiladores (como los de GNU) no sólo le indican qué hizo mal, sino que también le indican el lugar exacto del código en donde cometió el error. Los mejores hasta le sugerirán un remedio!

Puede ver esto si coloca intencionalmente un error en su programa. Si el programa `¡Hola, mundo!` se ejecuta sin problemas, editelo ahora y quite la llave de cierre de la línea 7. Ahora su programa debe lucir como el del listado 1.2.

### ENTRADA LISTADO 1.2 Muestra de un error de compilación

```
1: # include <iostream.h>
2:
3: int main()
4: {
5:     cout << "¡Hola, mundo!\n";
6:     return 0;
```

Compile este programa y GNU le mostrará el siguiente error (se da por hecho que utilizó el listado `lst01-02.cxx`):

```
./lst01-02.cxx: In function 'int main()':
./lst01-02.cxx:7: parse error at end of input
```

Otros compiladores le mostrarán un mensaje de error como el siguiente:

```
lst01-01.cpp, line 7: Compound statement missing terminating; in
➥function main().
```

o tal vez un error como el siguiente:

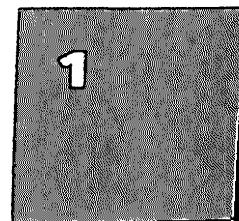
```
F:\Mcp\Tycpp21d\Testing\List0101.cpp(8) : fatal error C1004:
unexpected end of file found
Error executing cl.exe.
}
```

Este error le indica el archivo y el número de línea en el que está el problema, además de cuál es el problema (aunque confieso que es un poco críptico). Observe que el mensaje de error apunta a la línea 7. El compilador notará que hace falta una llave de cierre al encontrar el fin de archivo; es en esa línea donde se reconocerá el error. Algunas veces los errores sólo indican vagamente la razón del problema. Si un compilador pudiera identificar a la perfección cualquier problema, él mismo arreglaría el código.

## Resumen

Al terminar este día debe tener una buena comprensión de la forma en que evolucionó el lenguaje C++ y de los problemas para los que fue diseñado. Debe sentirse seguro de que aprender C++ es la elección correcta para cualquiera que esté interesado en la programación en estos tiempos. C++ proporciona las herramientas de la programación orientada a objetos y el rendimiento de un lenguaje de bajo nivel, lo que hace que sea el lenguaje de desarrollo preferido.

Hoy aprendió a escribir, compilar, enlazar y ejecutar su primer programa de C++, así como el concepto del ciclo normal de desarrollo de un programa. También aprendió un poco sobre los fundamentos de la programación orientada a objetos. Verá de nuevo estos temas durante las siguientes tres semanas. En la semana adicional aprenderá temas avanzados relacionados con la forma de trabajar con el conjunto de herramientas GNU y la programación en Linux.



## Preguntas y respuestas

- P ¿Cuál es la diferencia entre un editor de texto y un procesador de palabras?**
- R** Un editor de texto produce archivos que contienen texto plano. Un procesador de palabras no requiere comandos para dar formato ni cualquier otro símbolo especial. Los archivos de texto no tienen ajuste de línea automático, formato en negritas o cursivas, etcétera.
- P Si mi compilador tiene un editor integrado, ¿debo usarlo?**
- R** Los compiladores GNU no tienen editores de texto integrados. Linux viene con vi y EMACS. Casi todos los demás compiladores pueden compilar código producido por cualquier editor de texto. Sin embargo, una de las ventajas de utilizar el editor de texto integrado puede ser la capacidad para alternar rápidamente entre los pasos de edición y de compilación del ciclo de desarrollo. Los compiladores sofisticados incluyen un entorno de desarrollo completamente integrado, lo cual permite que el programador tenga acceso a archivos de ayuda, que edite y compile el código sin tener que cambiar de herramienta y que resuelva errores de compilación y de enlace sin tener que salir del entorno.
- P ¿Puedo ignorar los mensajes de advertencia de mi compilador?**
- R** Muchos libros dicen que sí, pero mi consejo es que no. Desde el primer día, usted debe formarse el hábito de tratar los mensajes de advertencias como errores. C++ utiliza el compilador para advertirle cuando haga algo que tal vez no sea lo que usted pretende. Preste atención a esas advertencias y haga lo necesario para que ya no aparezcan. Las advertencias significan que el compilador puede crear un archivo ejecutable a partir de su código fuente, ¡pero el compilador no cree que usted realmente quiera hacer lo que escribió en el código!

**P ¿Qué es tiempo de compilación?**

**R** El tiempo de compilación es cuando usted ejecuta su compilador, en contraste con el *tiempo de enlace* (cuando ejecuta el enlazador) o el *tiempo de ejecución* (cuando ejecuta el programa). Éstos son términos utilizados por el programador para identificar los tres tiempos en los que generalmente surgen los errores. Debido a que el compilador GNU invoca al enlazador si la compilación tiene éxito, tal vez no se distinga bien el tiempo de enlace (pero, créame, está ahí).

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, "Respuestas a los cuestionarios y ejercicios", y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Cuál es la diferencia entre un intérprete y un compilador?
2. ¿Cómo compila el código fuente con su compilador?
3. ¿Para qué sirve el enlazador?
4. ¿Cuáles son los pasos del ciclo normal de desarrollo?

### Ejercicios

1. Vea el siguiente programa y, sin ejecutarlo, trate de determinar lo que hace.

```
1: #include <iostream.h>
2: int main()
3: {
4:     int x = 5;
5:     int y = 7;
6:     cout << "\n";
7:     cout << x + y << " " << x * y;
8:     cout << "\n";
9:     return 0;
10:}
```

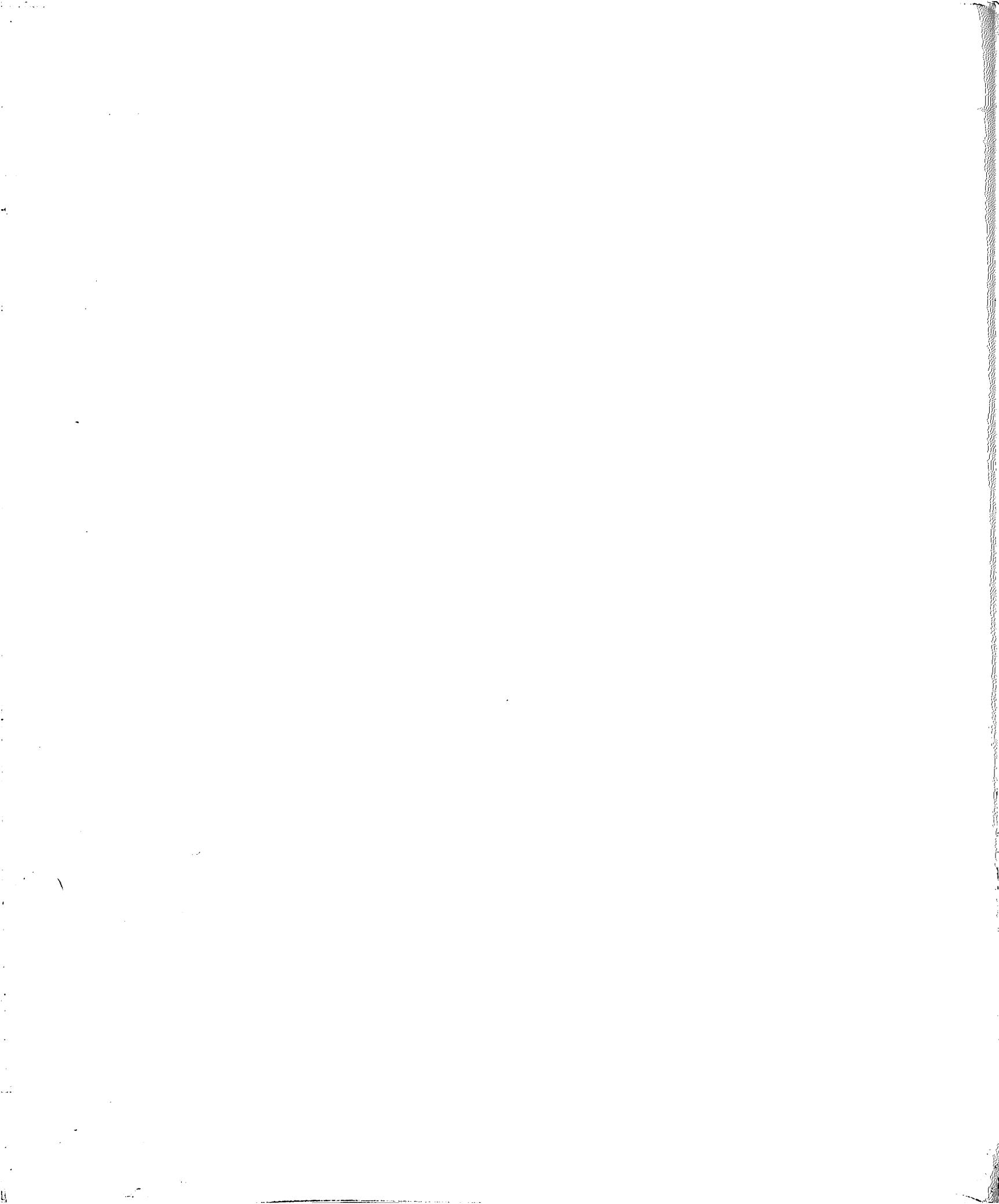
2. Escriba el programa del ejercicio 1, y luego completo y enlácelo. ¿Qué es lo que hace? ¿Hace lo que usted pensó?

3. Escriba el siguiente programa y compílelo. ¿Qué error aparece?

```
1: include <iostream.h>
2: int main()
3: {
4:     cout << "¡Hola, mundo!\n";
5:     return 0;
6: }
```

1

4. Corrija el error del programa del ejercicio 3 y vuelva a compilar, enlazar y ejecutar dicho programa. ¿Qué es lo que hace?



# SEMANA 1

## DÍA 2

### Los componentes de un programa de C++

Los programas de C++ constan de objetos, funciones, variables y otros componentes. La mayor parte de este libro se dedica a explicar con detalle estos componentes, pero para que usted pueda comprender cómo se integran todos estos componentes en un programa, debe ver uno que funcione y esté completo. Hoy aprenderá lo siguiente:

- Los componentes de un programa de C++
- Cómo funcionan en conjunto esos componentes
- Qué es una función y qué hace
- Las opciones que se pueden utilizar en el compilador g++

#### Un programa sencillo

Hasta el sencillo programa “¡Hola, mundo!” del día 1, “Comencemos”, tiene muchas partes interesantes. En esta sección se examina este programa con más detalle. Para su conveniencia, el listado 2.1 reproduce la versión original de “¡Hola, mundo!”

**ENTRADA** **L**ISTADO 2.1 Aquí se muestran los componentes de un programa de C++

```

1: #include <iostream.h>
2:
3: int main()
4: {
5:     cout << "¡Hola, mundo!\n";
6:     return 0;
7: }
```

**SALIDA** ¡Hola, mundo!**A**ANÁLISIS En la línea 1 se incluye el archivo `iostream.h` en el archivo actual.

He aquí la forma en que trabaja el preprocesador: el primer carácter es el símbolo `#` (de numeral), el cual es una señal para el preprocesador. Otro nombre para este signo es *gato*. El preprocesador se ejecuta cada vez que inicia su compilador. El preprocesador lee el código fuente en busca de líneas que inicien con `#` y actúa sobre esas líneas antes de que se ejecute el compilador. En el día 21, “Qué sigue”, se habla detalladamente sobre el preprocesador.

`include` es una instrucción del preprocesador que dice: “Lo que sigue es un nombre de archivo. Encuentre ese archivo y coloque aquí su contenido”. Los paréntesis angulares (`<` y `>`) que rodean al nombre de archivo le indican al preprocesador que busque este archivo en todas las ubicaciones usuales. Si su compilador está configurado en forma correcta, estos símbolos ocasionan que el preprocesador busque el archivo `iostream.h` en el directorio que contiene todos los archivos `.h` para el compilador (subdirectorios con el nombre *include*). El archivo `iostream.h` (flujo de entrada-salida) es utilizado por `cout`, el cual ayuda a escribir en la pantalla. El efecto de la línea 1 es incluir el archivo `iostream.h` en este programa como si usted lo hubiera escrito. El preprocesador se ejecuta antes que el compilador cada vez que éste es invocado. El preprocesador convierte cualquier línea que empiece con el signo `#` en un comando especial, con lo cual prepara su archivo de código para el compilador.

El programa en sí empieza en la línea 3 con una función llamada `main()`. Todo programa de C++ tiene una función `main()`. Una *función* es un bloque de código que realiza una o más acciones. Por lo general, las funciones son invocadas o llamadas por otras funciones, pero `main()` es especial. Al iniciar su programa, el sistema operativo llama automáticamente a `main()`.

`main()`, al igual que todas las funciones, debe declarar el tipo de valor que va a regresar. En el listado 2.1 (`lst02-01.cxx`), el tipo de valor de retorno para `main()` es `int`, lo que significa que, cuando termine, esta función regresará un entero al sistema operativo. En este caso, regresa el valor entero 0, como se muestra en la línea 6. Regresar un valor al

sistema operativo es una característica que en muchos sistemas operativos casi no tiene importancia y se utiliza poco, pero en Linux (y en todos los sistemas UNIX) se utiliza para conocer el estado con que finalizó un proceso o programa. El estándar de C++ requiere que `main()` sea declarada como se muestra.

**Nota**

Los compiladores de GNU y algunos otros le permitirán declarar a `main()` para que regrese el valor `void`. Esto ya no es C++ legítimo, y usted no debería formarse malos hábitos. Haga que `main()` regrese `int`, y que simplemente regrese un 0 como su última línea. Esto le indicará que el programa terminó sin problemas.

2

**Nota**

Algunos sistemas operativos (como Linux) le permiten probar el valor regresado por un programa. La convención es regresar 0 para indicar que el programa terminó en forma normal. Si utiliza el intérprete de comandos bash o pdksh, la variable de entorno `$?` contiene este valor de retorno. Si el valor regresado por el programa es diferente de 0, significa que hubo un problema. Utilice distintos valores enteros para catalogar el tipo de errores que ocurren en la ejecución de un programa.

Todas las funciones empiezan con una llave de apertura (`{`) y terminan con una llave de cierre (`}`). Las llaves de la función `main()` se encuentran en las líneas 4 y 7. Todo lo que está dentro de estas llaves se considera parte de la función.

El procesamiento principal de este programa está en la línea 5.

El objeto `cout` se utiliza para imprimir un mensaje en la pantalla. En el día 6, “Clases base”, se trata el tema de los objetos en general, y en el día 16, “Flujos”, se ve con detalle el tema relacionado con `cout` y su objeto relacionado `cin`. En C++, estos dos objetos, `cin` y `cout`, se utilizan para manejar la entrada (por ejemplo, desde el teclado) y la salida (por ejemplo, a la pantalla), respectivamente.

`cout` se utiliza de esta manera: se escribe la palabra `cout`, seguida del operador de inserción(`<<`). Cualquier cosa que esté después del operador de inserción se escribe en la pantalla. Si quiere escribir una cadena de caracteres, asegúrese de encerrarlos entre comillas dobles (“”), como se muestra en la línea 5.

Una cadena de texto es un conjunto de caracteres imprimibles.

Los dos últimos caracteres, `\n`, le indican a `cout` que debe insertar una nueva línea después de las palabras “¡Hola, mundo!” Este código especial se explica con detalle al hablar sobre `cout` en el día 17, “Espacios de nombres”.

La función `main()` termina en la línea 7 con la llave de cierre.

## Un vistazo breve a cout

En el día 16 verá la manera de utilizar el objeto `cout` para imprimir datos en la pantalla. Por ahora, puede utilizar este objeto sin necesidad de entender completamente como funciona. Para imprimir un valor en la pantalla, escriba la palabra `cout`, seguida del operador de inserción (`<<`), el cual se crea oprimiendo “`<`” dos veces. Aunque en realidad son dos caracteres, C++ los trata como si fueran uno.

Coloque sus datos después del operador de inserción. El listado 2.2 muestra como se utiliza esto. Escriba el ejemplo exactamente como está escrito, pero sustituya el nombre “Jesse Liberty” por el suyo.

### ENTRADA LISTADO 2.2 Uso de cout

```

1: // Listado 2.2 uso de cout
2: #include <iostream.h>
3: int main()
4: {
5:     cout << "Saludos a todos.\n";
6:     cout << "Aqui hay un 5: " << 5 << "\n";
7:     cout << "El manipulador endl escribe una nueva linea en la pantalla.";
8:     cout <<
9:         endl;
10:    cout << "Aqui hay un numero muy grande:\t" << 70000 << endl;
11:    cout << "Aqui esta la suma de 8 y 5:\t" << 8+5 << endl;
12:    cout << "Aqui hay una fraccion:\t\t" << (float) 5/8 << endl;
13:    cout << "Y un numero muy, muy grande:\t";
14:    cout << (double) 7000 * 7000 <<
15:        endl;
16:    cout << "No olvide reemplazar Jesse Liberty con su nombre... \n";
17:    cout << "¡Jesse Liberty es un programador de C++! \n";
18:    return 0;
19: }
```

### SALIDA

Saludos a todos.  
Aquí hay un 5: 5  
El manipulador endl escribe una nueva linea en la pantalla.  
Aqui hay un numero muy grande: 70000  
Aqui esta la suma de 8 y 5: 13  
Aqui hay una fraccion: 0.625  
Y un numero muy, muy grande: 4.9e+07  
No olvide reemplazar Jesse Liberty con su nombre...  
¡Jesse Liberty es un programador de C++!

**Nota**

Algunos compiladores (distintos de g++) emiten un mensaje de error que indica que se coloque la suma entre paréntesis antes de pasarla a cout. En este caso, la linea 11 quedaría de la siguiente manera:

```
11: cout << "Aqui está la suma de 8 y 5:\t" << (8+5) << endl;
```

**ANÁLISIS**

En la línea 2, la instrucción `#include <iostream.h>` ocasiona que se agregue el archivo `iostream.h` a su código fuente. Esto es necesario si usa cout y sus funciones relacionadas.

En la línea 5 está el uso más sencillo de cout: imprimir una cadena o serie de caracteres. El símbolo `\n` es un carácter de formato especial, el cual le indica a cout que imprima un carácter de nueva línea en la pantalla.

En la línea 6 se pasan tres valores a cout, y cada valor está separado por el operador de inserción. El primer valor es la cadena "Aqui hay un 5: ". Observe el espacio que está después de los dos puntos. Éste es parte de la cadena. A continuación, el valor 5 se pasa al operador de inserción y al carácter de nueva línea (siempre encerrado entre comillas dobles o sencillas). Esto ocasiona que la línea

**Aquí hay un 5: 5**

se imprima en la pantalla. Como no hay un carácter de nueva línea después de la primera cadena, el siguiente valor se imprime inmediatamente después. Esto se conoce como *concatenación* de los dos valores.

En la línea 7 se imprime un mensaje informativo, y luego se utiliza el manipulador endl. El propósito de endl es escribir una nueva línea en la pantalla. (En el día 16 verá otros usos para endl.)

2

**Nota**

endl significa *fin de línea* (end of line), y el último carácter es una letra e, no un uno.

En la línea 10 se presenta un nuevo carácter de formato, `\t`. Este carácter inserta un carácter de tabulación y se utiliza en las líneas 10 a 13 para alinear la salida. La línea 10 muestra que no sólo se pueden imprimir enteros, sino también enteros largos. La línea 11 muestra que cout hará una suma simple. El valor 8 + 5 se pasa a cout, pero se imprime un 13.

En la línea 12, el valor 5/8 se inserta en cout. El término `(float)` le indica a cout que este valor se debe evaluar como su equivalente en decimal, por lo que se imprime una fracción en su representación decimal. En la línea 14 se da el valor `7000 * 7000` a cout.

y se utiliza el término (`double`) para indicarle a `cout` que recibirá un número de punto flotante con doble precisión. Todo esto se va a explicar en el día 3, "Variables y constantes", al tratar el tema de los tipos de datos.

En la línea 17 usted puso su nombre, y la salida confirma que, efectivamente, usted es un programador de C++. ¡Y debe de ser verdad, ya que la computadora lo dijo!

## Comentarios

Al escribir un programa, siempre es claro y evidente lo que uno trata de hacer. Y es gracioso que, un mes después, al volver a ver el programa, éste pueda ser algo confuso e incierto. En realidad no sé cómo surge esta confusión en su programa, pero siempre es así.

Para evitar cualquier confusión, y para ayudar a que otros entiendan el código que usted escribe, debe utilizar los comentarios. Los *comentarios* son texto que el compilador ignora, pero que puede informar al lector lo que usted está haciendo en algún punto específico del programa.

### Tipos de comentarios

Los comentarios de C++ pueden ser de dos formas: el comentario con doble barra diagonal (`//`), y el comentario con barra diagonal y asterisco (`/*`). El primero, que se conoce como comentario estilo C++ (o comentario corto), le indica al compilador que ignore todo lo que esté después de las dos barras diagonales (`//`) hasta el final de la línea.

El comentario con barra diagonal y asterisco le indica al compilador que ignore todo lo que esté después de la barra diagonal y el asterisco (`/*`), hasta que encuentre una marca de comentario con asterisco y barra diagonal (`*/`). Estas marcas se conocen como comentarios estilo C (o comentarios largos). Todos los `/*` deben tener un `*/` para cerrar el comentario.

Como puede imaginar, los comentarios estilo C se utilizan también en el lenguaje C, pero los comentarios estilo C++ no son parte de la definición oficial de C.

Muchos programadores de C++ utilizan el comentario estilo C++ la mayor parte del tiempo, y reservan los comentarios estilo C para apartar grandes bloques de un programa. Puede incluir comentarios estilo C++ dentro de un bloque de comentarios estilo C; todo lo que está entre las marcas de comentario estilo C, incluyendo los comentarios estilo C++, se ignora.

### Uso de comentarios

Como regla general, el programa debe tener comentarios al principio que indiquen lo que hace. Asimismo, cada función debe tener comentarios que expliquen su funcionamiento y los valores que regresa. Estos comentarios se deben actualizar cada vez que se hagan cambios al programa. Cuando menos se debe mantener un historial de cambios.

Es necesario nombrar las funciones de manera que no exista confusión en cuanto a lo que hacen, y rediseñar y rescribir las piezas de código confusas para que sean evidentes. En la mayoría de los casos, los comentarios son la excusa para que un programador flojo mantenga su código confuso.

Esto no es para sugerir que no se deben utilizar los comentarios, sólo que no se debe depender en exceso de ellos para clarificar el código confuso; en vez de eso, arregle el código. En resumen, escriba bien su código y utilice comentarios para complementar la comprensión.

El listado 2.3 muestra que el uso de los comentarios no afecta el procesamiento del programa o su salida.

**ENTRADA** **LISTADO 2.3** Uso de comentarios

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:     /* éste es un comentario
6:        y se extiende hasta la marca de cierre de
7:        comentario representada por un asterisco y una barra diagonal */
8:     cout << "¡Hola, mundo!\n";
9:     // este comentario termina al final de la linea
10:    cout << "¡Ese comentario terminó!\n";
11:
12:    // los comentarios con doble barra diagonal pueden ir solos en una linea
13:    /* igual que los comentarios con barra diagonal y asterisco */
14:    return 0;
15: }
```

**SALIDA** ¡Hola, mundo!
¡Ese comentario terminó!**ANÁLISIS** El compilador ignora completamente los comentarios de las líneas 5 a 7, así como los de las líneas 9, 12 y 13. El comentario de la línea 9 termina al finalizar la línea, pero los comentarios de las líneas 5 y 13 necesitan una marca de cierre de comentario.

## Un consejo final sobre los comentarios

Los comentarios que indican lo que es obvio son innecesarios. De hecho, pueden ser contraproducentes, ya que el código puede cambiar y tal vez el programador olvide actualizar el comentario. Lo que es obvio para una persona puede ser confuso para otra, por lo que se requiere de buen juicio al decidir si se van a incluir comentarios o no.

El caso es que los comentarios no deben decir *qué* está pasando; deben decir *por qué* está pasando.

## Funciones

Aunque `main()` es una función, es algo inusual. Para que una función sea útil, se debe llamar, o invocar, durante el curso del programa. `main()` es invocada por el sistema operativo.

Un programa se ejecuta línea por línea en el orden en que aparece en su código fuente, hasta que llega a una función. Entonces el programa se ramifica para ejecutar la función. Cuando la función termina, regresa el control a la línea de código que se encuentra inmediatamente después de la llamada a la función.

Una buena analogía para esto es el proceso de afilar su lápiz. Si está haciendo un dibujo y se rompe la punta de su lápiz, tiene que dejar de dibujar, ir a sacar punta al lápiz y luego regresar a dibujar. Cuando un programa necesita que se realice un servicio, puede llamar a una función para que realice el servicio y luego continuar cuando se termina de ejecutar la función. El listado 2.4 muestra esta idea.

---

**ENTRADA** **LISTADO 2.4** Muestra de una llamada a una función

```
1: #include <iostream.h>
2:
3: // función FuncionDeMuestra
4: // imprime un mensaje útil
5: void FuncionDeMuestra()
6: {
7:     cout << "Estamos dentro de FuncionDeMuestra\n";
8: }
9:
10: // función main - imprime un mensaje y luego
11: // llama a FuncionDeMuestra, luego imprime
12: // un segundo mensaje.
13: int main()
14: {
15:     cout << "Estamos dentro de main\n" ;
16:     FuncionDeMuestra();
17:     cout << "Estamos de regreso en main\n";
18:     return 0;
19: }
```

---

**SALIDA**

```
Estamos dentro de main
Estamos dentro de FuncionDeMuestra
Estamos de regreso en main
```

**ANÁLISIS**

La función `FuncionDeMuestra()` se define en las líneas 5 a 8. Al ser llamada, imprime un mensaje en la pantalla y luego regresa.

El programa en sí empieza en la línea 13. En la línea 15, `main()` imprime un mensaje que dice que se encuentra en `main()`. Después de imprimir el mensaje, en la línea 16 se llama a `FuncionDeMuestra()`. Esta llamada ocasiona que se ejecuten los comandos de `FuncionDeMuestra()`. En este caso, toda la función consiste en el código de la línea 7, el cual imprime otro mensaje. Al finalizar `FuncionDeMuestra()` (en la línea 8), regresa al lugar donde fue llamada. En este caso, el programa regresa a la línea 17, en donde `main()` imprime el último mensaje.

2

## Uso de funciones

Las funciones regresan ya sea un valor o un tipo `void` (vacío), lo que significa que no regresan ningún valor. Una función que realiza la suma de dos enteros podría regresar el resultado de la suma, por lo que se definiría para regresar un valor entero. Una función que sólo imprime un mensaje no tiene nada que regresar, por lo que se declararía para regresar `void`.

Las funciones se componen de un encabezado y un cuerpo. El encabezado consta del tipo de valor de retorno, el nombre de la función y los parámetros para esa función. Los parámetros para una función permiten que se pasen valores a esa función. Por lo tanto, si la función fuera a sumar dos números, los números serían los parámetros para la función. Un encabezado de función típico se vería así:

```
int Suma(int a, int b)
```

Un parámetro es una declaración del tipo de valor que se va a pasar; el valor real pasado por la función que hace la llamada se conoce como argumento. Muchos programadores utilizan estos dos términos, parámetros y argumentos, como sinónimos. Otros son cuidadosos en cuanto a su distinción técnica. En este libro se utilizan los dos términos indistintamente.

El cuerpo de una función consta de una llave de apertura, cero o más instrucciones y una llave de cierre. Las instrucciones son el trabajo que va a realizar la función. Una función puede regresar un valor por medio de la instrucción `return`. Esta instrucción también hace que la función termine. Si no coloca una instrucción `return` en su función, ésta regresará automáticamente `void` (ningún valor) al final de la función. El valor regresado debe ser del tipo declarado en el encabezado de la función.

### Nota

En el día 5, "Funciones", se trata con más detalle el tema de las funciones. Los tipos que puede regresar una función se tratan con más detalle en el día 3. La información que se proporciona hoy es para que usted obtenga un panorama general, pues en casi todos sus programas de C++ utilizará las funciones.

El listado 2.5 muestra una función que toma dos parámetros enteros y regresa un valor entero. Por ahora no se preocupe por la sintaxis ni por los detalles específicos sobre la forma de trabajar con valores enteros (por ejemplo, `int x`); eso se trata con detalle en el día 3.

**ENTRADA** **LISTADO 2.5** Muestra de una función sencilla

```

1:  #include <iostream.h>
2:  int Suma (int x, int y)
3:  {
4:
5:      cout << "En Suma(), se recibieron " << x << " y " << y << "\n";
6:      return (x+y);
7:  }
8:
9:  int main()
10: {
11:     cout << "¡Estoy en main()!\n";
12:     int a, b, c;
13:     cout << "Escriba dos números: ";
14:     cin >> a;
15:     cin >> b;
16:     cout << "\nLlamando a Suma()\n";
17:     c=Suma(a,b);
18:     cout << "\nDe regreso en main().\n";
19:     cout << "c contiene el número " << c;
20:     cout << "\nSaliendo...\n\n";
21:     return 0;
22: }
```

**SALIDA**

¡Estoy en main()!  
Escriba dos números: 3 5

Llamando a Suma()  
En Suma(), se recibieron 3 y 5

De regreso en main().  
c contiene el número 8  
Saliendo...

**ANÁLISIS**

La función `Suma()` se define en la línea 2. Toma dos parámetros enteros y regresa un valor entero. El programa en sí empieza en las líneas 9 y 11, en donde imprime un mensaje. El programa pide dos números al usuario (líneas 13 a 15). El usuario escribe los dos números, separados por un espacio, y luego oprime “Entrar”. En la línea 17, `main()` pasa los dos números escritos por el usuario como argumentos para la función `Suma()`.

El procesamiento se ramifica hacia la función `Suma()`, la cual empieza en la línea 2. Los parámetros `a` y `b` se imprimen y luego se suman. El resultado se regresa en la línea 6 y se asigna a la variable `c`, con lo cual termina la función; la función `main()` toma nuevamente el control.

En las líneas 14 y 15 se utiliza el objeto `cin` para obtener un número para las variables `a` y `b`, y se utiliza `cout` para escribir los valores en la pantalla. En los siguientes días verá con más detalle las variables, así como otros aspectos de este programa.

2

## Más acerca del compilador GNU

Siempre que tenga duda sobre la manera de hacer algo, debe revisar la documentación de su compilador. El compilador `g++` de GNU no es la excepción. Con el compilador se incluyen un manual en línea (conocido como páginas de manual) y archivos de información.

Puede tener acceso al manual en línea con sólo escribir lo siguiente:

```
man g++
```

O si se encuentra en MS-DOS

```
man gxx
```

Y puede revisar las páginas de manual más grande para `gcc` y `g++` con el siguiente comando:

```
man gcc
```

También puede revisar el archivo de información (`info`) combinado para `gcc` y `g++` con el siguiente comando:

```
info gcc
```

Si solicita información para `g++`, sólo obtendrá la página de manual (así que no desperdice su tiempo).

### Nota

`info` es la principal documentación en línea para Linux y las herramientas GNU. Está basado en hipertexto y puede producir salida en varios formatos. Puede obtener información rápida sobre esto con el siguiente comando:

```
man info  
o  
info
```

Las dos secciones siguientes proporcionan información acerca de las opciones de línea de comandos más comunes y algunos tips sobre el uso del compilador.

## Opciones del compilador GCC

Las opciones del compilador se especifican por medio de parámetros o argumentos de línea de comandos. Las opciones del compilador que se utilizan con más frecuencia se muestran en la tabla 2.1.

**TABLA 2.1** Opciones del compilador

Opción	Significado
(ninguna opción utilizada)	Compilar y enlazar el programa en la línea de comandos y producir un archivo ejecutable con nombre predeterminado ( <code>a.out</code> para Linux, <code>a.exe</code> para MS-DOS).
<code>-c</code>	Compilar pero no enlazar; crear archivos con extensión <code>.o</code> .
<code>-Dmacro=valor</code>	Definir <i>macro</i> dentro del programa como <i>valor</i> .
<code>-E</code>	Preprocesar, no compilar ni enlazar; la salida se envía a la pantalla.
<code>-g</code>	Incluir información de depuración en el archivo ejecutable.
<code>-Idir</code>	Incluir <i>dir</i> como directorio para buscar archivo de encabezado (en donde el nombre está entre < y >).
<code>-Ldir</code>	Incluir <i>dir</i> como directorio para buscar archivos de biblioteca (utilizados por el enlazador para resolver referencias externas); varios directorios se separan con punto y coma ( ; ).
<code>-lbiblioteca</code>	Incluir <i>biblioteca</i> al enlazar.
<code>-O</code>	Optimizar.
<code>-oarchivo</code>	Guardar el archivo ejecutable como <i>archivo</i> .
<code>-Wall</code>	Habilitar advertencias para todo el código que se deba evitar.
<code>-w</code>	Deshabilitar todos los mensajes de advertencia.

Puede utilizar varias opciones en cada línea de comandos de compilación. Debe insertar un espacio antes del comienzo de cada opción (antes del signo de resta). No debe dejar espacios entre la opción y sus argumentos (como entre `-o` y *archivo*, aunque no siempre es necesario).

## Tips sobre el compilador GNU

Es una buena idea utilizar la opción `-o` al crear un archivo ejecutable para poder especificar el nombre de dicho archivo (de no ser así, la siguiente compilación lo sobrescribirá con el nombre predeterminado).

De manera predeterminada, cualquier mensaje de error se dirige a la pantalla. Tal vez quiera redirigir los mensajes a un archivo para poder revisarlos con calma por medio de un editor de texto. Esto es especialmente útil cuando hay muchos errores. Puede redirigir la salida hacia un archivo de la siguiente manera:

```
g++ lst02-01.cxx -o lst02-01 > lst02-01.lst
```

Desde luego que la selección del nombre de archivo es cuestión de usted. A mi me gusta utilizar la extensión .lst.

A medida que construya bibliotecas de funciones (verá más sobre las funciones en el día 5), podrá compilarlas una vez y volver a utilizarlas. Si no cambia el código fuente, no hay necesidad de volver a compilar. La opción `-c` crea el archivo objeto intermedio compilado y con formato. Cuando quiera utilizar el código de un archivo compilado (el archivo objeto), sólo necesitará incluir el nombre de éste en la línea de comandos.

Como siempre, el tip más importante sobre cualquier compilador o herramienta de Linux es revisar las páginas del manual (o archivo de información). Revise las opciones, pruébelas y vea cómo se comportan. ¡Tal vez puedan ayudarle!

2

## Resumen

La dificultad para aprender un tema complejo, como la programación, está en que mucho de lo que usted aprende depende de todo lo demás que hay por aprender. Este capítulo presentó los componentes básicos de un programa sencillo de C++. También presentó el ciclo de desarrollo y muchos términos nuevos importantes.

## Preguntas y respuestas

**P** ¿Qué hace la directiva `#include`?

**R** Ésta es una directiva para el preprocesador, el cual se ejecuta cada vez que usted llama al compilador. Esta directiva en especial ocasiona que se lea el archivo que está después de la palabra `include`, como si se hubiera escrito en esa ubicación de su código fuente.

**P** ¿Cuál es la diferencia entre los comentarios estilo `//` y los comentarios estilo `/*-*/`?

**R** Los comentarios con doble barra diagonal (`//`) terminan al final de la línea. Los comentarios con barra diagonal y asterisco (`/*`) terminan hasta donde se encuentre una marca de cierre de comentario (`*/`). Recuerde, ni siquiera el fin de una función termina un comentario con barra diagonal y asterisco; debe colocar la marca de cierre de comentario, o se producirá un error en tiempo de compilación.

**P** ¿Cuál es la diferencia entre un buen comentario y un mal comentario?

**R** Un buen comentario le indica al lector por qué este código específico está haciendo una tarea determinada, o le explica qué está por hacer una sección de código. Un mal comentario vuelve a decir lo que está haciendo una línea específica de código. Las líneas de código se deben escribir de forma que hablen por sí solas. Leer la línea de código debería indicarle lo que éste hace sin necesitar un comentario.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Cuál es la diferencia entre el compilador y el preprocesador?
2. ¿Por qué es especial la función `main()`?
3. ¿Cuáles son los dos tipos de comentarios, y en qué se diferencian?
4. ¿Se pueden anidar los comentarios?
5. ¿Pueden los comentarios ser de más de una línea?

### Ejercicios

1. Escriba un programa que imprima en la pantalla el mensaje “Me gusta C++”.
2. Escriba el programa más pequeño que se pueda compilar, enlazar y ejecutar.
3. **CAZA ERRORES:** Escriba el siguiente programa y compílelo. ¿Por qué falla?  
¿Cómo puede arreglarlo?  
`1: #include <iostream.h>`  
`2: int main()`  
`3: {`  
`4: cout << ¿Hay un error aquí?;`  
`5: return 0;`  
`6: }`
4. Encuentre el error del ejercicio 3 y vuelva a compilar, enlazar y ejecutar el programa.

# SEMANA 1

## DÍA 3

### Variables y constantes

Los programas necesitan una manera de guardar la información que utilizan. Las variables y constantes ofrecen varias maneras de representar y manipular esa información.

Hoy aprenderá lo siguiente:

- Cómo declarar y definir variables y constantes
- Cómo asignar valores a las variables y manipular esos valores
- Cómo escribir en la pantalla el valor de una variable

#### Qué es una variable

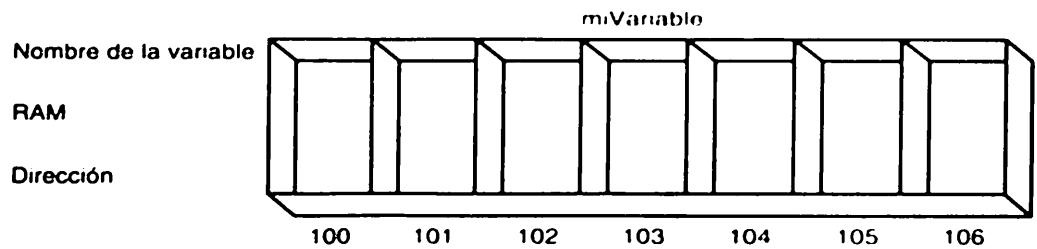
En C++, una *variable* es un lugar para guardar información. Una variable es una ubicación en la memoria de su computadora en la que puede guardar un valor, y desde la cual puede recuperar posteriormente ese valor.

Imagine que la memoria de su computadora es una serie de pequeñas casillas. Hay muchas casillas alineadas unas con otras. Cada casilla (o ubicación de memoria) está numerada en forma secuencial. Estos números se conocen como direcciones de memoria. Una variable reserva una o más casillas en las que usted puede guardar un valor.

El nombre de la variable (por ejemplo, `miVariable`) es una etiqueta colocada en una de estas casillas para que la pueda encontrar fácilmente sin necesidad de conocer su dirección de memoria. La figura 3.1 es una representación esquemática de esta idea. Como puede ver en la figura, `miVariable` empieza en la dirección de memoria 103. Dependiendo de su tamaño, `miVariable` puede ocupar una o más direcciones de memoria.

**FIGURA 3.1**

*Una representación esquemática de la memoria.*



### Nota

RAM significa memoria de acceso aleatorio. Al ejecutar un programa, éste se carga en RAM desde el archivo en disco. Todas las variables se crean también en RAM. Cuando los programadores hablan sobre la memoria, por lo general se refieren a la RAM.

Las variables también se conocen como *valores-i* (l-value), debido a que se pueden utilizar del lado izquierdo de un operador de asignación. El operador de asignación es el signo de igual (en la lección de hoy verá más acerca del operador de asignación, en la sección “Cómo asignar valores a sus variables”, y en el día 4, “Expresiones e instrucciones”, en la sección “Expresiones”). Las variables también se pueden utilizar del lado derecho del operador de asignación.

## Cómo reservar memoria

Al definir una variable de C++, debe indicarle al compilador de qué tipo es: entero, carácter, etc. Esta información le indica al compilador cuánto espacio debe reservar, así como el tipo de valor que usted quiere guardar en la variable.

Cada casilla es de 1 byte de longitud. Si el tipo de variable que usted desea crear es de 4 bytes de longitud, necesitará 4 bytes de memoria, o 4 casillas contiguas. El tipo de variable (por ejemplo, entero) le indica al compilador cuánta memoria (cuántas casillas) debe reservar para la variable.

Debido a que las computadoras utilizan bits y bytes para representar valores, y debido a que la memoria se mide en bytes, es importante que usted comprenda y se familiarice con estos conceptos. Para ver un repaso completo sobre este tema, lea el apéndice C, “Números binarios, octales y hexadecimales y una tabla de valores ASCII”.

## Cómo determinar el tamaño de los enteros y otros tipos de datos

En cualquier computadora, cada tipo de variable ocupa una sola cantidad invariable de espacio. Es decir, un entero podrá ser de 2 bytes en un equipo y de 4 en otro, pero siempre va a ser así en las dos computadoras.

Una variable de tipo `char` (utilizada para guardar caracteres) es, por lo general, de 1 byte de longitud.

En la mayoría de las computadoras, un entero corto es de 2 bytes de longitud, un entero largo es de 4 bytes (aunque los hay de 8 bytes), y un entero (sin las palabras reservadas `short` o `long`) puede ser de 2 o 4 bytes. El tamaño de un entero se determina según la computadora (si es de 16, 32 o 64 bits) y el compilador utilizados.

En computadoras personales modernas de 32 bits (Pentium o posteriores) que utilizan compiladores modernos (por ejemplo, `gcc` versión 2 o posterior), los enteros son de 4 bytes. En este libro se da por hecho que un entero ocupa 4 bytes, aunque este valor puede variar. El listado 3.1 le ayudará a determinar el tamaño exacto de estos tipos en su computadora.

3

Un carácter es una sola letra, número o símbolo que ocupa un byte de memoria.

### LISTADO 3.1 Cómo determinar el tamaño de los tipos de variables en su computadora

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:     cout << "El tamaño de un entero es:\t\t"      << sizeof(int)
6:     << " bytes.\n";
7:     cout << "El tamaño de un entero corto es:\t"    << sizeof(short)
8:     << " bytes.\n";
9:     cout << "El tamaño de un entero largo es:\t"    << sizeof(long)
10:    << " bytes.\n";
11:    cout << "El tamaño de un carácter es:\t\t"      << sizeof(char)
12:    << " bytes.\n";
13:    cout << "El tamaño de un punto flotante es:\t\t" << sizeof(float)
14:    << " bytes.\n";
```

continúa

**LISTADO 3.1** CONTINUACIÓN

```

10:    cout << "El tamaño de un doble es:\t"      << sizeof(double)
11:    << " bytes.\n";
12:    cout << "El tamaño de un booleano es:\t"    << sizeof(bool)
13:    << " bytes.\n";
14:    return 0;
15: }

```

**SALIDA**

El tamaño de un entero es:	4 bytes.
El tamaño de un entero corto es:	2 bytes.
El tamaño de un entero largo es:	4 bytes.
El tamaño de un carácter es:	1 bytes.
El tamaño de un punto flotante es:	4 bytes.
El tamaño de un doble es:	8 bytes.
El tamaño de un booleano es:	1 bytes.

**Nota**

Tal vez el número de bytes presentados sea diferente en su computadora.

**ANÁLISIS**

La mayor parte del listado 3.1 debe parecerle bastante familiar. La nueva característica es el uso de la función `sizeof()` en las líneas 5 a 11. Esta función es proporcionada por su compilador, y le indica el tamaño del objeto que se pasa como parámetro. Por ejemplo, en la línea 5 la palabra reservada `int` se pasa a `sizeof()`. Por medio de `sizeof()` yo pude determinar que en mi computadora un entero tiene la misma longitud que un entero largo, que es de 4 bytes.

**Uso de enteros con signo y sin signo**

Todos los tipos enteros vienen en dos variedades: con signo y sin signo. La idea aquí es que algunas veces se necesitan los números negativos, y otras no. Se sobreentiende que los enteros (cortos y largos) que no llevan la palabra “`unsigned`” (sin signo) tienen signo. Los enteros con signo pueden ser negativos o positivos. Los enteros sin signo siempre son positivos.

Debido a que usted dispone del mismo número de bytes para los enteros con signo y sin signo, el número más grande que puede guardar en un entero sin signo es dos veces más grande que el número positivo más grande que puede guardar en un entero con signo. Un entero corto sin signo puede manejar números desde 0 hasta 65,535. La mitad de los números representados por un entero corto con signo son negativos, por lo que este tipo de datos sólo puede representar números desde -32,768 hasta 32,767. Si esto le parece confuso, asegúrese de leer el apéndice C.

## Tipos de variables fundamentales

Hay muchos otros tipos de variables integrados en C++, los cuales se pueden dividir convenientemente en variables de tipo entero (de las que hemos hablado hasta ahora), variables de punto flotante y variables de tipo carácter.

Las variables de punto flotante tienen valores que se pueden expresar como decimales (es decir, son números racionales). Las variables tipo carácter almacenan un solo byte y se utilizan para guardar los 256 caracteres y símbolos de los conjuntos de caracteres ASCII y ASCII extendido.

El conjunto de caracteres ASCII es el conjunto de caracteres estandarizado para ser utilizado en las computadoras. ASCII es el acrónimo en inglés de Código Estándar Estadounidense para el Intercambio de Información. Casi cualquier sistema operativo de computadora soporta este conjunto de caracteres, aunque muchos también soportan otros conjuntos de caracteres internacionales.

Los tipos de variables utilizados en los programas de C++ se describen en la tabla 3.1. Esta tabla muestra el tipo de variable, cuánto espacio asume este libro que ocupa en memoria, y qué tipos de valores se pueden guardar en estas variables. Los valores que se pueden guardar se determinan según el tamaño de los tipos de variables, por lo que necesita comparar la salida que obtenga al ejecutar el programa del listado 3.1 con lo que viene en este libro.

3

**TABLA 3.1** Tipos de variables

Tipo	Tamaño	Valores
bool	1 byte	Verdadero (True) o Falso (False)
unsigned short int	2 bytes	0 hasta 65,535
short int	2 bytes	-32,768 hasta 32,767
unsigned long int	4 bytes	0 hasta 4,294,967,295
long int	4 bytes	-2,147,483,648 hasta 2,147,483,647
int (16 bits)	2 bytes	-32,768 hasta 32,767
int (32 bits)	4 bytes	-2,147,483,648 hasta 2,147,483,647
unsigned int (16 bits)	2 bytes	0 hasta 65,535
unsigned int (32 bits)	4 bytes	0 hasta 4,294,967,295
char	1 byte	256 valores de carácter, si tienen signo, entonces de -128 hasta 127; si no tienen signo, entonces de 0 hasta 255
float	4 bytes	-1.2e-38 hasta 3.4e38
double	8 bytes	-2.2e-308 hasta 1.8e308

 Nota

Los tamaños de las variables pueden ser distintos de los que se muestran en la tabla 3.1, dependiendo del compilador y de la computadora que este utilizando. Si al ejecutar el programa del listado 3.1 en su computadora la salida muestra los mismos valores que el libro, entonces la tabla 3.1 se aplica a su compilador y a su equipo. Si la salida que obtuvo es distinta de la que se muestra en el listado 3.1, entonces debe consultar las páginas del manual para saber los valores que sus tipos de variables pueden almacenar en su sistema.

Otro lugar en el que puede buscar es el archivo de encabezado `limits.h`.

## Definición de una variable

Usted crea o define una variable declarando su tipo, seguido de uno o más espacios, del nombre de la variable y un punto y coma. El nombre de la variable debe empezar con una letra o un guión bajo y puede contener casi cualquier combinación de letras, números y guiones bajos, pero no puede contener espacios. Algunos nombres de variables válidos son `x`, `J23qrsnf`, `mi_Edad`, y `miEdad`. Los buenos nombres de variables le indican el uso de la variable; usar buenos nombres facilita la comprensión del flujo de un programa. La siguiente instrucción define una variable de tipo entero llamada `miEdad`:

```
int miEdad;
```

 Nota

Al definir o declarar una variable, se asigna un espacio en memoria (se reserva) para esa variable. El valor de la variable será lo que estaba en ese espacio de memoria al momento de declarar la variable. En un momento verá cómo asignar un nuevo valor a ese espacio de memoria.

Las estructuras y las clases de objetos se comportan de una forma un poco distinta a las variables normales. Aprenderá esa diferencia cuando llegue a la lección de ese día. Puede definir o declarar una clase de objeto, pero no puede utilizar memoria. El espacio de memoria se asigna al momento de crear el objeto.

Como práctica general de programación, evite nombres horribles, como `J23qrsnf`, y restrinja los nombres de variables de una sola letra (como `x` o `i`) para las variables que utilice en raras ocasiones. Trate de utilizar nombres expresivos, como `miEdad` o `Cuantos`. Estos nombres serán más fáciles de comprender tres semanas después cuando se esté rascando la cabeza tratando de averiguar lo que quiso hacer al escribir esa línea de código.

Pruebe este experimento. Adivine lo que hacen estas piezas de código, basándose en las primeras líneas:

#### Ejemplo 1

```
int main()
{
    unsigned short x;
    unsigned short y;
    unsigned short z;
    z = x * y;
    return 0;
}
```

#### Ejemplo 2

```
int main()
{
    unsigned short Ancho;
    unsigned short Longitud;
    unsigned short Area;
    Area = Ancho * Longitud;
    return 0;
}
```

3

#### Nota

Si compila este programa, el compilador le advertirá que estos valores no están inicializados. Más adelante verá cómo solucionar este problema.

Evidentemente, el propósito del segundo programa es más fácil de adivinar, y la inconveniencia de tener que escribir los nombres de variables más largos queda más que recompensada por la facilidad de dar mantenimiento a este programa.

## Sensibilidad al uso de mayúsculas

C++ es sensible al uso de mayúsculas y minúsculas. En otras palabras, las letras mayúsculas y minúsculas se consideran distintas. Una variable llamada `edad` es diferente de `Edad`, la cual a su vez es diferente de `EDAD`.

#### Nota

Algunos compiladores le permiten desactivar la sensibilidad al uso de mayúsculas y minúsculas. Los compiladores GNU no lo permiten, por lo que no debe tratar de hacer esto.

Existen varias convenciones para nombrar variables, y aunque no importa mucho cuál método utilice, es importante ser consistente en todo el programa.

Muchos programadores prefieren utilizar sólo letras en minúscula para los nombres de sus variables. Si el nombre requiere dos palabras (por ejemplo, mi carro), se utilizan dos convenciones populares: `mi_carro` o `miCarro`. La última se conoce como *notación de camello* debido a que el uso de mayúsculas se parece a la joroba de un camello.

Algunas personas sienten que es más fácil leer el carácter de guión bajo (`mi_carro`), pero otras prefieren evitarlo porque piensan que es más difícil escribirlo. En este libro se utiliza la notación de camello, en la que la primera letra de la segunda palabra, y de todas las subsecuentes, se escribe con mayúscula: `miCarro`, `elZorroColorCafe`, etcétera.

### Nota

Muchos programadores avanzados emplean un estilo de notación que se conoce comúnmente como *notación húngara*. El objetivo de la notación húngara es poner un prefijo (un conjunto de caracteres) a las variables que describa su tipo. Las variables de tipo entero podrían empezar con una letra `i` minúscula, y los enteros largos podrían empezar con una `l` minúscula. Para las constantes, variables globales, apuntadores, etcétera, se utilizan otras notaciones. La mayor parte de esto es mucho más importante en la programación en C, debido a que C++ soporta la creación de tipos definidos por el usuario (vea el día 6, "Clases base"), y a que tiene muchos tipos.

## Palabras reservadas

Algunas palabras están reservadas para C++, y no se pueden utilizar como nombres de variables. Otro nombre para las *palabras reservadas* es palabras clave. Estas palabras son utilizadas por el compilador para controlar el programa. Algunas palabras reservadas son `if`, `while`, `for` y `main`. El manual del compilador debe proporcionar una lista completa, pero por lo general es muy poco probable que cualquier nombre razonable para una variable sea una palabra reservada. En el apéndice B, "Palabras reservadas de C++", hay una lista de palabras reservadas de C++.

DEBE	NO DEBE
<p><b>DEBE</b> definir una variable escribiendo el tipo y luego el nombre de la variable.</p> <p><b>DEBE</b> utilizar nombres de variables significativos.</p> <p><b>DEBE</b> recordar que C++ es sensible al uso de mayúsculas y minúsculas.</p> <p><b>DEBE</b> entender el número de bytes que ocupa en memoria cada tipo de variable, así como los valores que se pueden guardar en cada tipo de variable.</p>	<p><b>NO DEBE</b> utilizar palabras reservadas de C++ como nombres de variables.</p> <p><b>NO DEBE</b> utilizar variables sin signo para números negativos.</p>

**Nota**

Los nombres de las variables pueden contener palabras reservadas, siempre y cuando el nombre no conste sólo de esa palabra. `return_valor` es un nombre de variable válido, pero `return` no lo es.

## Cómo crear más de una variable a la vez

Puede crear más de una variable del mismo tipo en una sola instrucción escribiendo el tipo y luego los nombres de las variables separados por comas. Por ejemplo:

```
unsigned int miEdad, miPeso;      // dos variables enteras sin signo
long int area, ancho, longitud;   // tres enteros largos
```

Como puede ver, `miEdad` y `miPeso` se declaran como variables de tipo entero sin signo. La segunda línea declara tres variables de tipo entero largo individuales llamadas `area`, `ancho` y `longitud`. El tipo (`long`) se asigna a todas las variables, por lo que no se pueden mezclar tipos en una instrucción de definición.

**3**

## Cómo asignar valores a las variables

Puede asignar un valor a una variable por medio del operador de asignación (`=`). Por ejemplo, puede asignar un 5 a la variable `Ancho` escribiendo lo siguiente:

```
unsigned short Ancho;
Ancho = 5;
```

**Nota**

`long` es una versión abreviada de `long int` (entero largo), y `short` es una versión abreviada de `short int` (entero corto).

Puede combinar estos pasos e inicializar la variable `Ancho` al declararla escribiendo lo siguiente:

```
unsigned short Ancho = 5;
```

La inicialización es muy similar a la asignación, y con variables de tipo entero la diferencia es mínima. Más adelante, al tratar el tema de las constantes, verá que algunos valores deben ser inicializados debido a que no pueden ser asignados. La principal diferencia es que la inicialización ocurre en el momento en que se crea la variable.

Así como puede definir más de una variable a la vez, también puede inicializar más de una variable al mismo tiempo. Por ejemplo:

```
// crear dos variables de tipo long e inicializarlas
long ancho = 5, longitud = 7;
```

Este ejemplo inicializa la variable tipo `long int` llamada `ancho` con el valor 5 y la variable tipo `long int` llamada `longitud` con el valor 7. También se pueden mezclar definiciones e inicializaciones:

```
int miEdad = 39, tuEdad, suEdad = 40;
```

Este ejemplo crea tres variables de tipo `int`, e inicializa la primera y la tercera.

El listado 3.2 muestra un programa completo, listo para compilarse, que calcula el área de un rectángulo y escribe la respuesta en la pantalla.

### ENTRADA LISTADO 3.2 Una muestra del uso de variables

```
1: // Muestra de las variables
2: #include <iostream.h>
3:
4: int main()
5: {
6:     unsigned short int Ancho = 5, Longitud;
7:     Longitud = 10;
8:
9:     // crear una variable de tipo unsigned short e inicializarla con el
10:    // resultado de la multiplicación de Ancho por Longitud
11:    unsigned short int Area = (Ancho * Longitud);
12:
13:    cout << "Ancho:" << Ancho << "\n";
14:    cout << "Longitud: " << Longitud << endl;
15:    cout << "Area: " << Area << endl;
16:
17: }
```

### SALIDA

```
Ancho:5
Longitud: 10
Area: 50
```

### ANÁLISIS

En la línea 2, la directiva `include` solicita el uso de la biblioteca `iostream` para que `cout` pueda funcionar. El programa empieza en la línea 4.

En la línea 6, `Ancho` se define como entero corto sin signo, y se inicializa con el valor 5. También se define otro entero corto sin signo llamado `Longitud`, pero no se inicializa. En la línea 7 se asigna el valor 10 a `Longitud`.

En la línea 11 se define un entero corto sin signo llamado `Area`, y se inicializa con el valor obtenido de la multiplicación de `Ancho` por `Longitud`. En las líneas 13 a 15 se imprimen en pantalla los valores de las variables. Observe que la palabra especial `endl` crea una nueva línea.

## Uso de `typedef`

Escribir `unsigned short int` muchas veces puede ser tedioso, repetitivo y, lo que es peor, puede propiciar errores. C++ le permite crear un alias para esta frase mediante el uso de la palabra reservada `typedef`, que significa definición de tipo.

En efecto, está creando un sinónimo, y es importante distinguir esto de la creación de un nuevo tipo de datos (lo que hará en el día 6) pues `typedef` no crea un nuevo tipo de datos.

`typedef` se utiliza escribiendo la palabra reservada `typedef`, seguida del tipo existente, el nuevo nombre y, por último, un punto y coma. Por ejemplo,

```
typedef unsigned short int USHORT;
```

crea el nuevo nombre `USHORT` que usted puede utilizar en cualquier parte en la que necesite escribir `unsigned short int`. El listado 3.3 es una reproducción del listado 3.2, sólo que se utiliza la definición de tipo `USHORT` en lugar de `unsigned short int`.

3

### ENTRADA LISTADO 3.3 Una muestra de `typedef`

```
1: // ****
2: // Muestra de la palabra reservada typedef
3: #include <iostream.h>
4:
5: typedef unsigned short int USHORT;           //typedef definido
6:
7: int main()
8: {
9:     USHORT Ancho = 5;
10:    USHORT Longitud;
11:    Longitud = 10;
12:    USHORT Area = Ancho * Longitud;
13:    cout << "Ancho:" << Ancho << "\n";
14:    cout << "Longitud: " << Longitud << endl;
15:    cout << "Area: " << Area << endl;
16:    return 0;
17: }
```

### SALIDA

```
Ancho:5
Longitud: 10
Area: 50
```

### ANÁLISIS

En la línea 5, `typedef` define a `USHORT` como sinónimo de `unsigned short int`. El programa es muy parecido al del listado 3.2, y la salida es la misma.

## Cuándo utilizar short y cuándo utilizar long

Una fuente de confusión para los programadores principiantes de C++ es cuando declarar una variable como tipo `long` y cuándo declararla como tipo `short`. La regla, al comprenderla, es bastante clara: si hay alguna probabilidad de que el valor que quiera colocar en su variable sea demasiado grande para su tipo, entonces utilice un tipo más grande.

Como se muestra en la tabla 3.1, los enteros cortos sin signo (`unsigned short int`), asumiendo que sean de 2 bytes, pueden contener un valor de hasta 65,536 solamente. Los enteros cortos con signo (`short int`) dividen sus valores entre números positivos y negativos, y por consecuencia su valor máximo equivale sólo a la mitad del valor máximo de los enteros cortos sin signo.

Aunque los enteros largos sin signo (`unsigned long int`) pueden contener un número extremadamente grande (4,294,967,295), aún así son números finitos. Si necesita un número más grande, tendrá que utilizar `float` o `double`, pero en ese caso perdería algo de precisión. Los valores `float` y `double` pueden contener números extremadamente grandes, pero sólo los primeros 7 o 19 dígitos son significativos en la mayoría de las computadoras. Esto significa que el número se redondea después de esos dígitos.

Las variables más pequeñas ocupan menos memoria. En la actualidad, la memoria es económica y la vida es corta. Siéntase en libertad de utilizar `int`, el cual probablemente tendrá una longitud de 4 bytes en su equipo.

## Cómo sobregirar el valor de un entero sin signo

El hecho de que los enteros largos sin signo tengan un límite para los valores que pueden contener, raras veces es un problema, pero, ¿qué ocurre si se acaba el espacio?

Cuando un entero sin signo llega a su valor máximo, se regresa a cero, como lo hace el odómetro de un auto. El listado 3.4 muestra lo que ocurre si trata de colocar un valor demasiado grande en un entero corto.

**ENTRADA** **LISTADO 3.4** Una muestra de lo que ocurre al colocar un valor demasiado grande en un entero corto sin signo

---

```
1: #include <iostream.h>
2: int main()
3: {
4:     unsigned short int numeroChico;
5:     numeroChico = 65535;
6:     cout << "número chico:" << numeroChico << endl;
7:     numeroChico++;
8:     cout << "número chico:" << numeroChico << endl;
9:     numeroChico++;
10:    cout << "número chico:" << numeroChico << endl;
11:    return 0;
12: }
```

**SALIDA**

```
número chico:65535  
número chico:0  
número chico:1
```

**ANÁLISIS**

En la línea 4 se declara `numeroChico` como entero corto sin signo, que en mi computadora es una variable de 2 bytes capaz de contener un valor entre 0 y 65.535.

En la línea 5 se asigna el valor máximo a `numeroChico`, y se imprime en la línea 6.

En la línea 7 se incrementa `numeroChico`; es decir, se le agrega 1. El símbolo de incremento es `++` (como en el nombre C++, que es un incremento de C). Por consecuencia, el valor de `numeroChico` sería 65.536. Sin embargo, los enteros cortos sin signo no pueden contener un número más grande que 65.535, por lo que el valor se regresa a 0, que es lo que se imprime en la línea 8.

En la línea 9 se incrementa `numeroChico` otra vez, y luego se imprime su nuevo valor, 1.

3

## Cómo sobregirar el valor de un entero con signo

Un entero con signo difiere de un entero sin signo en que la mitad de los valores que se pueden representar son negativos. En lugar de imaginar un odómetro de auto tradicional, imagine uno que gira en aumento para los números positivos y disminuye para los negativos. Una milla a partir de cero puede ser 1 o -1. Al agotarse los números positivos, llega a los números más negativos (el valor absoluto más grande) y luego regresa a 0. El listado 3.5 muestra lo que ocurre al sumar 1 al número positivo máximo que puede contener un entero corto.

**ENTRADA**

**LISTADO 3.5** Una muestra de lo que ocurre al colocar un número demasiado grande en un entero con signo

```
1: #include <iostream.h>  
2: int main()  
3: {  
4:     short int numeroChico;  
5:     numeroChico = 32767;  
6:     cout << "número chico:" << numeroChico << endl;  
7:     numeroChico++;  
8:     cout << "número chico:" << numeroChico << endl;  
9:     numeroChico++;  
10:    cout << "número chico:" << numeroChico << endl;  
11:    return 0;  
12: }
```

**SALIDA**

```
número chico:32767  
número chico:-32768  
número chico:-32767
```

**ANÁLISIS**

En la línea 4 se declara `numeroChico`, pero esta vez como entero corto con signo (si no dice en forma explícita que es sin signo, se da por hecho que es con signo). El programa funciona en forma muy parecida al anterior, solo que la salida es bastante diferente. Para comprender completamente esta salida, debe familiarizarse con la forma en que se representan como bits los números con signo en un entero de 2 bytes.

No obstante, lo importante es que al igual que un entero sin signo, el entero con signo se regresa de su valor positivo más alto a su valor más negativo (el valor absoluto más alto).

## Uso de variables de tipo carácter

Las variables de tipo carácter (tipo `char`) son generalmente de 1 byte, tamaño suficiente para contener 256 valores (vea el apéndice C). Un `char` se puede interpretar como un número pequeño (desde -128 hasta 127 o, si no tiene signo, desde 0 hasta 255) o como un miembro del conjunto ASCII (Código Estándar Estadounidense para el Intercambio de Información). El conjunto de caracteres ASCII y su equivalente ISO (Organización Internacional de Estándares) son una manera de codificar todas las letras, números y signos de puntuación.

**Nota**

Las computadoras no conocen letras, puntuación ni oraciones. Todo lo que entienden son números. De hecho, todo lo que saben es si hay una cantidad suficiente de energía en una unión específica de cables. Si es así, esto se representa internamente como 1; si no, se representa como 0. Mediante la agrupación de ceros y unos, la computadora es capaz de generar patrones que se puedan interpretar como números, y éstos a su vez se pueden asignar a letras y signos de puntuación.

Su programa también se almacena en memoria como un conjunto de ceros y unos; la computadora sabe cómo interpretarlos apropiadamente.

En el código ASCII, la letra "a" minúscula tiene el valor 97. Todas las letras mayúsculas y minúsculas, todos los números y signos de puntuación tienen valores asignados entre 0 y 127. Se reservan 128 signos y símbolos adicionales para el fabricante de computadoras, aunque el conjunto de caracteres extendido de IBM casi se ha convertido en un estándar.

**Nota**

ASCII se pronuncia "Aski".

**Nota**

Linux está basado en ASCII; los sistemas operativos más antiguos (como los de mainframes IBM y otros) utilizan el conjunto de caracteres EBCDIC (Código Extendido de Caracteres Decimales Codificados en Binario para el Intercambio de Información), el cual está relacionado con la forma en que se perforan los agujeros de las tarjetas Hollerith (un método para introducir datos que espero usted nunca tenga que ver, excepto en un museo).

## Los caracteres como números

Al colocar un carácter, por ejemplo una “a”, en una variable `char`, lo que se tiene realmente es un número entre -128 y 127. Algunos sistemas utilizan caracteres sin signo, que son números entre 0 y 255. Sin embargo, el compilador sabe cómo traducir un carácter (representado por una comilla sencilla y luego una letra, número o signo de puntuación, seguido de otra comilla sencilla) a uno de los valores ASCII, o viceversa.

La relación valor/letra es arbitraria; no hay una razón específica para que la letra “a” tenga asignado el valor 97. Mientras todos (su teclado, compilador y pantalla) estén de acuerdo, no hay problema. No obstante, es importante tener en cuenta que existe una gran diferencia entre el valor 5 y el carácter “5”. Este último tiene en realidad el valor 53, así como la letra “a” tiene el valor 97.

El listado 3.6 muestra que los caracteres están guardados como números en la memoria.

3

**ENTRADA****LISTADO 3.6** Cómo imprimir caracteres con base en los números

```

1: #include <iostream.h>
2: int main()
3: {
4:     for (int i = 32; i<128; i++)
5:         cout << (char) i;
6:     return 0;
7: }
```

**SALIDA**

```
!"#$%'()*+,. /0123456789:;,>?@ABCDEFGHIJKLMOP
_QRSTUVWXYZ[\]^_`abcdefghijklmnoprstuvwxyz{}~-
```

**ANÁLISIS**

Este programa sencillo imprime los valores de los caracteres para los enteros del 32 al 127.

En el día 12, “Arreglos, cadenas tipo C y listas enlazadas”, se proporciona más información sobre la forma de combinar caracteres en arreglos y cadenas para formar cosas tales como palabras.

## Caracteres de impresión especiales

El compilador de C++ reconoce algunos caracteres especiales para dar formato. La tabla 3.2 muestra los más comunes. Puede colocarlos en su código escribiendo la barra diagonal inversa (conocida como carácter de escape), seguida del carácter. Por ejemplo, para colocar un carácter de tabulación en su código, debe escribir una comilla sencilla, una barra diagonal inversa, la letra t y, para terminar, una comilla sencilla:

```
char caracterTab = '\t';
```

Este ejemplo declara una variable `char` (`caracterTab`) y la inicializa con el valor de tipo carácter `\t`, el cual se reconoce como tabulador. Los caracteres de impresión especiales se utilizan para imprimir en la pantalla, en un archivo o en cualquier otro dispositivo de salida.

Un carácter de escape cambia el significado del carácter que lo sigue. Por ejemplo, normalmente el carácter `n` representa la letra `n`, pero cuando se pone después del carácter de escape (`\`), representa una nueva línea.

**TABLA 3.2** Los caracteres de escape

Carácter	Significado
<code>\n</code>	Nueva línea
<code>\t</code>	Tabulador
<code>\b</code>	Retroceso
<code>\\"</code>	Doble comilla
<code>\'</code>	Comilla sencilla
<code>\?</code>	Signo de interrogación
<code>\\\</code>	Barra diagonal inversa

## Uso de constantes

Al igual que las variables, las constantes son lugares para guardar datos. A diferencia de las variables, y como el nombre lo indica, las constantes no cambian. Debe inicializar una constante al crearla, y no puede asignar un nuevo valor después.

### Constantes literales

C++ tiene dos tipos de constantes: literales y simbólicas.

Una constante literal es un valor escrito directamente en el lugar de su programa donde se necesite, por ejemplo:

```
int miEdad = 39;
```

`miEdad` es una variable de tipo `int`; `39` es una constante literal. No puede asignar un valor a `39`, y no puede cambiar su valor. Es un *valor-d* (r-value) debido a que sólo puede aparecer del lado derecho de una instrucción de asignación.

## Constantes simbólicas

Una constante simbólica es una constante que está representada por un nombre, así como se representa a una variable. Sin embargo, a diferencia de una variable, después de inicializar una constante no se puede cambiar su valor.

Si su programa tiene una variable de tipo entero llamada `estudiantes` y otra llamada `clases`, usted puede calcular la cantidad de estudiantes, dado un número conocido de `clases`, si sabe que cada clase consta de 15 estudiantes:

```
estudiantes = clases * 15;
```

### Nota

El símbolo `*` denota una multiplicación.

3

En este ejemplo, `15` es una constante literal. Su código sería más fácil de leer y de mantener si substituye una constante simbólica por este valor:

```
estudiantes = clases * estudiantesPorClase
```

Si después decidiera cambiar el número de estudiantes que hay en cada clase, podría hacerlo en donde está definida la constante `estudiantesPorClase` sin tener que hacer un cambio en cada lugar en que utilizó ese valor.

Hay dos formas de declarar una constante simbólica de C++. La forma antigua, tradicional y ya obsoleta es usando la directiva `#define` del preprocesador, y la otra utilizando la palabra reservada `const`.

### Definición de constantes con `#define`

Para definir una constante de la manera tradicional, usted escribiría lo siguiente:

```
#define estudiantesPorClase 15
```

Observe que `estudiantesPorClase` no es de ningún tipo específico (`int`, `char`, etcétera). `#define` hace una simple substitución de texto. Cada vez que el preprocesador vea la palabra `estudiantesPorClase` en el código, la substituirá por el número `15`.

Debido a que el preprocesador se ejecuta antes que el compilador, éste nunca verá la constante; verá el número `15`.

Sólo porque esta manera de definir constantes es vieja y obsoleta no significa que usted no deba comprenderla; muchos programadores crecieron utilizando esta directiva, y existen muchas líneas de código que la utilizan.

### Definición de constantes con const

Aunque `#define` funciona, en C++ existe una nueva y mejor manera de definir constantes:

```
const unsigned short int estudiantesPorClase = 15;
```

Este ejemplo también declara una constante simbólica llamada `estudiantesPorClase`, pero esta vez dicha constante sí tiene tipo: `unsigned short int`. Este método tiene varias ventajas en cuanto a facilitar el mantenimiento del código y prevenir errores. La mayor diferencia es que esta constante tiene un tipo, y el compilador puede hacer que se utilice de acuerdo con su tipo.

#### Nota

Las constantes no se pueden cambiar mientras el programa se encuentre en ejecución. Por ejemplo, si quiere cambiar `estudiantesPorClase`, necesita cambiar el código y volver a compilar.

DEBE	NO DEBE
<b>DEBE</b> vigilar que los números no sobrepasen el tamaño del tipo entero y se establezcan en valores incorrectos.	<b>NO DEBE</b> utilizar el término <code>int</code> . Utilice <code>short</code> y <code>long</code> para dejar claro el tamaño del entero que desea utilizar.
<b>DEBE</b> dar a sus variables nombres significativos que reflejen su uso.	<b>NO DEBE</b> utilizar palabras reservadas como nombres de variables.

## Uso de constantes enumeradas

Las constantes enumeradas le permiten crear nuevos tipos y luego definir variables de esos tipos cuyos valores estén restringidos a un conjunto de valores posibles. Por ejemplo, puede declarar `COLOR` como una enumeración, y puede definir cinco valores para `COLOR`: ROJO, AZUL, VERDE, BLANCO y NEGRO.

La sintaxis para constantes enumeradas es escribir la palabra reservada `enum`, seguida por el nombre del tipo, una llave de apertura, cada uno de los valores válidos separados por comas y, finalmente, una llave de cierre y un punto y coma. He aquí un ejemplo:

```
enum COLOR { ROJO, AZUL, VERDE, BLANCO, NEGRO };
```

Esta instrucción realiza dos tareas:

1. Hace que `COLOR` sea el nombre de una enumeración, es decir, un tipo nuevo.

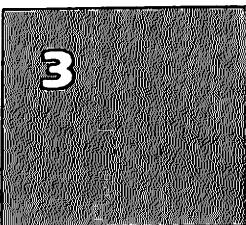
2. Hace que ROJO sea una constante simbólica de valor 0, AZUL una constante simbólica de valor 1, VERDE una constante simbólica de valor 2, y así sucesivamente.

Cada constante enumerada tiene un valor entero. Si no lo especifica de otra manera, la primera constante tendrá el valor 0, y a cada constante subsecuente se le irá asignando un valor igual a la constante anterior más uno. No obstante, cualquiera de las constantes puede ser inicializada con un valor específico, y las que no sean inicializadas tendrán un valor igual al de la constante anterior más uno. Por lo tanto, si escribe:

```
enum Color { ROJO=100, AZUL, VERDE=500, BLANCO, NEGRO=700 };
```

entonces ROJO tendrá el valor 100; AZUL el valor 101; VERDE el valor 500; BLANCO el valor 501; y NEGRO el valor 700.

Puede definir variables de tipo COLOR, pero sólo les puede asignar uno de los valores enumerados (en este caso, ROJO, AZUL, VERDE, BLANCO o NEGRO, o puede ser también 100, 101, 500, 501 o 700). Puede asignar cualquier valor de color a su variable COLOR. De hecho, puede asignar cualquier valor entero, incluso si no es un color válido, aunque un buen compilador mostrará un mensaje de advertencia si trata de hacer eso. Es importante tener en cuenta que las variables enumeradoras en realidad son de tipo `unsigned int`, y que las constantes enumeradas son iguales a las variables de tipo entero. Sin embargo, es muy conveniente poder nombrar estos valores al trabajar con colores, días de la semana o conjuntos similares de valores.



Usted no crea un nuevo tipo de datos con las variables enumeradas, sólo oculta los detalles de implementación. Como programador, usted utiliza palabras como ROJO o AZUL, pero el compilador sustituye los números por esas palabras en forma transparente para usted. Este proceso hace que la creación y la comprensión del programa sean mucho más sencillas.

El listado 3.7 presenta un programa que utiliza un tipo enumerado.

#### ENTRADA LISTADO 3.7 Una muestra de las constantes enumeradas

```
1: #include <iostream.h>
2: int main()
3: {
4:     enum Dias { Domingo, Lunes, Martes,
5:                 Miercoles, Jueves, Viernes, Sabado };
6:     int opcion;
7:     cout << "Escriba un dia (0-6): ";
8:     cin >> opcion;
9:     if (opcion == Domingo || opcion == Sabado)
10:        cout << "\nYa se le agotaron los fines de semana!\n";
11:     else
12:        cout << "\nEstá bien, incluiré un dia de descanso.\n";
13:     return 0;
14: }
```

**SALIDA**    **Escriba un dia (0-6): 6**  
*¡Ya se le agotaron los fines de semana!*

**ANÁLISIS**    En la línea 4 se define la constante enumerada DIAS con siete valores. Cada uno de éstos se evalúa como entero, contando en forma ascendente a partir de 0; por lo tanto, el valor de Martes es 2.

Se pide al usuario que proporcione un valor entre 0 y 6. El usuario no puede escribir la palabra “Domingo” cuando se le pide un día; el programa no sabe cómo traducir los caracteres que forman la palabra Domingo a uno de los valores enumerados. No obstante, usted puede comparar el valor proporcionado por el usuario con una o más de las constantes enumeradas, como se muestra en la línea 9. Aquí el uso de constantes enumeradas hace más explícita la intención de la comparación. Este mismo efecto se hubiera podido lograr mediante el uso de constantes de tipo entero, como se muestra en el listado 3.8, pero con un poco más de esfuerzo en la programación.

### Nota

Para este y todos los pequeños programas que vienen en este libro se ha pasado por alto todo el código que normalmente se escribiría para encargarse de lo que ocurre cuando el usuario escribe datos inapropiados. Por ejemplo, este programa no se asegura, como se haría en un programa real, de que el usuario escriba un número entre 0 y 6. Este detalle se ha pasado por alto para mantener estos programas pequeños y sencillos, y para enfocarnos en la cuestión que se está explicando.

En programas reales se debe incluir la validación de datos. Nunca sabe lo que un usuario (o algún cracker que trate de entrar en forma ilícita a su sistema) hará. El término genérico para esto es *programación a la defensiva* debido a que se defienden la consistencia y el comportamiento apropiado del código y de los datos.

**ENTRADA**    **LISTADO 3.8** El mismo programa, pero esta vez con constantes de tipo entero

```

1: #include <iostream.h>
2: int main()
3: {
4:     const int Domingo = 0;
5:     const int Lunes = 1;
6:     const int Martes = 2;
7:     const int Miercoles = 3;
8:     const int Jueves = 4;
9:     const int Viernes = 5;
10:    const int Sabado = 6;
11:
12:    int opcion;
13:    cout << "Escriba un dia (0-6): ";
14:    cin >> opcion;
15:
16:    if (opcion == Domingo || opcion == Sabado)
17:        cout << "\n¡Ya se le agotaron los fines de semana!\n";

```

```
18: else
19:     cout << "\nEstá bien, incluiré un dia de descanso.\n";
20:
21: return 0;
22: }
```

**SALIDA** Escriba un dia (0-6): 6  
¡Ya se le agotaron los fines de semana!

**ANÁLISIS** La salida de este listado es idéntica a la que se muestra en el listado 3.7. Aquí, cada una de las constantes (Domingo, Lunes, etc.) se definió en forma explícita, y no existe el tipo enumerado DIAS. Las constantes enumeradas tienen la ventaja de documentarse por sí mismas (la intención del tipo enumerado DIAS se identifica de inmediato).

3

## Resumen

En la lección de hoy se habló sobre las variables y constantes numéricas y de tipo carácter utilizadas en C++ para guardar información durante la ejecución de su programa. Las variables numéricas pueden ser de tipo entero (`char`, `int`, `short int` y `long int`) o de punto flotante (`float` y `double`). Las variables numéricas también pueden ser con signo (`signed`) o sin signo (`unsigned`). Aunque todos los tipos pueden ser de varios tamaños en distintas computadoras, el tipo especifica un tamaño exacto en cualquier computadora.

Debe declarar una variable antes de poder utilizarla, y luego debe guardar el tipo de datos correcto en esa variable. Si coloca un número demasiado grande en una variable de tipo entero, ésta no podrá almacenarlo completamente y producirá un resultado incorrecto.

En esta lección también se habló sobre las constantes literales y simbólicas, así como las constantes enumeradas, y se mostraron dos maneras de declarar una constante simbólica: usando `#define` y usando la palabra reservada `const`.

## Preguntas y respuestas

- P** Si una variable de tipo `short int` se puede quedar sin espacio y sobregirarse, ¿por qué no utilizar siempre enteros largos?
- R** Tanto los enteros largos como los cortos se pueden quedar sin espacio y sobregirarse, pero un entero largo lo hará con un número mucho más grande. Por ejemplo, una variable de tipo `unsigned short int` se sobregirará después de 65,535, mientras que una de tipo `long int` lo hará hasta llegar a 4,294,967,295. Sin embargo, en la mayoría de los equipos, un entero largo ocupa hasta el doble de memoria cada vez que usted declara uno (4 bytes en comparación con 2 bytes para el entero corto), y un programa con 100 de esas variables consumirá 200 bytes adicionales de RAM. Francamente, esto no llega a ser un problema en la actualidad, pues la mayoría de

las computadoras personales vienen con muchos miles (si no millones) de bytes de memoria.

**P** ¿Qué pasa si asigno un número con punto decimal a una variable de tipo `int` en lugar de a una de tipo `float`? Considere la siguiente línea de código:

```
int unNumero = 5.4;
```

**R** Un buen compilador mostrará un mensaje de advertencia, pero la asignación es completamente válida. El número que usted asigne será truncado a un entero. Por lo tanto, si asigna 5.4 a una variable de tipo entero, esa variable tendrá el valor 5. Como puede ver, se perderá información, y si trata entonces de asignar el valor de esa variable de tipo `int` a una de tipo `float`, esta última tendrá sólo un 5.

**P** ¿Por qué no utilizar constantes literales? ¿Por qué tomarse la molestia de usar constantes simbólicas?

**R** Si utiliza el valor en muchos lugares en su programa, una constante simbólica permite que todos los valores cambien con sólo cambiar la definición de la constante. Las constantes simbólicas también se describen a sí mismas. Podría ser difícil entender por qué se multiplica un número por 360, pero sería más fácil entender lo que pasa si el número se multiplica por `gradosDeUnCírculo`.

**P** ¿Qué ocurre si asigno un número negativo a una variable sin signo? Considere la siguiente línea de código:

```
unsigned int unNumeroPositivo = -1;
```

**R** Un buen compilador mostrará una advertencia, pero la asignación es válida. El número negativo se valorará como patrón de bits y se asignará a la variable. El valor de esa variable se interpretará entonces como número sin signo. Por lo tanto, un -1, cuyo patrón de bits puede ser 11111111 11111111 en algunas representaciones binarias (0xFFFF en hexadecimal), será valorado como el valor sin signo 65,535. Si esta información es confusa para usted, vea el apéndice C.

**P** ¿Puedo trabajar con C++ sin entender patrones de bits, aritmética binaria y números hexadecimales?

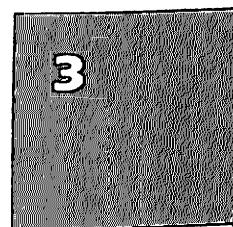
**R** Sí, pero no en forma tan efectiva como lo haría si comprendiera esos temas. C++ no hace un trabajo tan bueno como otros lenguajes en cuanto a “proTEGERLO” de lo que está haciendo en realidad su computadora. A decir verdad, esto es un beneficio ya que le proporciona un tremendo poder, algo que otros lenguajes no pueden hacer. Sin embargo, como con cualquier herramienta de poder, para obtener el máximo rendimiento de C++, debe comprender su funcionamiento. Los programadores que tratan de programar en C++ sin comprender los fundamentos del sistema binario, a menudo se confunden con sus resultados.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Cuál es la diferencia entre una variable de tipo entero y una de punto flotante?
2. ¿Cuáles son las diferencias entre un entero corto sin signo y un entero largo?
3. ¿Cuáles son las ventajas de usar una constante simbólica en lugar de una constante literal?
4. ¿Cuáles son las ventajas de usar la palabra reservada `const` en lugar de `#define`?
5. ¿Qué hace que el nombre de una variable sea bueno o malo?
6. Dado el siguiente enum, ¿cuál es el valor de AZUL?  
`enum COLOR { BLANCO, NEGRO = 100, ROJO, AZUL, VERDE = 300 };`
7. ¿Cuáles de los siguientes nombres de variables son buenos, cuáles son malos y cuáles no son válidos?
  - a. Edad
  - b. lex
  - c. R79J
  - d. IngresoTotal
  - e. \_\_Invalido



### Ejercicios

1. ¿Cuál sería el tipo de variable correcto para guardar la siguiente información?
  - a. Su edad.
  - b. El área de su patio.
  - c. El número de estrellas de la galaxia.
  - d. La cantidad promedio de lluvia para el mes de enero.
2. Cree nombres buenos de variables para la información de la pregunta 1.
3. Declare una constante para `pi` como 3.14159.
4. Declare una variable de tipo float e inicialíscela usando su constante `pi`.



# SEMANA 1

## DÍA 4

### Expresiones e instrucciones

Básicamente, un programa es un conjunto de comandos ejecutados en secuencia. El poder de un programa viene de su capacidad para ejecutar uno u otro conjunto de comandos, dependiendo de si una condición específica es verdadera o falsa. Hoy verá lo siguiente:

- Qué son las instrucciones
- Qué son los bloques
- Qué son las expresiones
- Cómo ramificar el código con base en ciertas condiciones
- Qué es la verdad, y cómo actuar con base en ella

#### Instrucciones

En C++, una *instrucción* controla la secuencia de la ejecución, evalúa una expresión, o no hace nada (la instrucción `null`). Todas las instrucciones de C++ terminan con punto y coma, incluso la instrucción `null`, la cual consta única-

mente del punto y coma. Una de las instrucciones más comunes es la siguiente instrucción de asignación:

```
x = a + b;
```

A diferencia del álgebra, esta instrucción no significa que  $x$  es igual a  $a + b$ . Esto se lee: "asignar el valor de la suma de  $a$  y  $b$  a  $x$ ", o "asignar a  $x$  el valor de  $a + b$ ". Aún cuando esta instrucción hace dos cosas, es una instrucción y por lo tanto solo tiene un punto y coma. El operador de asignación asigna lo que esté de su lado derecho a lo que esté de su lado izquierdo.

## Espacio en blanco

Por lo general, el espacio en blanco (que se crea por medio de tabuladores, espacios y caracteres de nueva línea) se ignora en las instrucciones. La instrucción de asignación descrita anteriormente se podría escribir de la siguiente manera:

```
x=a+b;
o
x           =a
+         b   ;
;
```

Aunque la última variación es perfectamente válida, también es perfectamente confusa. Puede utilizar el espacio en blanco para hacer que sus programas sean más legibles y fáciles de mantener, o lo puede usar para crear código horrendo e indescifrable. Aquí, como en todas las cosas, C++ proporciona el poder; usted pone la sensatez.

Los caracteres de espacio en blanco (espacios, tabuladores y caracteres de nueva línea) no se pueden ver. Si estos caracteres se imprimen, se verá sólo lo blanco del papel.

## Bloques de instrucciones e instrucciones compuestas

En cualquier parte donde coloque una instrucción sencilla, puede colocar una instrucción compuesta, conocida también como bloque. Un bloque empieza con una llave de apertura ({) y termina con una llave de cierre (}). Aunque todas las instrucciones del bloque deben terminar con punto y coma, el bloque en sí no termina con punto y coma, como se muestra en el siguiente ejemplo:

```
{
    temp = a;
    a = b;
    b = temp;
}
```

Este bloque de código actúa como una instrucción, e intercambia los valores de las variables  $a$  y  $b$ .

DEBE	No DEBE
DEBE utilizar una llave de cierre siempre que tenga una llave de apertura.	Debe utilizar una llave de cierre siempre que tenga una llave de apertura.
DEBE terminar sus instrucciones con punto y coma.	Debe terminar sus instrucciones con punto y coma.
DEBE utilizar el espacio en blanco con sencillez para que su código sea más claro.	Debe utilizar el espacio en blanco con sencillez para que su código sea más claro.

Las instrucciones compuestas son muy importantes, como verá al llegar a las secciones que hablan sobre las instrucciones `if/else`, `for, while` y `do/while`. Estas instrucciones sólo pueden tener una instrucción después de ellas (como parte de los resultados verdaderos o falsos de `if/else` o del cuerpo de los ciclos `for, while` y `do/while`). Para poder colocar más de una instrucción después de cualquiera de estas instrucciones, necesita utilizar un bloque (que cuenta como una sola instrucción) para que contenga tantas instrucciones como sea necesario. Aprenderá más sobre esto en la sección titulada “La instrucción `if`” de esta lección.

4

## Expresiones

En C++, cualquier cosa que tenga un valor es una *expresión*. Se dice que una expresión regresa un valor. Por lo tanto, la instrucción `3+2;` regresa el valor 5, por lo que se considera una expresión. Todas las expresiones son instrucciones.

Podría sorprenderse por la cantidad de piezas de código que califican como expresiones. He aquí algunos ejemplos:

```
3.2           // regresa el valor 3.2
PI            // constante de tipo float que regresa el valor 3.14
SegundosPorMinuto // constante de tipo int que regresa el valor 60
```

Asumiendo que `PI` es una constante igual a 3.14, y que `SegundosPorMinuto` es una constante igual a 60, las tres instrucciones son expresiones.

La expresión compleja

```
x = a + b;
```

no sólo suma  $a$  y  $b$  y asigna el resultado a  $x$ , sino que también regresa el valor de esa asignación (el valor de  $x$ ). Por lo tanto, este resultado también es una expresión. Debido a que es una expresión, puede estar en el lado derecho de un operador de asignación:

$y = x = a + b;$

Esta línea se evalúa en el siguiente orden:

Sumar  $a$  y  $b$ .

Asignar a  $x$  el resultado de la expresión  $a + b$ .

Asignar a  $y$  el resultado de la expresión de asignación  $x = a + b$ .

Si  $a$ ,  $b$ ,  $x$  y  $y$  son enteros, y si  $a$  tiene el valor 2 y  $b$  tiene el valor 5, entonces  $a$ ,  $x$  y  $a$  y se les asignará el valor 7.

El listado 4.1 muestra la forma en que C++ evalúa expresiones complejas.

#### ENTRADA LISTADO 4.1 Evaluación de expresiones complejas

```

1: #include <iostream.h>
2: int main()
3: {
4:     int a=0, b=0, x=0, y=35;
5:     cout << "a: " << a << " b: " << b;
6:     cout << " x: " << x << " y: " << y << endl;
7:     a = 9;
8:     b = 7;
9:     y = x = a+b;
10:    cout << "a: " << a << " b: " << b;
11:    cout << " x: " << x << " y: " << y << endl;
12:    return 0;
13: }
```

SALIDA  
a: 0 b: 0 x: 0 y: 35  
a: 9 b: 7 x: 16 y: 16

ANÁLISIS En la línea 4 se declaran e inicializan las cuatro variables. Sus valores se imprimen en las líneas 5 y 6. En la línea 7 se asigna el valor 9 a la variable  $a$ . En la línea 8 se asigna el valor 7 a  $b$ . En la línea 9 se suman los valores de  $a$  y  $b$  y el resultado se asigna a  $x$ . Esta expresión ( $x = a+b$ ) tiene un valor (la suma de  $a + b$ ), y ese valor se asigna a su vez a  $y$ .

## Operadores

Un *operador* es un símbolo que hace que el compilador realice una acción. Los operadores actúan sobre los operandos, y en C++ todos los operandos son expresiones. En C++ existen varias categorías de operadores. Dos de estas categorías son las siguientes:

- Operadores de asignación
- Operadores matemáticos

### Operador de asignación

El operador de asignación (=) hace que el operando de su lado izquierdo cambie su valor por el que se encuentra de su lado derecho. La siguiente expresión

```
x = a + b;
```

le asigna al operando x el valor del resultado de sumar a y b.

Un operando que puede estar de forma válida del lado izquierdo de un operador de asignación se conoce como *valor-i* (lvalue). Lo que puede estar del lado derecho se conoce como (sí, adivinó) *valor-d* (rvalue).

Las constantes son valores-d. No pueden ser valores-i. Por lo tanto, puede escribir

```
x = 35;           // correcto
```

pero no es válido que escriba

```
35 = x;           // error, no es un valor-i!
```

Es importante que recuerde esto, así es que vamos a repetirlo: Un valor-i es un operando que puede estar del lado izquierdo de una expresión. Un valor-d es un operando que puede estar del lado derecho de una expresión. Observe que todos los valores-i son valores-d, pero no todos los valores-d son valores-i. Un ejemplo de un valor-d que no es valor-i es una literal. Usted puede escribir `x = 5;`, pero no puede escribir `5 = x;` (`x` puede ser un valor-i o un valor-d, mientras que `5` sólo puede ser un valor-d).

4

### Operadores matemáticos

Los cinco principales operadores matemáticos son: suma (+), resta (-), multiplicación (\*), división (/) y residuo (%).

La suma y la resta funcionan como suma y resta normales, aunque la resta con enteros sin signo puede producir resultados inesperados si se utilizan números negativos. Ayer

vio algo muy similar a esto, al hablar sobre los desbordamientos de variables. El listado 4.2 muestra lo que ocurre al restar un número grande sin signo a un número pequeño sin signo.

### ENTRADA LISTADO 4.2 Una muestra de la resta y el desbordamiento de enteros

```

1: // Listado 4.2 - muestra la resta y
2: // el desbordamiento de enteros
3: #include <iostream.h>
4:
5: int main()
6: {
7:     unsigned int diferencia;
8:     unsigned int numeroGrande = 100;
9:     unsigned int numeroChico = 50;
10:    diferencia = numeroGrande - numeroChico;
11:    cout << "La diferencia es: " << diferencia;
12:    diferencia = numeroChico - numeroGrande;
13:    cout << "\nAhora la diferencia es: " << diferencia << endl;
14:    return 0;
15: }
```

### SALIDA

La diferencia es: 50  
Ahora la diferencia es: 4294967246

### ANÁLISIS

El operador de resta se invoca en la línea 10, y el resultado se imprime en la línea 11, como podría esperarse. En la línea 12 se llama otra vez al operador de resta, pero esta vez se resta un número grande sin signo a un número pequeño sin signo. El resultado sería negativo, pero como se evalúa (y se imprime) como número sin signo, el resultado es un desbordamiento, como se describió ayer. Este tema se describe con detalle en el apéndice A, “Precedencia de operadores”.

## División de enteros y el operador de módulo

La división de enteros es un poco distinta a la división común. De hecho, la división de enteros es *igual que* la división que usted aprendió en la primaria. Al dividir 21 entre 4 ( $21 / 4$ ) se realiza una división de enteros, y el resultado es 5, con un residuo de 1.

Para obtener el residuo, se hace la operación 21 módulo 4 ( $21 \% 4$ ) y el resultado es 1. El operador de módulo le indica el residuo producido por una división de enteros.

Encontrar el residuo puede ser muy útil. Por ejemplo, tal vez quiera imprimir una frase cada 10 acciones. Cualquier número cuyo residuo sea 0 al hacer la operación módulo 10, es un múltiplo exacto de 10. Por lo tanto,  $1 \% 10$  es 1,  $2 \% 10$  es 2, y así sucesivamente,

hasta  $10 \% 10$ , cuyo residuo es  $0$ .  $11 \% 10$  es igual a  $1$ , y este patrón continúa hasta el siguiente múltiplo de  $10$ , que es  $20$ . Verá esta técnica al hablar sobre los ciclos en el día 7, "Más flujo de programa".

### Preguntas frecuentes

**FAQ:** Al dividir  $5/3$ , obtengo  $1$ . ¿Qué estoy haciendo mal?

**Respuesta:** Si divide un entero entre otro, obtendrá un entero como resultado. Por lo tanto  $5/3$  será igual a  $1$ .

Para obtener un valor decimal debe utilizar valores y variables de tipo `float`.

$5.0 / 3.0$  le dará un resultado decimal:  $1.666667$ .

Si su método toma enteros como parámetros, necesita utilizar la especificación de tipo, en este caso de tipo `float`.

Al hacer una especificación de tipo con una variable, obliga a que cambie su tipo. En esencia, le está diciendo al compilador: "Sé lo que estoy haciendo". Y más vale que sea así, porque el compilador le dirá: "Está bien, jefe, es su responsabilidad".

En este caso específico, usted necesita decirle al compilador: "Sé que piensas que éste es un tipo `int`, pero sé lo que estoy haciendo: en realidad es un tipo `float`".

Existen dos maneras de realizar la especificación de tipo: puede utilizar la antigua especificación de tipo al estilo C, o puede utilizar el nuevo operador `static_cast` aprobado por ANSI. El listado 4.3 muestra la especificación de tipo para `float`.

4

### ENTRADA LISTADO 4.3 Especificación de tipo para float

```

1: #include <iostream.h>
2:
3: void intDiv(int x, int y)
4: {
5:     int z = x / y;
6:     cout << "z: " << z << endl;
7: }
8:
9: void floatDiv(int x, int y)
10:{ 
11:     float a = (float)x;           // estilo antiguo
12:     float b = static_cast<float>(y); // estilo preferido
13:     float c = a / b;
14:
15:     cout << "c: " << c << endl;
16:}
17:
18: int main()
19: {
```

continúa

**LISTADO 4.3** CONTINUACIÓN

```

20:     int x = 5, y = 3;
21:     intDiv(x,y);
22:     floatDiv(x,y);
23:     return 0;
24: }
```

**SALIDA**      z: 1  
c: 1.66667

**ANÁLISIS** En la línea 20 se declaran dos variables de tipo entero. En la línea 21 se pasan como parámetros a `intDiv`, y en la línea 22 se pasan como parámetros a `floatDiv`. Este segundo método empieza en la línea 9. En las líneas 11 y 12, los enteros se convierten a tipo `float` y se asignan a las variables de este tipo. En la línea 13, el resultado de la división se asigna a un tercer tipo `float`, y se imprime en la línea 15.

## Cómo combinar los operadores de asignación y matemáticos

Es muy común querer sumar un valor a una variable y luego asignar el resultado a esa misma variable. Si tiene una variable llamada `miEdad` y quiere sumar 2 a ese valor, puede escribir lo siguiente:

```

int miEdad = 5;
int temp;
temp = miEdad + 2; // sumar 5 + 2 y colocar el resultado en temp
miEdad = temp;      // colocar otra vez el resultado en miEdad
```

Sin embargo, este método es terriblemente complicado y desperdicia muchos pasos. En C++, usted puede colocar la misma variable en ambos lados del operador de asignación; por consecuencia, lo anterior se convierte en

```
miEdad = miEdad + 2;
```

que es mucho mejor. En álgebra, esta expresión no tendría sentido, pero en C++ se lee como: "sumar dos al valor de `miEdad` y asignar el resultado a `miEdad`".

Esto se puede simplificar más, pero tal vez sea un poco más difícil de entender:

```
miEdad += 2;
```

El operador aritmético de suma (`+=`) suma el valor que se encuentra a su derecha (valor-d) con el valor que se encuentra a su izquierda (valor-i) y luego vuelve a asignar el resultado al valor de la izquierda (valor-i). Este operador se pronuncia "más igual a". La instrucción se leería como "`miEdad` más igual a dos". Si `miEdad` tuviera un valor inicial de 4, tendría 6 después de esta instrucción.

También existen otros operadores aritméticos: el de resta (`-=`), de división (`/=`), de multiplicación (`*=`) y de módulo (`%=`).

## Incremento y decremento

El valor más común para sumar (o restar) y luego volver a asignar a una variable es 1. En C++, sumar 1 a un valor se conoce como *incrementar*, y restar 1 se conoce como *decrementar*. Debido a que esta operación es muy común, la mayoría de las computadoras tiene operadores integrados en el hardware para realizar estas acciones. C++ tiene operadores especiales para realizar estas acciones.

El operador de incremento (`++`) incrementa en 1 el valor de una variable, y el operador de decremento (`--`) lo decremente en 1. Por lo tanto, si tiene una variable llamada `C` y quiere incrementarla, puede utilizar la siguiente instrucción:

```
C++; // Empezar con C e incrementarla.
```

Esta instrucción equivale a la siguiente, que es un poco más elaborada:

```
C = C + 1;
```

la cual, a su vez, equivale a la instrucción

```
C += 1;
```

4

### Nota

Tal vez haya observado que el nombre de este lenguaje ("C++") es similar a un operador de incremento aplicado a la variable `C`. Esto no es casualidad. El lenguaje C++ fue desarrollado como una mejora del lenguaje C existente, y no como un lenguaje completamente nuevo. Es por esto que el nombre imita esa filosofía.

## Prefijo y posfijo

Tanto el operador de incremento (`++`) como el operador de decremento (`--`) vienen en dos variedades: prefijo y posfijo. La variedad de prefijo se escribe antes del nombre de la variable (`++miEdad`); la variedad de posfijo se escribe después (`miEdad++`).

En una instrucción sencilla no importa mucho cuál variedad se utilice, pero en una instrucción compleja, en la que se incremente (o decremente) el valor de una variable y luego se asigne el resultado a otra variable, sí es muy importante. El operador de prefijo se evalúa antes de la asignación; el de posfijo se evalúa después.

La semántica del prefijo es la siguiente: Incrementar el valor y luego usarlo. La semántica del posfijo es distinta: Usar el valor y luego incrementar el original.

Esto puede ser confuso al principio, pero si *x* es una variable de tipo entero cuyo valor es 5, y usted escribe

```
int a = ++x;
```

esto le indica al compilador que incremente *x* (cambiando su valor a 6) y luego tome ese valor y lo asigne a la variable *a*. Por lo tanto, ahora *a* vale 6 y *x* tambien vale 6.

Si después de hacer esto, escribe

```
int b = x++;
```

ahora el compilador toma el valor contenido en *x* (6) y lo asigna a *b*, luego incrementa el valor de *x*. Por lo tanto, ahora *b* vale 6, pero *x* vale 7. El listado 4.4, muestra el uso y las implicaciones de ambos tipos.

---

**ENTRADA LISTADO 4.4 Una muestra de los operadores de prefijo y posfijo**

---

```
1: // Listado 4.4 - muestra el uso de
2: // los operadores de incremento de
3: // prefijo y posfijo
4: #include <iostream.h>
5: int main()
6: {
7:     int miEdad = 39;          // inicializar dos variables de tipo entero
8:     int suEdad = 39;
9:     cout << "Yo tengo: " << miEdad << " años.\n";
10:    cout << "Usted tiene: " << suEdad << " años \n";
11:    miEdad++;                // incremento mediante posfijo
12:    ++suEdad;                // incremento mediante prefijo
13:    cout << "Pasa un año...\n";
14:    cout << "Ahora tengo: " << miEdad << " años.\n";
15:    cout << "Usted tiene: " << suEdad << " años\n";
16:    cout << "Pasa otro año\n";
17:    cout << "Ahora tengo: " << miEdad++ << " años.\n";
18:    cout << "Usted tiene: " << ++suEdad << " años\n";
19:    cout << "Imprimamos eso de nuevo.\n";
20:    cout << "Yo tengo: " << miEdad << " años.\n";
21:    cout << "Usted tiene: " << suEdad << " años\n";
22:    return 0;
23: }
```

---

**SALIDA**

```

Yo tengo: 39 años.
Usted tiene: 39 años.
Pasa un año...
Ahora tengo: 40 años.
Usted tiene: 40 años.
Pasa otro año
Ahora tengo: 40 años.
Usted tiene: 41 años.
Imprimamos eso de nuevo.
Yo tengo: 41 años.
Usted tiene: 41 años.

```

**ANÁLISIS**

En las líneas 7 y 8 se declaran dos variables de tipo entero, y cada una se inicializa con el valor 39. Sus valores se imprimen en las líneas 9 y 10.

En la línea 11 `miEdad` se incrementa usando el operador de incremento de posfijo, y en la línea 12 `suEdad` se incrementa utilizando el operador de prefijo. Los resultados se imprimen en las líneas 14 y 15, y ambos son idénticos (40).

En la línea 17 `miEdad` se incrementa como parte de la instrucción de impresión, usando el operador de incremento de posfijo. debido a que es posfijo, el incremento ocurre después de la impresión, y esto ocasiona que se vuelva a imprimir el valor de 40. En contraste, en la línea 18 `suEdad` se incrementa utilizando el operador de incremento de prefijo. Por consecuencia, se incrementa antes de que se imprima, y se despliega el valor 41.

Finalmente, en las líneas 20 y 21 se imprimen otra vez los valores. Como ya se ha completado la instrucción de incremento, el valor de `miEdad` ahora es 41, así como el de `suEdad`.

4

## Precedencia de operadores

¿Qué se realiza primero en la siguiente instrucción compleja, la suma o la multiplicación?:

`x = 5 + 3 * 8;`

Si se realiza primero la suma, el resultado es  $8 * 8$ , o 64. Si se realiza primero la multiplicación, la respuesta es  $5 + 24$ , o 29.

Todos los operadores tienen un valor de precedencia, y la lista completa se muestra en el apéndice A. La multiplicación tiene una precedencia mayor que la suma; por lo tanto, el valor de la expresión es 29.

Cuando dos operadores matemáticos tienen la misma precedencia, se procesan en orden de izquierda a derecha. Por ejemplo, en la expresión

`x = 5 + 3 + 8 * 9 + 6 * 4;`

primero se evalúa la multiplicación, de izquierda a derecha. Por lo tanto,  $8 * 9 = 72$ , y  $6 * 4 = 24$ . Ahora la expresión quedaría de la siguiente manera:

$$x = 5 + 3 + 72 + 24;$$

Y la suma, que se evalúa de izquierda a derecha, sería  $5 + 3 = 8$ ;  $8 + 72 = 80$ ;  $80 + 24 = 104$ .

Tenga cuidado con esto. Algunos operadores, como los de asignación, ¡se evalúan de derecha a izquierda! De cualquier manera, ¿qué pasa si la precedencia no satisface sus necesidades? Considere la siguiente expresión:

```
TotalSegundos = NumMinutosParaPensar + NumMinutosParaEscribir * 60
```

En esta expresión no se quiere multiplicar la variable `NumMinutosParaEscribir` por `60` y luego sumarla a `NumMinutosParaPensar`. Se quiere sumar las dos variables para obtener el número total de minutos, y luego multiplicar ese número por `60` para obtener el total de segundos.

En este caso se utilizan paréntesis para cambiar el orden de precedencia. Los elementos entre paréntesis se evalúan con una mayor precedencia que cualquiera de los operadores matemáticos. Por lo tanto,

```
TotalSegundos = (NumMinutosParaPensar + NumMinutosParaEscribir) * 60
```

lograría lo que se necesita.

## Paréntesis anidados

Para expresiones complejas, tal vez sea necesario anidar paréntesis, es decir, colocar unos dentro de otros. Por ejemplo, podría necesitar calcular el total de segundos y luego calcular el número total de personas involucradas antes de multiplicar los segundos por las personas:

```
TotalSegundosPersona = ( (NumMinutosParaPensar + NumMinutosParaEscribir) * 60 + NumSegundosParaEscuchar) * (PersonasEnLaOficina + PersonasDeVacaciones)
```

Esta expresión compleja se lee de adentro hacia fuera. Primero, `NumMinutosParaPensar` se suma a `NumMinutosParaEscribir`, ya que estas dos variables se encuentran en los paréntesis de más adentro. Luego esta suma se multiplica por `60` y el producto se suma a `NumSegundosParaEscuchar`. A continuación, `PersonasEnLaOficina` se suma a `PersonasDeVacaciones`. Finalmente, el número total de personas encontradas se multiplica por el número total de segundos.

Este ejemplo trae a la mente una cuestión relacionada importante. Para una computadora es fácil comprender esta expresión, pero para un humano es muy difícil leerla, comprenderla o modificarla. He aquí la misma expresión escrita de otra manera, usando algunas variables temporales de tipo entero:

```
TotalMinutos = NumMinutosParaPensar + NumMinutosParaEscribir;
TotalSegundos = TotalMinutos * 60;
TotalPersonas = PersonasEnLaOficina + PersonasDeVacaciones;
TotalSegundosPersona = TotalPersonas * TotalSegundos;
```

Este ejemplo es más largo y utiliza más variables temporales que el ejemplo anterior, pero es mucho más sencillo comprenderlo. Agregue un comentario al principio para explicar lo que hace este código y cambie el 60 por una constante simbólica. Entonces tendrá código fácil de entender y de mantener.

DEBE	NO DEBE
<p><b>DEBE</b> recordar que las expresiones tienen un valor.</p> <p><b>DEBE</b> utilizar el operador de prefijo (<code>++variable</code>) para incrementar o decrementar la variable antes de usarla en la expresión.</p> <p><b>DEBE</b> utilizar el operador de posfijo (<code>variable++</code>) para incrementar o decrementar la variable después de utilizarla.</p> <p><b>DEBE</b> utilizar paréntesis para cambiar el orden de precedencia.</p>	<p><b>NO DEBE</b> anidar muchos paréntesis ya que la expresión se vuelve difícil de comprender y de mantener.</p>

4

## La naturaleza de la verdad

En versiones anteriores de C++, la verdad y la falsedad se representaban con enteros, pero el nuevo estándar ANSI ha introducido un tipo nuevo: `bool`. Este tipo nuevo tiene dos valores posibles: falso (`false`) o verdadero (`true`).

Cada expresión puede ser evaluada para ver si es verdadera o falsa. Las expresiones que se evalúan matemáticamente en cero regresarán el valor `false`; todas las demás regresarán `true`.

### Nota

Anteriormente, muchos compiladores ofrecían un tipo `bool`, el cual se representaba internamente como `int` y por lo general tenía un tamaño de 4 bytes. Ahora los compiladores que se apegan al estándar ANSI por lo general proporcionan un tipo `bool` de 1 byte.

Los compiladores GNU se apegan al estándar ANSI; su tipo `bool` ocupa 1 byte. Me aseguré de ello. Tal vez usted se pregunte cómo. Lo hice escribiendo un programa rápido de muestra que utiliza la función `sizeof()` como en los ejemplos del día 3, "Variables y constantes".

## Operadores relacionales

Este tipo de operadores se utiliza para determinar si dos números son iguales o uno es mayor o menor que el otro. Cada expresión relacional se evalúa como verdadera (`true`) o falsa (`false`). Los operadores relacionales se presentan en la tabla 4.1.

### Nota

El nuevo estándar ANSI ha introducido el nuevo tipo `bool`, y ahora todos los operadores relacionales regresan un valor de tipo `bool`: `true` o `false`. En versiones anteriores de C++, estos operadores regresaban 0 para `false` o un valor distinto de cero (por lo general 1) para `true`.

Cuando se trabaja con los intérpretes de comandos de Linux (`csh`, `tcsh`, `zsh`, `bash`, entre otros) los valores de `true` y `false` se invierten. Si realiza programación "shell", el valor 0 corresponderá a `true` y cualquier otro valor corresponderá a `false`. Revise la documentación de su sistema para profundizar en estos detalles.

Si la variable de tipo entero `miEdad` tiene el valor 39, y la variable de tipo entero `suEdad` tiene el valor 40, usted puede determinar si son iguales mediante el uso del operador relacional "igual a" (`==`):

```
miEdad == suEdad; // ¿es igual el valor de miEdad al de suEdad?
```

Esta expresión tendría el valor 0, o `false`, debido a que las variables no son iguales. La expresión

```
miEdad < suEdad; // ¿es miEdad menor que suEdad?
```

tendría un valor diferente de 0, o `true`.

### Precaución

Muchos programadores de C++ novatos confunden el operador de asignación (`=`) con el operador relacional igual a (igualdad) (`==`). Esto puede crear un terrible error en su programa.

Los seis operadores relacionales son: igual a (`==`), menor que (`<`), mayor que (`>`), menor o igual a (`<=`), mayor o igual a (`>=`) y diferente de (`!=`). La tabla 4.1 muestra cada operador relacional, su uso y un pequeño ejemplo.

**TABLA 4.1** Los operadores relacionales

Nombre	Operador	Ejemplo	Valor
Igual a	<code>==</code>	<code>100 == 50;</code> <code>50 == 50;</code>	<code>false</code> <code>true</code>

Nombre	Operador	Ejemplo	Valor
No igual o diferente de	<code>!=</code>	<code>100 != 50;</code> <code>50 != 50;</code>	<code>true</code> <code>false</code>
Mayor que	<code>&gt;</code>	<code>100 &gt; 50;</code> <code>50 &gt; 50;</code>	<code>true</code> <code>false</code>
Mayor o igual a	<code>&gt;=</code>	<code>100 &gt;= 50;</code> <code>50 &gt;= 50;</code>	<code>true</code> <code>true</code>
Menor que	<code>&lt;</code>	<code>100 &lt; 50;</code> <code>50 &lt; 50;</code>	<code>false</code> <code>false</code>
Menor o igual a	<code>&lt;=</code>	<code>100 &lt;= 50;</code> <code>50 &lt;= 50;</code>	<code>false</code> <code>true</code>

**DEBE**

**DEBE** recordar que los operadores relacionales regresan los valores `true` (verdadero) o `false` (falso).

**No DEBE**

**NO DEBE** confundir el operador de asignación (`=`) con el operador relacional igual a (`==`). Éste es uno de los errores más comunes en la programación con C++; tenga cuidado con esto.

4

## La instrucción if

Por lo general, su programa fluye línea por línea en el orden en el que aparece en su código fuente. La instrucción `if` le permite probar una condición (por ejemplo, si dos variables son iguales) y saltar hacia distintas partes del código, dependiendo del resultado.

La forma más simple de una instrucción `if` es la siguiente:

```
if (expresión)
    instrucción1;
```

La expresión que está entre paréntesis puede ser cualquier expresión, pero por lo general contiene una de las expresiones relacionales. Si la expresión tiene el valor `false`, la instrucción1 no se ejecutará. Si tiene el valor `true`, la instrucción1 se ejecutará. Considere el siguiente ejemplo:

```
if (numeroGrande > numeroChico)
    numeroGrande = numeroChico;
```

Este código compara `numeroGrande` y `numeroChico`. Si `numeroGrande` es mayor, la segunda línea le asigna el valor de `numeroChico`.

Dado que un bloque de instrucciones encerradas entre llaves es equivalente a una sola instrucción, el siguiente tipo de ramificación puede ser bastante grande y poderoso:

```
if (expresión)
{
    instrucción1;
    instrucción2;
    instrucción3;
}
```

Un ejemplo simple de este uso se vería así:

```
if (numeroGrande > numeroChico)
{
    numeroGrande = numeroChico;
    cout << "numeroGrande: " << numeroGrande << "\n";
    cout << "numeroChico: " << numeroChico << "\n";
}
```

Esta vez, si `numeroGrande` es mayor que `numeroChico`, no sólo se le asigna el valor de `numeroChico`, sino que también se imprime un mensaje de información. El listado 4.5 muestra un ejemplo más detallado de la ramificación basada en los operadores relacionales.

#### **ENTRADA** **LISTADO 4.5** Una muestra de la ramificación basada en los operadores relacionales

```
1: // Listado 4.5 - muestra el uso de la instrucción if
2: // con los operadores relacionales
3: #include <iostream.h>
4: int main()
5: {
6:     int CarrerasMediasRojas, CarrerasYanquis;
7:     cout << "Escriba las carreras anotadas por los Medias rojas: ";
8:     cin >> CarrerasMediasRojas;
9:
10:    cout << "\nEscriba las carreras anotadas por los Yanquis: ";
11:    cin >> CarrerasYanquis;
12:
13:    cout << "\n";
14:
15:    if (CarrerasMediasRojas > CarrerasYanquis)
16:        cout << "¡Vamos, Medias rojas!\n";
17:
18:    if (CarrerasMediasRojas < CarrerasYanquis)
19:    {
20:        cout << "¡Vamos, Yanquis!\n";
21:        cout << "¡Son días felices en Nueva York!\n";
22:    }
23:
24:    if (CarrerasMediasRojas == CarrerasYanquis)
25:    {
```

```
26:         cout << "¿Un empate? Nooo, no puede ser.\n";
27:         cout << "Escriba las carreras que anotaron realmente los Yanquis: ";
28:         cin >> CarrerasYanquis;
29:
30:         if (CarrerasMediasRojas > CarrerasYanquis)
31:             cout << "¡Lo sabia! ¡Vamos, Medias rojas!";
32:
33:         if (CarrerasYanquis > CarrerasMediasRojas)
34:             cout << "¡Lo sabia! ¡Vamos, Yanquis!";
35:
36:         if (CarrerasYanquis == CarrerasMediasRojas)
37:             cout << "¡Vaya! ¡Realmente fue un empate!";
38:     }
39:
40:     cout << "\nGracias por decírmelo.\n";
41:     return 0;
42: }
```

**SALIDA**

Escriba las carreras anotadas por los Medias rojas: 10

Escriba las carreras anotadas por los Yanquis: 10

¿Un empate? Nooo, no puede ser.

Escriba las carreras que anotaron realmente los Yanquis: 8

¡Lo sabia! ¡Vamos, Medias rojas!

Gracias por decírmelo.

4

**ANÁLISIS**

Este programa pide al usuario la puntuación (número de carreras anotadas) de los dos equipos de béisbol; las puntuaciones se guardan en variables de tipo entero.

Las variables se comparan en la instrucción `if` de las líneas 15, 18 y 24.

Si una puntuación es mayor que la otra, se imprime un mensaje de información. Si las puntuaciones son iguales, el programa entra al bloque de código que empieza en la línea 25 y termina en la línea 38. Se pide otra vez la segunda puntuación, y luego se comparan nuevamente las puntuaciones.

Observe que si la puntuación inicial de los Yanquis fuera más alta que la de los Medias Rojas, la instrucción `if` de la línea 15 tendría el valor `false` (falso), y no se invocaría la línea 16. La prueba de la línea 18 tendría el valor `true` (verdadero), y se invocarían las instrucciones de las líneas 20 y 21. Luego se probaría la instrucción `if` de la línea 24 y tendría el valor `false` (si la línea 18 fuera `true`). Por lo tanto, el programa saltaría el bloque completo, hasta llegar a la línea 39.

En este ejemplo, si se obtiene un resultado `true` en una instrucción `if`, esto no evita que se prueben las otras instrucciones `if`.



### Precaución

Muchos programadores de C++ novatos colocan sin querer un punto y coma después de la instrucción `if`:

```
if (AlgunValor < 10);
    AlgunValor = 10;
```

Lo que se trata de hacer aquí es probar si `AlgunValor` es menor que 10, y de ser así, establecer su valor en 10, haciendo que 10 sea el valor mínimo para `AlgunValor`. Si ejecuta esta pieza de código, descubrirá que `AlgunValor` siempre se establece en 10. ¿Por qué? La instrucción `if` termina con el punto y coma (el operador que no hace nada).

Recuerde que la sangría no tiene ningún significado para el compilador. Esta pieza de código se habría podido escribir más apropiadamente como

```
if (AlgunValor < 10) // probar
; // no hacer nada
AlgunValor = 10; // asignar
```

Quitar el punto y coma hará que la línea final forme parte de la instrucción `if`, y así el código funcionará como es debido.

## Estilos de sangría

El listado 4.5 muestra un estilo de sangría para las instrucciones `if`. Sin embargo, no hay nada más propenso a crear una guerra religiosa que preguntar a un grupo de programadores cuál es el mejor estilo para alinear las llaves. Aunque puede haber docenas de variaciones, las tres siguientes parecen ser las favoritas:

- Colocar la llave inicial después de la condición y alinear la llave final debajo de `if` para cerrar el bloque de instrucciones:

```
if (expresión) {
    instrucciones
}
```

- Alinear las llaves debajo de `if` y utilizar sangrías en las instrucciones:

```
if (expresión)
{
    instrucciones
}
```

- Utilizar sangrías en las llaves e instrucciones:

```
if (expresión)
{
    instrucciones
}
```

En este libro se utiliza la segunda alternativa, ya que ambos autores encuentran que es más fácil entender dónde empiezan y terminan los bloques de instrucciones si se alinean las llaves una con otra y con la condición que se está probando. De nuevo, no importa mucho cuál estilo elija, siempre y cuando sea consistente. Para el compilador, esto tampoco es importante.

### else

A menudo su programa necesitará ejecutar ciertas instrucciones si la condición es `true` (verdadera), y otras si la condición es `false` (falsa). En el listado 4.5 se imprimía un mensaje (`¡Vamos, Medias rojas!`) si la primera prueba (`CarrerasMediasRojas > CarrerasYanquis`) se evaluaba como verdadera, y otro mensaje (`¡Vamos, Yanquis!`) si se evaluaba como falsa.

El método mostrado hasta ahora (probar primero una condición y luego la otra) funciona bien, pero es un poco incómodo. La cláusula `else` puede ayudar a tener un código mucho más legible:

```
if (expresión)
    instrucción;
else
    instrucción;
```

El listado 4.6 muestra el uso de la cláusula `else`.

4

#### ENTRADA LISTADO 4.6 Muestra de la cláusula else

```
1: // Listado 4.6 - muestra el uso de la instrucción if
2: // con la cláusula else
3: #include <iostream.h>
4: int main()
5: {
6:     int primerNumero, segundoNumero;
7:     cout << "Escriba un número grande: ";
8:     cin >> primerNumero;
9:     cout << "\nEscriba un número más pequeño: ";
10:    cin >> segundoNumero;
11:    if (primerNumero >= segundoNumero)
12:        cout << "\n¡Gracias!\n";
13:    else
14:        cout << "\nOh. ¡El segundo es más grande!\n";
15:
16:    return 0;
17: }
```

#### SALIDA

Escriba un número grande: 10

Escriba un número más pequeño: 12  
Oh. ¡El segundo es más grande!

**ANÁLISIS**

La instrucción `if` se evalúa en la línea 11. Si la condición es `true` (verdadera), se ejecuta la instrucción de la línea 12; si la condición es `false` (falsa), se ejecuta la instrucción de la línea 14. Si se quitara la cláusula `else` de la linea 13, la instrucción de la línea 14 se ejecutaría sin importar si la instrucción `if` fuera o no `true`. Recuerde, la instrucción `if` termina después de la línea 12. Si `else` no estuviera ahí, la linea 14 sería solamente la siguiente línea del programa.

Recuerde que cada una o ambas instrucciones se pueden reemplazar con un bloque de código entre llaves.

**La instrucción if**

La sintaxis para la instrucción `if` es la siguiente:

**Forma 1**

```
if (expresión)
    instrucción;
    siguiente instrucción;
```

Si la expresión se evalúa como verdadera, se ejecuta la instrucción y el programa continúa con la siguiente instrucción. Si la expresión es falsa, se ignora la instrucción y el programa salta hasta la siguiente instrucción.

Recuerde que la instrucción puede ser sencilla con un punto y coma al final, o un bloque de instrucciones encerradas entre llaves (aunque también puede ser vacía, o `null`).

**Forma 2**

```
if (expresión)
    instrucción1;
else
    instrucción2;
    siguiente instrucción;
```

Si la expresión se evalúa como verdadera, se ejecuta `instrucción1`; de no ser así, se ejecuta `instrucción2`. Después de eso, el programa continúa con la siguiente instrucción.

**Ejemplo 1**

```
if (AlgunValor < 10)
    cout << "AlgunValor es menor que 10";
else
    cout << "AlgunValor no es menor que 10!";
cout << "Listo." << endl;
```

**Instrucciones if avanzadas**

Vale la pena mencionar que se puede utilizar cualquier instrucción en las cláusulas `if` o `else`, incluso otra instrucción `if` o `else`. Por lo tanto, usted podría ver instrucciones `if` complejas de la siguiente forma:

```
if (expresión1)
{
```

```

if (expresión2)
    instrucción 1;
else
{
    if (expresión3)
        instrucción 2;
    else
        instrucción 3;
}
else
    instrucción 4;

```

Esta voluminosa instrucción **if** dice: "Si tanto **expresión1** como **expresión2** son verdaderas (**true**), ejecutar **instrucción1**. Si **expresión1** es verdadera, pero **expresión2** no lo es, entonces si **expresión3** es verdadera, ejecutar **instrucción2**. Si **expresión1** es verdadera pero **expresión2** y **expresión3** no lo son, ejecutar **instrucción3**. Finalmente, si **expresión1** no es verdadera, ejecutar **instrucción4**". Como puede ver, las instrucciones **if** complejas pueden ser confusas!

El listado 4.7 proporciona un ejemplo de una instrucción **if** compleja.

4

**ENTRADA LISTADO 4.7 Una instrucción if compleja**

```

1: // Listado 4.7 - una instrucción if
2: // compleja
3: #include <iostream.h>
4: int main()
5: {
6:     // Pedir dos números
7:     // Asignar los números a primerNumero y segundoNumero
8:     // Si primerNumero es mayor que segundoNumero,
9:     // ver si primerNumero es un múltiplo de segundoNumero
10:    // Si esto sucede, ver si son el mismo número
11:
12:    int primerNumero, segundoNumero;
13:    cout << "Escriba dos números.\nPrimer: ";
14:    cin >> primerNumero;
15:    cout << "\nSegundo: ";
16:    cin >> segundoNumero;
17:    cout << "\n\n";
18:
19:    if (primerNumero >= segundoNumero)
20:    {
21:        if ((primerNumero % segundoNumero) == 0)
22:        // ¿es primerNumero múltiplo de segundoNumero?
23:        {
24:            if (primerNumero == segundoNumero)
                cout << "Son iguales!\n";

```

continúa

**LISTADO 4.7** CONTINUACIÓN

```

25:         else
26:             cout << "¡El primer número es múltiplo del segundo!\n";
27:         }
28:     else
29:         cout << "¡El primer número no es múltiplo del segundo!\n";
30:     }
31: else
32:     cout << "¡Hey! ¡El segundo es más grande!\n";
33: return 0;
34: }
```

**SALIDA**

Escriba dos números.

Primero: 10

Segundo: 2

¡El primer número es múltiplo del segundo!

**ANÁLISIS**

Se piden dos números, uno a la vez, y luego se comparan. La primera instrucción `if`, en la línea 19, comprueba que el primer número sea mayor o igual que el segundo. Si no es así, se ejecuta la cláusula `else` de la línea 31.

Si la primera instrucción `if` es verdadera, se ejecuta el bloque de código que empieza en la línea 20, y se prueba la segunda instrucción `if`, que está en la línea 21. Esto comprueba si la operación módulo del primer número con el segundo no produce residuo. De ser así, el primer número es múltiplo del segundo o son iguales. La instrucción `if` de la línea 23 comprueba la igualdad y despliega el mensaje apropiado según sea el caso.

Si la instrucción `if` de la línea 21 falla, se ejecuta la cláusula `else` de la línea 28.

## Llaves en instrucciones `if` complejas

Aunque es válido no escribir las llaves en instrucciones `if` que tengan sólo una instrucción, y aunque sea válido anidar instrucciones `if` como la siguiente:

```

if (x > y)          // si x es mayor que y
    if (x < z)      // y si x es menor que z
        x = y;       // entonces asignar el valor de y a x
```

cuando escriba instrucciones anidadas largas, puede haber mucha confusión.

Recuerde, los espacios en blanco y las sangrías son una conveniencia para el programador; no hacen ninguna diferencia para el compilador. Es fácil confundir la lógica y asignar sin querer una cláusula `else` a la instrucción `if` equivocada. El listado 4.8 ilustra este problema.

**ENTRADA**

**LISTADO 4.8** Una muestra de por qué las llaves ayudan a aclarar cuál instrucción `else` va con cuál instrucción `if`

```

1: // Listado 4.8 - muestra por qué las llaves
2: // son importantes en instrucciones if anidadas
3: #include <iostream.h>
4: int main()
5: {
6:     int x;
7:     cout << "Escriba un número menor que 10 o mayor que 100: ";
8:     cin >> x;
9:     cout << "\n";
10:
11:    if (x >= 10)
12:        if (x > 100)
13:            cout << "Mayor que 100, ¡gracias!\n";
14:    else // ino es la instrucción else que se quiere!
15:        cout << "Menor que 10, ¡gracias!\n";
16:
17:    return 0;
18: }
```

**SALIDA**

Escriba un número menor que 10 o mayor que 100: 20

Menor que 10, ¡gracias!

4

**ANÁLISIS**

El programador quería pedir un número menor que 10 o mayor que 100, comprobar el valor correcto y luego imprimir un mensaje de agradecimiento.

Si la instrucción `if` de la línea 11 es verdadera, se ejecuta la siguiente instrucción (línea 12). En este caso, la línea 12 se ejecuta cuando el número escrito es mayor que 10. La línea 12 contiene también una instrucción `if`. Esta instrucción `if` es verdadera si el número escrito es mayor que 100. Si el número es mayor que 100, se ejecuta la instrucción de la línea 13.

Si el número escrito es menor que 10, la instrucción `if` de la línea 11 es falsa. El control del programa se va hasta la siguiente línea después de la instrucción `if`, en este caso la línea 16. Si escribe un número menor que 10, la salida es la siguiente:

Escriba un número menor que 10 o mayor que 100: 9

El propósito de la cláusula `else` de la línea 14 era incluirla en la instrucción `if` de la línea 11, para lo cual tiene la sangría adecuada. Desafortunadamente, la instrucción `else` se incluye, en realidad, en la instrucción `if` de la línea 12, y por consecuencia el programa tiene un ligero error.

El error es ligero porque el compilador no se quejará. Éste es un programa de C++ válido, pero no proporciona el resultado deseado. Más aún, la mayoría de las veces que el programador pruebe este programa, parecerá funcionar. Mientras se escriba un número mayor que 100, el programa parecerá funcionar bien. ¡Éste es un buen ejemplo de un error de lógica!

**Precaución**

Todas las cláusulas `else` buscarán la instrucción `:f` inmediata anterior para incluirse en ella.

El listado 4.9 arregla el problema poniendo las llaves necesarias.

**ENTRADA LISTADO 4.9 Una muestra del uso adecuado de las llaves en una instrucción if**

```

1: // Listado 4.9 - muestra el uso apropiado de las llaves
2: // en instrucciones if anidadas
3: #include <iostream.h>
4: int main()
5: {
6:     int x;
7:     cout << "Escriba un número menor que 10 o mayor que 100: ";
8:     cin >> x;
9:     cout << "\n";
10:
11:    if (x >= 10)
12:    {
13:        if (x > 100)
14:            cout << "Mayor que 100, ¡gracias!\n";
15:    }
16:    else // arreglado!
17:        cout << "Menor que 10, ¡gracias!\n";
18:    return 0;
19: }
```

**SALIDA** Escriba un número menor que 10 o mayor que 100: 20**ANÁLISIS** Las llaves de las líneas 12 y 15 hacen que todo lo que esté dentro de ellas sea una sola instrucción, y ahora la cláusula `else` de la línea 16 se incluye en la instrucción `if` de la línea 11, como se quería.

El usuario escribió 20, por lo que la instrucción `if` de la línea 11 es verdadera (`true`); sin embargo, la instrucción `if` de la línea 13 es falsa (`false`), por lo tanto no se imprime nada. Sería mejor si el programador colocara otra cláusula `else` después de la línea 14 para que se detectaran los errores y se imprimiera un mensaje.

**Nota**

Los programas mostrados en este libro están escritos para demostrar las cuestiones específicas que se están describiendo. Se mantienen simples de manera intencional; no se hace ningún intento por hacer el código “a prueba de balas” para protegerlo contra los errores del usuario. Con un código de calidad profesional, cada posible error del usuario se anticipa y se maneja de manera efectiva.

¡Recuerde programar a la defensiva!

## Operadores lógicos

A menudo querrá hacer más de una pregunta relacional a la vez. “¿Es verdad que *x* es mayor que *y*, y es también verdad que *y* es mayor que *z*?”. Un programa podría necesitar determinar que ambas condiciones sean verdaderas (o que alguna otra condición sea verdadera) para poder realizar una acción.

Imagine un sofisticado sistema de alarma que tenga esta lógica: “Si suena la alarma de la puerta después de las 6 p.m., Y (AND) NO (NOT) es un día festivo, O (OR) si es un fin de semana, entonces llamar a la policía”. Los tres operadores lógicos de C++ se utilizan para hacer este tipo de evaluación. Estos operadores se muestran en la tabla 4.2.

**TABLA 4.2** Los operadores lógicos

Operador	Símbolo	Ejemplo
AND	&&	expresión1 && expresión2
OR		expresión1    expresión2
NOT	!	!expresión

4

### Operador lógico AND

Una expresión lógica AND evalúa dos expresiones, y si ambas expresiones son verdaderas (true), la expresión lógica AND también es verdadera. Si es verdad que usted está hambriento, y es verdad que tiene dinero, entonces es verdad que puede comprar el almuerzo. Por lo tanto,

```
if ( (x == 5) && (y == 5) )
```

sería verdadera si tanto *x* como *y* son iguales a 5, y sería falsa (false) si cualquiera de las dos o las dos son diferentes de 5. Observe que ambos lados deben ser verdaderos para que toda la expresión sea verdadera.

Observe que el operador lógico AND está compuesto por dos símbolos &. Un solo símbolo & es un operador diferente, el cual se describe en el día 21, “Qué sigue”.

### Operador lógico OR

Una expresión lógica OR evalúa dos expresiones. Si cualquiera de ellas es verdadera, entonces la expresión OR es verdadera. Si usted tiene dinero o una tarjeta de crédito, puede pagar la cuenta. No necesita tener dinero y tarjeta de crédito a la vez; sólo necesita una de estas cosas, aunque tener las dos también sería perfecto. Por lo tanto,

```
if ( (x == 5) || (y == 5) )
```

sería verdadera si *x* o *y* son iguales a 5, o si ambas son iguales a 5.

Observe que el operador lógico OR está compuesto por dos símbolos ||. Un solo símbolo ! es un operador diferente, el cual se describe en el día 21.

## Operador lógico NOT

Una expresión lógica NOT es verdadera (true) si la expresión que se prueba es falsa (false). De nuevo, si la expresión que se está probando es falsa, ¡el valor de la prueba es verdadero! Por lo tanto,

```
if ( ! (x == 5) )
```

es verdadero sólo si x no es igual a 5. Esto es lo mismo que escribir

```
if (x != 5)
```

## Evaluación de corto circuito

Cuando el compilador evalúa una expresión AND como la siguiente:

```
if ( (x == 5) && (y == 5) )
```

comprobará si la primera expresión ( $x == 5$ ) es verdadera, y si esto falla (es decir, si x no es igual a 5), el compilador NO evaluará si la segunda expresión ( $y == 5$ ) es cierta o falsa, debido a que AND requiere que ambas sean verdaderas.

Asimismo, si el compilador evalúa una expresión OR como la siguiente:

```
if ( (x == 5) || (y == 5) )
```

si la primera expresión es verdadera ( $x == 5$ ), el compilador nunca evaluará la segunda expresión ( $y == 5$ ), debido a que en una expresión OR basta con que *cualquiera* de las dos sea verdadera.

Esto es importante si hay efectos secundarios (por ejemplo, si hubiera utilizado los operadores de incremento o decremento en x o en y). Si no se evaluara la segunda expresión, no habría incremento o decremento.

## Precedencia relacional

Debido a que en C++ los operadores relacionales y los operadores lógicos son expresiones, cada uno de ellos regresa un valor: verdadero o falso. Al igual que todas las expresiones, tienen un orden de precedencia (vea el apéndice A) que determina cuáles relaciones se evalúan primero. Este hecho es importante al determinar el valor de la expresión, por ejemplo:

```
if (x > 5 && y > 5 || z > 5)
```

Tal vez el programador quería que esta expresión fuera verdadera (true) si tanto x como y eran mayores que 5, o si z era mayor que 5. Por otra parte, tal vez el programador haya

querido que esta expresión fuera verdadera sólo si  $x$  era mayor que 5 y si también era verdad que  $y$  era mayor que 5, o que  $z$  era mayor que 5.

Si  $x$  es igual a 3, y tanto  $y$  como  $z$  tienen el valor 10, la primera interpretación sería verdad ( $z$  es mayor que 5, por lo que se ignoran  $x$  y  $y$ ), pero la segunda sería falsa (no es verdad que  $x$  es mayor que 5 y por consecuencia no importa qué haya del lado derecho del símbolo `&&` ya que ambos lados deben ser verdaderos).

Aunque la precedencia determina cuál relación se evalúa primero, los paréntesis pueden cambiar el orden y hacer que la instrucción sea más clara:

```
if ( (x > 5) && (y > 5 || z > 5) )
```

Si se utilizan los mismos valores que en el ejemplo anterior, esta expresión es falsa.

Como no es verdad que  $x$  es mayor que 5, el lado izquierdo de la expresión AND falla, y por consecuencia toda la expresión es falsa. Recuerde que una expresión AND requiere que ambos lados sean verdaderos (algo no es “sabroso” y “bueno para usted” si no sabe bien).

### Nota

Por lo general es una buena idea utilizar paréntesis adicionales para dejar más en claro lo que se quiere agrupar. Recuerde, el objetivo es escribir programas que funcionen y que sean fáciles de leer y de comprender.

Siempre que tenga dudas sobre el orden de las operaciones, utilice paréntesis. Esto ocasionará que el compilador utilice unos cuantos milisegundos más, lo que probablemente no se notará al momento de la ejecución.

4

## Más sobre falso y verdadero

En C++, un cero se evalúa como falso, y todos los demás valores se evalúan como verdaderos. Debido a que una expresión siempre tiene un valor, muchos programadores de C++ sacan ventaja de esta característica en sus instrucciones `if`. Una instrucción como la siguiente:

```
if (x) // si x es verdadero (distinto de cero)
    x = 0;
```

se puede leer de la siguiente manera: “Si  $x$  tiene un valor distinto de cero, asignarle el valor 0”. Aquí estamos haciendo una poca de trampa; sería más claro si se escribe

```
if (x != 0) // si x es distinto de cero
    x = 0;
```

Ambas instrucciones son válidas, pero la última es más clara. Es una buena práctica de programación reservar el primer método para verdaderas pruebas de lógica, en lugar de usarlo para probar si hay valores distintos de cero.

Estas dos instrucciones también son equivalentes:

```
if (!x)           // si x es falso (cero)
if (x == 0)       // si x es cero
```

Sin embargo, la segunda instrucción es un poco más fácil de comprender y es más explícita si usted está probando el valor matemático de *x* en vez de su estado lógico.

DEBE	No DEBE
<b>DEBE</b> colocar paréntesis alrededor de sus pruebas lógicas para que sean más claras y para que la precedencia sea explícita.	<b>NO DEBE</b> utilizar <code>if (x)</code> como sinónimo para <code>if (x != 0)</code> ; este último es más claro.
<b>DEBE</b> utilizar llaves en instrucciones <code>if</code> anidadas para que las cláusulas <code>else</code> sean más claras y para evitar errores.	<b>NO DEBE</b> utilizar <code>if (!x)</code> como sinónimo para <code>if (x == 0)</code> ; este último es más claro.

## Operador condicional (ternario)

El operador condicional (`:?`) es el único operador ternario de C++; es decir, es el único operador que toma tres términos.

El operador condicional toma tres expresiones y regresa un valor:

```
(expresión1) ? (expresión2) : (expresión3)
```

Esta línea se lee de la siguiente manera: "Si `expresión1` es verdadera, regresar el valor de `expresión2`; de no ser así, regresar el valor de `expresión3`". Por lo general, este valor se asignará a una variable.

El listado 4.10 muestra una instrucción `if` escrita utilizando el operador condicional.

### ENTRADA

### LISTADO 4.10 Una muestra del operador condicional

```
1:  // Listado 4.10 - Muestra el uso del operador condicional
2:  //
3:  #include <iostream.h>
4:  int main()
5:  {
6:      int x, y, z;
7:      cout << "Escriba dos números.\n";
8:      cout << "Primero: ";
9:      cin >> x;
10:     cout << "\nSegundo: ";
11:     cin >> y;
12:     cout << "\n";
```

```
13:  
14:    if (x > y)  
15:        z = x;  
16:    else  
17:        z = y;  
18:  
19:    cout << "z: " << z;  
20:    cout << "\n";  
21:  
22:    z = (x > y) ? x : y;  
23:  
24:    cout << "z: " << z;  
25:    cout << "\n";  
26:    return 0;  
27: }
```

**SALIDA**

Escriba dos números.  
Primero: 5

Segundo: 8

z: 8  
z: 8

**ANÁLISIS**

Se crean tres variables de tipo entero: x, y y z. El usuario asigna un valor a las dos primeras. La instrucción if de la línea 14 hace una prueba para ver cuál es más grande y asigna el valor más grande a z. Este valor se imprime en la línea 19.

4

El operador condicional de la línea 22 hace la misma prueba y asigna el valor más grande a z. Se lee así: “Si x es mayor que y, regresar el valor de x; de no ser así, regresar el valor de y”. El valor regresado se asigna a z. Ese valor se imprime en la línea 24. Como puede ver, la instrucción condicional es un equivalente más corto para la instrucción if...else.

## Resumen

En esta lección se ha cubierto bastante material. Ha aprendido lo que son las instrucciones y las expresiones de C++, lo que hacen los operadores de C++ y cómo funcionan las instrucciones if de C++.

También ha visto que en cualquier parte donde pueda utilizar una instrucción sencilla, también puede utilizar un bloque de instrucciones encerradas por un par de llaves ({y}).

Asimismo, ha aprendido que todas las expresiones se evalúan y producen un cierto valor, y que ese valor se puede probar en una instrucción if o mediante el operador condicional. También ha visto cómo evaluar varias instrucciones por medio del operador lógico, cómo comparar valores por medio de los operadores relacionales y cómo asignar valores por medio del operador de asignación.

También ha explorado la precedencia de los operadores, y ha visto la forma en que se pueden utilizar los paréntesis para cambiar la precedencia a fin de hacerla explícita y, por consecuencia, más fácil de manejar.

## Preguntas y respuestas

- P** ¿Por qué utilizar paréntesis innecesarios si la precedencia determina cuáles operadores se evalúan primero?
- R** Aunque es cierto que el compilador conoce la precedencia y que un programador puede consultar el orden de precedencia, un código fácil de comprender es más fácil de mantener.
- P** Si los operadores relacionales siempre regresan verdadero o falso, ¿por qué cualquier valor distinto de cero se considera verdadero?
- R** Los operadores relacionales regresan verdadero o falso, pero toda expresión regresa un valor, y ese valor también se pueden evaluar en una instrucción `if`. He aquí un ejemplo:

```
if ( (x = a + b) == 35 )
```

Ésta es una instrucción perfectamente válida en C++. Tiene un valor aunque la suma de `a` y `b` no sea igual a 35. Además, hay que observar que en cualquier caso el valor de la suma de `a` y `b` se asigna a `x`.

- P** ¿Qué efecto tienen en un programa los tabuladores, espacios y caracteres de nueva línea?
- R** Los tabuladores, espacios y caracteres de nueva línea (con los que se crean los espacios en blanco) no tienen efecto en el programa, aunque el uso sensato del espacio en blanco puede facilitar la legibilidad del programa.
- P** ¿Qué son los números negativos, verdaderos o falsos?
- R** Todos los valores distintos de cero, positivos o negativos, son verdaderos.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, "Respuestas a los cuestionarios y ejercicios", y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Qué es una expresión?
2. ¿Es  $x = 5 + 7$  una expresión? ¿Cuál es su valor?

3. ¿Cuál es el valor de  $201 / 4$ ?
4. ¿Cuál es el valor de  $201 \% 4$ ?
5. Si `miEdad`, `a` y `b` son variables de tipo `int`. ¿cuáles son sus valores después de ejecutar las siguientes instrucciones?  
`miEdad = 39;`  
`a = miEdad++;`  
`b = ++miEdad;`
6. ¿Cuál es el valor de  $8+2*3$ ?
7. ¿Cuál es la diferencia entre `x = 3` y `x == 3`?
8. ¿Qué son los siguientes valores, verdaderos o falsos?
  - a. 0
  - b. 1
  - c. -1
  - d. `x = 0`
  - e. `x == 0` // suponga que `x` vale 0

## Ejercicios

1. Escriba una instrucción `if` sencilla que examine dos variables de tipo entero y que cambie la más grande a la más pequeña, usando sólo una cláusula `else`.
2. Examine el siguiente programa. Imagine que escribe tres números, y escriba la salida que espera obtener.

```
1: #include <iostream.h>
2: int main()
3: {
4:     int a, b, c;
5:     cout << "Escriba tres números\n";
6:     cout << "a:    ";
7:     cin >> a;
8:     cout << "\nb:    ";
9:     cin >> b;
10:    cout << "\nc:    ";
11:    cin >> c;
12:
13:    if (c = (a-b))
14:    {
15:        cout << "a:    ";
16:        cout << a;
17:        cout << "menos b:    ";
18:        cout << b;
19:        cout << "igual a c:    ";
20:        cout << c << endl;
21    }
22:    else
23:        cout << "a-b no es igual a c:    " << endl;
24:    return 0;
25: }
```

4

3. Escriba el programa del ejercicio 2; compílelo, enlace y ejecútelo. Escriba los números 20, 10 y 50. ¿Obtuvo la salida esperada? ¿Por qué no?

4. Examine este programa y trate de adivinar la salida.

```
1: #include <iostream.h>
2: int main()
3: {
4:     int a = 1, b = 1, c;
5:     if (c = (a·b))
6:         cout << "El valor de c es: " <<c;
7:     return 0;
8: }
```

5. Escriba, compile, enlace y ejecute el programa del ejercicio 4. ¿Cuál fue la salida? ¿Por qué?

# SEMANA 1

## DÍA 5

### Funciones

Aunque la programación orientada a objetos ha desviado la atención de las funciones hacia los objetos, las funciones siguen siendo, sin lugar a dudas, un componente central de cualquier programa. Hoy aprenderá lo siguiente:

- Qué es una función y cuáles son sus componentes
- Cómo declarar y definir funciones
- Cómo pasar parámetros a las funciones
- Cómo regresar un valor de una función
- Cómo crear y utilizar bibliotecas de funciones
- Qué son las bibliotecas estándar y cuál es su contenido

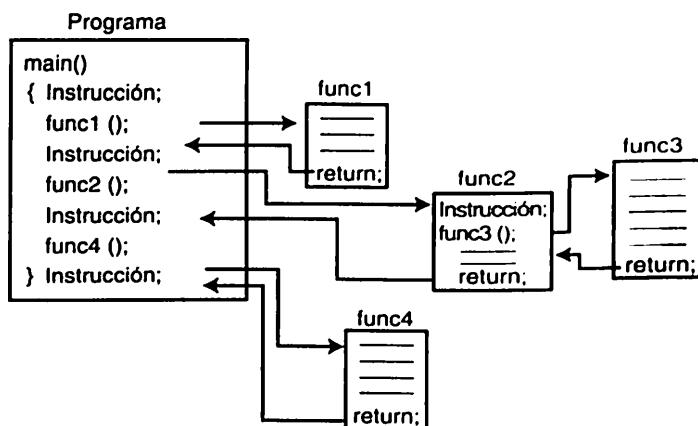
#### Qué es una función

Una función es, en efecto, un subprograma que puede actuar sobre los datos y regresar un valor. Cada programa de C++ tiene por lo menos una función: `main()`. Cuando su programa inicia, el sistema operativo llama a `main()` de forma automática. `main()` podría llamar a otras funciones, algunas de las cuales podrían también llamar a otras.

Cada función tiene su propio nombre, y cuando se encuentra ese nombre, la ejecución del programa se dirige hacia el cuerpo de esa función. Esto se conoce como *llamar* a la función. Cuando la función termina, la ejecución continúa en la siguiente línea de la función que realizó la llamada. Este flujo se muestra en la figura 5.1.

**FIGURA 5.1**

*Cuando un programa llama a una función, la ejecución continúa dentro de la función y luego regresa a la línea que está después de la llamada a la función.*



Las funciones bien diseñadas realizan una tarea específica y clara. Las tareas complicadas se deben dividir entre varias funciones, y luego se puede llamar en orden a cada una de ellas.

Las funciones vienen en dos variedades: definidas por el usuario e integradas. Las funciones *integradas* son parte del paquete del compilador (las proporciona el fabricante para que usted las utilice). Las funciones *definidas por el usuario* son las funciones que usted mismo escribe.

## Valores de retorno, parámetros y argumentos

Las funciones pueden *regresar* un valor. Al llamar a una función, ésta puede hacer su trabajo y luego regresar un valor como resultado de ese trabajo. Este valor se conoce como *valor de retorno*, y usted debe declarar la función con el tipo de ese valor de retorno. Por lo tanto, si usted escribe

```
int miFuncion();
```

está declarando que *miFuncion* regresará un valor de tipo entero.

También puede enviar valores *hacia* las funciones. La descripción de los valores enviados se conoce como *lista de parámetros*.

```
int miFuncion(int algunEntero, float algunFlotante);
```

Esta declaración indica que *miFuncion* no sólo regresará un entero, sino que también recibirá un valor de tipo *int* y uno de tipo *float* como parámetros.

Un parámetro describe el *tipo* del valor que se pasará hacia la función, así como el nombre de la variable utilizada en la función cuando ésta es llamada. Los valores reales que usted pasa a la función se conocen como *argumentos*.

```
int elValorRegresado = miFuncion(5, 6.7);
```

Puede ver aquí que la variable de tipo entero `elValorRegresado` se inicializa con el valor regresado por `miFuncion`, y que los valores 5 y 6.7 se pasan como argumentos. El tipo de los argumentos debe concordar con los tipos de los parámetros declarados.

## Declaración y definición de funciones

El uso de funciones en su programa requiere que primero declare la función y que luego la defina. La declaración le indica al compilador el nombre, el tipo de valor de retorno y los parámetros de la función. La definición le indica al compilador cómo trabaja la función. Ninguna función debe ser llamada desde otra función si no ha sido declarada. La declaración de una función se conoce como *prototipo*.

### Declaración de una función

Existen tres maneras de declarar una función:

- Escribir su prototipo en un archivo, y luego utilizar la directiva `#include` para incluirlo en su programa.
- Escribir el prototipo dentro del archivo en el que se utiliza su función.
- Definir la función antes de llamarla desde cualquier otra función. Al hacer esto, la definición actúa como su propia declaración.

Aunque puede definir la función antes de usarla y, por ende, evitar la necesidad de crear un prototipo de función, ésta no es una buena práctica de programación por tres razones.

En primer lugar, es mala idea hacer que las funciones aparezcan en un archivo en un orden específico. Esto dificulta el mantenimiento del programa a medida que cambian los requerimientos.

En segundo lugar, es posible que la función A() necesite llamar a la función B(), pero la función B() también puede necesitar llamar a la función A() bajo ciertas circunstancias. No es posible definir la función A() antes de definir la función B(), ni tampoco definir la función B() antes de definir la función A(), así que por lo menos una de ellas debe ser declarada en cualquier caso.

En tercer lugar, los prototipos de funciones son una técnica de depuración buena y poderosa. Si su prototipo declara que su función recibe un conjunto específico de parámetros o que regresa un tipo específico de valor, y luego su función no concuerda con el prototipo, el compilador puede emitir un mensaje de error en lugar de esperar a que éste aparezca cuando ejecute el programa.

## Uso de los prototipos de funciones

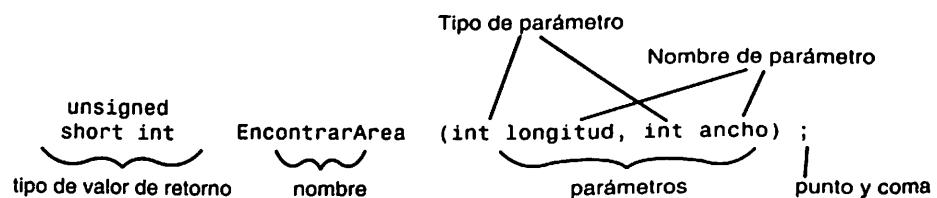
Muchas de las funciones integradas que utilice tendrán sus prototipos de función ya escritos en los archivos que incluya en su programa por medio de `#include`. Debe incluir el prototipo para las funciones que escriba usted mismo.

El prototipo de una función es una instrucción, lo que significa que termina con un punto y coma. Consta del tipo de valor de retorno de la función y la firma. La firma de una función es su nombre y su lista de parámetros.

La lista de parámetros es una lista de todos los parámetros y sus tipos, separados por comas. La figura 5.2 muestra los componentes del prototipo de una función.

**FIGURA 5.2**

*Componentes de un prototipo de función.*



El prototipo y la definición de la función deben concordar exactamente en el tipo de valor de retorno y la firma. Si no concuerdan, obtendrá un error en tiempo de compilación. No obstante, debe tener en cuenta que el prototipo de la función no necesita contener los nombres de los parámetros, sólo sus tipos. Un prototipo como el siguiente es perfectamente válido:

```
long Area(int, int);
```

Este prototipo declara una función llamada `Area()` que regresa un valor de tipo `long` y que tiene dos parámetros, ambos de tipo entero. Aunque esto es válido, no es una buena idea. Agregar los nombres de los parámetros ayuda a que el prototipo sea más claro. La siguiente es la misma función con los nombres de los parámetros:

```
long Area(int longitud, int ancho);
```

Ahora es obvio lo que esta función hace y qué valores contendrán los parámetros.

El compilador no requiere los nombres de las variables y esencialmente los ignora en el prototipo. Están para nosotros, los humanos.

Observe que todas las funciones tienen un tipo de valor de retorno. Si no se declara explícitamente uno, el tipo de valor de retorno predeterminado es `int`. Sin embargo, sus programas serán más fáciles de entender si declara explícitamente el tipo de valor de retorno para cada función, incluyendo el de `main()`. El listado 5.1 muestra un programa que incluye un prototipo de función para la función `Area()`.

**ENTRADA****LISTADO 5.1** La declaración de una función y la definición y el uso de esa función

```
1: // Listado 5.1 - Muestra el uso de los prototipos de funciones
2: //
3: #include <iostream.h>
4: int Area(int longitud, int ancho); //prototipo de la función
5:
6: int main()
7: {
8:     int longitudDeJardin;
9:     int anchoDeJardin;
10:    int areaDeJardin;
11:
12:    cout << "\n¿Cuál es el ancho de su jardín? ";
13:    cin >> anchoDeJardin;
14:    cout << "\n¿Cuál es la longitud de su jardín? ";
15:    cin >> longitudDeJardin;
16:
17:    areaDeJardin= Area(longitudDeJardin,anchoDeJardin);
18:
19:    cout << "\nSu jardín es de ";
20:    cout << areaDeJardin;
21:    cout << " metros cuadrados\n\n";
22:    return 0;
23: }
24:
25: int Area(int jardinLongitud, int jardinAncho)
26: {
27:     return jardinLongitud * jardinAncho;
28: }
```

**SALIDA**

```
¿Cuál es el ancho de su jardín? 100
¿Cuál es la longitud de su jardín? 200
Su jardín es de 20000 metros cuadrados
```

5

**ANÁLISIS**

El prototipo para la función `Area()` se encuentra en la línea 4. Compare el prototipo con la definición de la función de la línea 25. Observe que el nombre, el tipo de valor de retorno y los tipos de los parámetros son los mismos. Si fueran diferentes, se habría generado un error de compilación. De hecho, la única diferencia requerida es que el prototipo de la función termine con un punto y coma y que no tenga cuerpo.

Observe también que los nombres de los parámetros del prototipo son `longitud` y `ancho`, pero los nombres de los parámetros en la definición son `jardinLongitud` y `jardinAncho`. Como vio anteriormente, los nombres que vienen en el prototipo no se utilizan; están como información para el programador. Es una buena práctica de programación hacer que los nombres de los parámetros del prototipo concuerden con los nombres de los parámetros de la implementación, pero esto no es un requerimiento.

Los argumentos se pasan a la función en el orden en que están declarados y definidos, pero no se comparan los nombres. Si usted hubiera pasado `anchoDeJardin` seguido de `longitudDeJardin`, la función `Area()` habría utilizado el valor de `anchoDeJardin` para `jardinLongitud` y el valor de `longitudDeJardin` para `jardinAncho`. El cuerpo de la función siempre está encerrado entre llaves, aunque conste sólo de una instrucción, como en este caso.

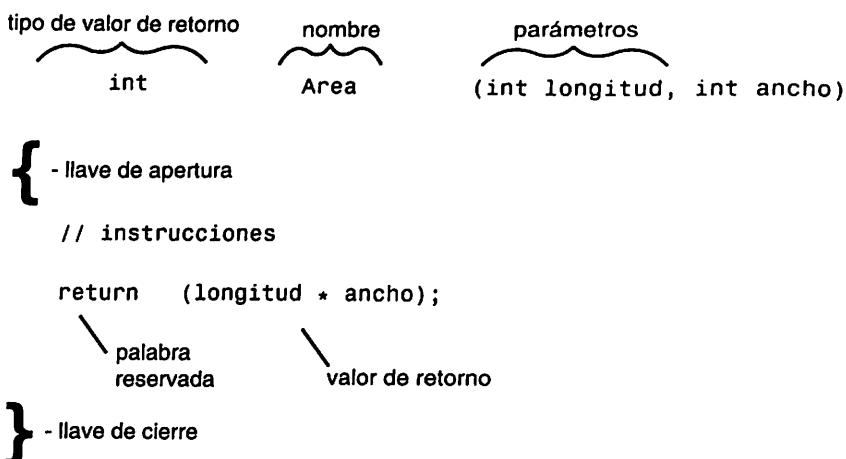
## Definición de una función

La definición de una función consiste en el encabezado de la función y su cuerpo. El encabezado es como el prototipo de la función, sólo que los parámetros deben tener nombre, y no se utiliza punto y coma al final.

El cuerpo de la función es un conjunto de instrucciones encerradas entre llaves. La figura 5.3 muestra el encabezado y el cuerpo de una función.

**FIGURA 5.3**

*El encabezado y el cuerpo de una función.*



### Funciones

#### Sintaxis del prototipo de función:

```
tipo_valor_retorno nombre_funcion ( [tipo [nombreParametro]]... );
```

#### Sintaxis de la definición de función:

```
tipo_valor_retorno nombre_funcion ( [tipo nombreParametro]... )
{
    instrucciones;
}
```

Un prototipo de función le indica al compilador el tipo del valor de retorno, el nombre de la función y la lista de parámetros. No se requiere que las funciones tengan parámetros, y si los tienen, no se requiere que el prototipo muestre sus nombres, sólo sus tipos. Un prototipo siempre termina con punto y coma (;

Una definición de función debe concordar con el tipo de valor de retorno y la lista de parámetros con su prototipo. Se deben proporcionar nombres para todos los parámetros, y el cuerpo de la definición de la función debe estar encerrado entre llaves. Todas las instrucciones que estén dentro del cuerpo de la función deben terminar con punto y coma, pero la función en sí no termina con punto y coma (;). Termina con una llave de cierre.

Si la función regresa un valor, éste debe terminar con la instrucción `return`, aunque las instrucciones `return` pueden aparecer en cualquier parte del cuerpo de la función.

Toda función tiene un tipo de valor de retorno. Si no se designa uno en forma explícita, el tipo de valor de retorno será `int`. Asegúrese de dar a todas las funciones un tipo de valor de retorno explícito. Si una función no regresa un valor, su tipo de valor de retorno será `void`.

#### Ejemplos de prototipos de funciones:

```
long EncontrarArea(long longitud, long ancho);
  // regresa un entero largo, tiene dos parámetros
void ImprimeMensaje (int numeroMensaje);
  // regresa void (es decir, no regresa nada), tiene un parámetro
int ObtenerOpcion();
  // regresa un entero, no tiene parámetros
FuncionMala();                                // regresa un entero, no tiene
                                                parámetros
```

#### Ejemplos de definición de función:

```
long EncontrarArea(long l, long a)
{
    return l * a;
}

void ImprimeMensaje(int cualMsg)
{
    if (cualMsg == 0)
        cout << "Hola.\n";
    if (cualMsg == 1)
        cout << "Adiós.\n";
    if (cualMsg > 1)
        cout << "Estoy confundido.\n";
}
```

5

## Ejecución de una función

Al llamar a una función, la ejecución empieza con la primera instrucción que va después de la llave de apertura ({). La ramificación se puede lograr mediante el uso de la instrucción `if` (y las instrucciones relacionadas que verá en el día 7, “Más flujo de programa”). Las funciones también pueden llamar a otras funciones e inclusive pueden llamarse a sí mismas (vea la sección “Recursión”, que se muestra más adelante en este día).

## Variables locales

No solamente puede pasar variables a la función, también puede declarar variables dentro del cuerpo de la función. Esto se hace utilizando variables locales, que se llaman así porque existen sólo localmente dentro de la misma función. Al salir de la función, las variables locales ya no están disponibles.

Las variables locales se definen de la misma forma que cualquier otra variable. Los parámetros que se pasan a la función también se consideran variables locales y se pueden usar como si se hubieran definido dentro del cuerpo de la función. El listado 5.2 es un ejemplo del uso de parámetros y variables definidas localmente dentro de una función.

**ENTRADA****LISTADO 5.2** El uso de variables locales y parámetros

```
1: #include <iostream.h>
2:
3: float Convertir(float);
4: int main()
5: {
6:     float TempFar;
7:     float TempCen;
8:
9:     cout << "Escriba la temperatura en grados Fahrenheit: ";
10:    cin >> TempFar;
11:    TempCen = Convertir(TempFar);
12:    cout << "\nAquí está la temperatura en grados centígrados: ";
13:    cout << TempCen << endl;
14:    return 0;
15: }
16:
17: float Convertir(float TempFar)
18: {
19:     float TempCen;
20:
21:     TempCen = ((TempFar - 32) * 5) / 9;
22:     return TempCen;
23: }
```

**SALIDA**

Escriba la temperatura en grados Fahrenheit: 212

Aquí está la temperatura en grados centígrados: 100

Escriba la temperatura en grados Fahrenheit: 32

Aquí está la temperatura en grados centígrados: 0

Escriba la temperatura en grados Fahrenheit: 85

Aquí está la temperatura en grados centígrados: 29.4444

**ANÁLISIS** En las líneas 6 y 7 se declaran dos variables de tipo `float`, una para guardar la temperatura en grados Fahrenheit y la otra para guardar la temperatura en grados centígrados. En la línea 9 se pide al usuario que escriba la temperatura en grados Fahrenheit, y ese valor se pasa a la función `Convertir()`.

La ejecución salta hasta la primera línea de la función `Convertir()`, la línea 19, en donde se declara una variable local, que se llama también `TempCen`. Observe que esta variable local no es la misma que la variable `TempCen` de la línea 7. Esta variable existe sólo dentro de la función `Convertir()`. El valor que se pasa como parámetro, `TempFar`, también es sólo una copia local de la variable que `main()` pasa a la función.

Esta función hubiera podido nombrar `FarTemp` al parámetro y `CenTemp` a la variable local, y el programa funcionaría igual de bien. Puede escribir estos nombres y volver a compilar el programa para ver su funcionamiento.

El valor que se obtiene al restar 32 al parámetro `TempFar`, multiplicar el resultado por 5 y luego dividirlo entre 9 se asigna a la variable local `TempCen`. El valor resultante se regresa entonces como el valor de retorno de la función, y se asigna a la variable `TempCen`, línea 11, de la función `main()`. El valor se imprime en la línea 13.

El programa se ejecuta tres veces. La primera vez se pasa el valor 212 para comprobar que el punto de ebullición del agua en grados Fahrenheit (212) genere la respuesta correcta en grados centígrados (100). La segunda prueba es el punto de congelación del agua. La tercera prueba es un número aleatorio elegido para generar un resultado decimal.

Como ejercicio, escriba el programa otra vez con otros nombres de variables, como se muestra a continuación:

```
1: #include <iostream.h>
2:
3: float Convertir(float);
4: int main()
5: {
6:     float TempFar;
7:     float TempCen;
8:
9:     cout << "Escriba la temperatura en grados Fahrenheit: ";
10:    cin >> TempFar;
11:    TempCen = Convertir(TempFar);
12:    cout << "\nAquí está la temperatura en grados centígrados: ";
13:    cout << TempCen << endl;
14:    return 0;
15: }
16:
17: float Convertir(float GradFar)
18: {
19:     float GradCen;
```

```

21:     GradCen = ((GradFar + 32) * 5) / 9;
22:     return GradCen;
23: }
```

Los resultados son los mismos.

Se dice que una variable tiene un alcance, el cual determina cuánto tiempo estará disponible para el programa y en dónde se puede utilizar. Las variables declaradas dentro de un bloque tienen su alcance limitado a ese bloque; se pueden acceder sólo dentro de ese bloque y “dejan de existir” cuando ese bloque termina. Las variables globales tienen alcance global y están disponibles en cualquier parte del programa.

Por lo general el alcance es obvio, pero existen algunas excepciones engañosas. Verá más sobre esto cuando hablemos de los ciclos `for` en el día 7.

Nada de esto importa mucho si usted tiene cuidado de no volver a utilizar los mismos nombres de variables dentro de cualquier otra función.

## Variables globales

Las variables que se definen fuera de cualquier función tienen alcance global y, por lo tanto, están disponibles para cualquier función del programa, incluyendo `main()`.

Las variables locales que tengan el mismo nombre que las variables globales no cambian a las variables globales. Sin embargo, una variable local que tenga el mismo nombre que una variable global *oculta* a la variable global. Si una función tiene una variable con el mismo nombre que una variable global, el nombre se referirá a la variable local, no a la global, cuando se utilice dentro de la función. El listado 5.3 ilustra estos puntos.

### ENTRADA LISTADO 5.3 Muestra de variables globales y locales

```

1: #include <iostream.h>
2: void miFuncion();           // prototipo
3:
4: int x = 5, y = 7;          // variables globales
5: int main()
6: {
7:
8:     cout << "x desde la función main: " << x << "\n";
9:     cout << "y desde la función main: " << y << "\n\n";
10:    miFuncion();
11:    cout << "¡Ya salimos de miFuncion!\n\n";
12:    cout << "x desde la función main: " << x << "\n";
13:    cout << "y desde la función main: " << y << "\n";
14:    return 0;
15: }
16:
```

```
17: void miFuncion()
18: {
19:     int y = 10;
20:
21:     cout << "x desde miFuncion: " << x << "\n";
22:     cout << "y desde miFuncion: " << y << "\n\n";
23: }
```

**SALIDA**

```
x desde la función main: 5
y desde la función main: 7

x desde miFuncion: 5
y desde miFuncion: 10

¡Ya salimos de miFuncion!

x desde la función main: 5
y desde la función main: 7
```

**ANÁLISIS**

Este programa sencillo ilustra unos cuantos puntos clave, y potencialmente engañosos, sobre las variables locales y las globales. En la línea 4 se declaran dos variables globales, `x` y `y`. La variable global `x` se inicializa con el valor 5, y la variable global `y` se inicializa con 7.

En las líneas 8 y 9, dentro de la función `main()`, estos valores se imprimen en la pantalla. Observe que la función `main()` no define ninguna de las dos variables; debido a que son globales, ya están disponibles para `main()`.

Cuando se llama a `miFuncion()` en la línea 10, la ejecución del programa pasa a la línea 18, se define una variable local llamada `y`, y se inicializa con el valor 10. En la línea 21, `miFuncion()` imprime el valor de la variable `x`, y el valor de la variable global `x` se utiliza como en `main()`. Sin embargo, cuando se utiliza el nombre de variable `y`, en la línea 22, se utiliza la variable local `y`, que oculta a la variable global que tiene el mismo nombre.

La llamada a la función termina, y el control regresa a `main()`, que vuelve a imprimir los valores de las variables globales. Observe que la variable global `y` no fue afectada por el valor asignado a la variable local `y` de `miFuncion()`.

5

## Variables globales: una advertencia

En C++, las variables globales son válidas, pero casi nunca se utilizan. C++ se originó a partir de C, y en C las variables globales son una herramienta peligrosa, pero necesaria. Son necesarias porque hay veces que el programador necesita hacer que algunos datos estén disponibles para muchas funciones, y no quiere pasar esa información como parámetro de función en función.

Las variables globales son peligrosas porque son información compartida, y una función puede cambiar una variable global de manera que el cambio sea invisible para otra función. Esto crea errores que son muy difíciles de encontrar.

En el día 14, “Clases y funciones especiales”, verá una poderosa alternativa que ofrece C++ para las variables globales, pero que no está disponible en C.

## Más acerca de las variables locales

Se dice que las variables declaradas dentro de la función tienen *alcance local*. Esto significa, como se dijo anteriormente, que son visibles y se pueden utilizar sólo dentro de la función en la que se definen. De hecho, en C++ usted puede definir variables en cualquier parte de la función, no sólo al principio. El alcance de la variable es el bloque en el que ésta se define. Por lo tanto, si define una variable dentro de un par de llaves que se encuentren dentro de una función, la variable estará disponible sólo dentro de ese bloque. El listado 5.4 ilustra esta idea.

**ENTRADA****LISTADO 5.4** Variables con alcance de bloque

---

```
1: // Listado 5.4 - muestra de variables
2: // que tienen alcance de bloque
3:
4: #include <iostream.h>
5:
6: void miFunc();
7:
8: int main()
9: {
10:     int x = 5;
11:     cout << "\nEn main x vale: " << x;
12:
13:     miFunc();
14:
15:     cout << "\nDe regreso en main, x vale: " << x << endl;
16:     return 0;
17: }
18:
19: void miFunc()
20: {
21:
22:     int x = 8;
23:     cout << "\nEn miFunc, la variable local x vale: " << x << endl;
24:
25:     {
26:         cout << "\nEn el bloque de miFunc, x vale: " << x;
27:
28:         int x = 9;
```

```
29:  
30:     cout << "\nLa misma variable local x vale: " << x;  
31: }  
32:  
33: cout << "\nFuera del bloque, en miFunc, x vale: " << x << endl;  
34: }
```

**SALIDA**

```
En main x vale: 5  
En miFunc, la variable local x vale: 8  
  
En el bloque de miFunc, x vale: 8  
La misma variable local x vale: 9  
Fuera del bloque, en miFunc, x vale: 8  
De regreso en main, x vale: 5
```

**ANÁLISIS**

Este programa empieza en la línea 10 con la inicialización de la variable local `x` de `main()`. La impresión de la línea 11 verifica que `x` haya sido inicializada con el valor 5.

Se llama a `miFunc()`, y en la línea 22 se inicializa una variable local, también llamada `x`, con el valor 8. Su valor se imprime en la línea 23.

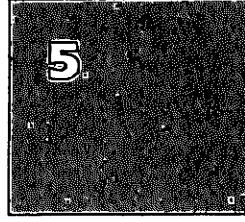
En la línea 25 se inicia un bloque, y la variable `x` de la función se imprime otra vez en la línea 26. En la línea 28 se crea una nueva variable, también llamada `x`, pero que es local para el bloque, y se inicializa con el valor 9.

El valor de la nueva variable `x` se imprime en la línea 30. El bloque local termina en la línea 31, y la variable creada en la línea 28 queda “fuera de alcance”, por lo que no se puede ver ni utilizar.

Al imprimir `x` en la línea 33, se imprime la `x` que se declaró en la línea 22. Esta `x` no resultó afectada por la `x` que se definió en la línea 28; su valor sigue siendo 8.

En la línea 34, `miFunc()` queda fuera de alcance, y su variable local `x` no se puede utilizar. La ejecución regresa a la línea 15, y se imprime el valor de la variable local `x` que se creó en la línea 10. Esta variable no resultó afectada por ninguna de las variables definidas en `miFunc()`.

Sin necesidad de decirlo, ¡este programa hubiera sido mucho menos confuso si estas tres variables tuvieran nombres distintos!

5

## Instrucciones de una función

No existe casi ningún límite para la cantidad o tipos de instrucciones que se pueden colocar en el cuerpo de una función. Aunque no puede definir una función dentro de otra función, sí puede *llamar* a una función, y desde luego que `main()` hace justo eso casi en cualquier programa de C++. Las funciones pueden inclusive llamarse a sí mismas, lo cual se trata en la sección que habla sobre la recursión.

Aunque en C++ no existe un límite para el tamaño de una función, las funciones bien diseñadas tienden a ser pequeñas. Muchos programadores aconsejan mantener las funciones lo suficientemente cortas como para que quepan en una sola pantalla, con el fin de que se pueda ver toda la función a la vez. Ésta es una regla empírica que a menudo quebrantan los programadores que son muy buenos, pero una función más pequeña es más fácil de comprender y de mantener.

Cada función debe realizar una tarea individual y fácil de comprender. Si su función empieza a crecer, busque lugares en donde pueda dividirla en tareas más pequeñas.

## Más acerca de los argumentos de funciones

Los argumentos de funciones no tienen que ser todos del mismo tipo. Es perfectamente razonable escribir una función que tome un entero, dos enteros largos y un carácter como sus argumentos.

Cualquier expresión válida de C++ puede ser un argumento de función, incluyendo constantes, expresiones matemáticas y lógicas y otras funciones que regresen un valor.

### Uso de funciones como parámetros para otras funciones

Aunque es válido para una función tomar como parámetro una segunda función que regrese un valor, esto puede ocasionar que el código sea difícil de leer y de depurar.

Como ejemplo, suponga que tiene las funciones `doble()`, `triple()`, `cuadrado()` y `cubo()`, cada una de las cuales regresa un valor. Podría escribir

```
Respuesta = (doble(triple(cuadrado(cubo(miValor)))));
```

Esta instrucción toma una variable, `miValor`, y la pasa como argumento para la función `cubo()`, cuyo valor de retorno se pasa como argumento a la función `cuadrado()`, cuyo valor de retorno se pasa a su vez a `triple()`, y ese valor de retorno se pasa a `doble()`. El valor de retorno de este número duplicado, triplicado, elevado al cuadrado y elevado al cubo se pasa entonces a `Respuesta`.

Es difícil estar seguro de lo que hace este código (¿era triplicado el valor antes o después de ser elevado al cuadrado?), y si la respuesta es equivocada, será difícil averiguar cuál función falló.

Una alternativa es asignar cada paso a su propia variable intermedia:

```
unsigned long miValor = 2;
unsigned long alcubo = cubo(miValor);           // alcubo = 8
unsigned long alcuadrado = cuadrado(alcubo);    // alcuadrado = 64
unsigned long triplicado = triple(alcuadrado); // triplicado = 192
unsigned long Respuesta = doble(triplicado);   // Respuesta = 384
```

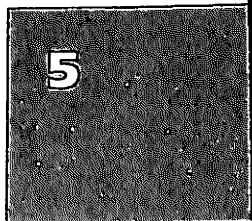
Ahora cada resultado intermedio puede ser examinado, y el orden de ejecución es explícito.

## Los parámetros son variables locales

Los argumentos que se pasan a una función son locales para esa función. Los cambios hechos a los argumentos no afectan los valores de la función que hace la llamada. Esto se conoce como *paso de parámetros por valor*, lo que significa que se hace una copia local de cada argumento de la función. Estas copias locales se tratan de la misma forma que cualquier otra variable local. El listado 5.5 ilustra este punto.

### ENTRADA LISTADO 5.5 Una muestra de parámetros pasados por valor

```
1: // Listado 5.5 - muestra de parámetros pasados por valor
2:
3: #include <iostream.h>
4:
5: void intercambiar(int x, int y);
6:
7: int main()
8: {
9:     int x = 5, y = 10;
10:
11:    cout << "main(). Antes del intercambio, x: " << x << " y:
12:    << y << "\n";
13:    intercambiar(x,y);
14:    cout << "main(). Después del intercambio, x: " << x << " y:
15:    << y << "\n";
16:    return 0;
17: }
18:
19: void intercambiar (int x, int y)
20: {
21:     int temp;
22:
23:     cout << "Intercambiar(). Antes del intercambio, x: " << x << " y:
24:     << y << "\n";
25:
26:     temp = x;
27:     x = y;
28:     y = temp;
29: }
```

5

### SALIDA

```
main(). Antes del intercambio, x: 5 y: 10
Intercambiar(). Antes del intercambio, x: 5 y: 10
Intercambiar(). Después del intercambio, x: 10 y: 5
main(). Después del intercambio: x: 5 y: 10
```

**ANÁLISIS** Este programa inicializa dos variables en `main()` y luego las pasa a la función `intercambiar()`, la cual parece intercambiárlas. Sin embargo, al examinarlas otra vez en `main()`, ¡permanecen sin cambio!

Las variables se inicializan en la línea 9, y sus valores se despliegan en la línea 11. Se llama a la función `intercambiar()`, y se pasan las variables.

La ejecución del programa cambia a la función `intercambiar()`, en donde se imprimen de nuevo los valores (línea 21). Éstos se encuentran en el mismo orden en el que estaban en `main()`, como era de esperarse. En las líneas 23 a 25 se intercambian los valores, y esta acción se confirma con la impresión de la línea 27. Evidentemente, al estar en la función `intercambiar()`, los valores se intercambian.

La ejecución regresa entonces a la línea 13, de nuevo en `main()`, en donde los valores ya no están intercambiados.

Como puede ver, los valores pasados a la función `intercambiar()` se pasan por valor, lo que significa que se hacen copias de los valores que son locales para `intercambiar()`. Estas variables locales se intercambian en las líneas 23 a 25, pero al regresar a `main()` permanecen sin cambio.

En el día 8, “Apuntadores”, y en el día 10, “Funciones avanzadas”, verá alternativas para pasar variables que permitan que se cambien los valores en `main()`.

## Más acerca de los valores de retorno

Las funciones regresan un valor o regresan `void`. `void` es una señal para el compilador de que no se regresa ningún valor.

Para regresar un valor de una función, escriba la palabra clave `return` seguida del valor que quiere regresar. El valor podría ser en sí una expresión que regrese un valor. Por ejemplo:

```
return 5;
return (x > 5);
return (MiFuncion());
```

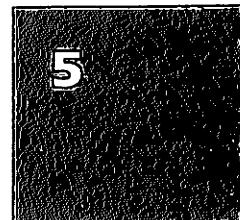
Éstas son instrucciones `return` válidas, asumiendo que la misma función `MiFuncion` regrese un valor. El valor de la segunda instrucción, `return (x > 5)`, será falso (`false`) si `x` no es mayor que 5, o será verdadero (`true`). Lo que se regresa es el valor de la expresión, `false` o `true`, no el valor de `x`.

Al encontrarse la palabra reservada `return`, la expresión que le sigue se regresa como el valor de la función. La ejecución del programa regresa inmediatamente a la función que hizo la llamada, y no se ejecuta ninguna instrucción que esté después de `return`.

Es válido tener más de una instrucción `return` en una sola función. Pero una sola instrucción `return` sólo puede regresar un valor. El listado 5.6 ilustra la idea de tener varias instrucciones `return`.

**ENTRADA****LISTADO 5.6 Una muestra de varias instrucciones return**

```
1: // Listado 5.6 - muestra varias instrucciones
2: // return
3:
4: #include <iostream.h>
5:
6: int Duplicador(int CantidadADuplicar);
7:
8: int main()
9: {
10:
11:     int resultado = 0;
12:     int entrada;
13:
14:     cout << "Escriba un número entre 0 y 10,000 para duplicarlo: ";
15:     cin >> entrada;
16:
17:     cout << "\nAntes de llamar al duplicador... ";
18:     cout << "\nentrada: " << entrada << " duplicada:
19:     << resultado << "\n";
20:
21:     resultado = Duplicador(entrada);
22:
23:     cout << "\nRegresando del duplicador...\n";
24:     cout << "\nentrada: " << entrada << "    duplicada:
25:     << resultado << "\n";
26:
27: }
28:
29: int Duplicador(int original)
30: {
31:     if (original <= 10000)
32:         return original * 2;
33:     else
34:         return -1;
35:     cout << "¡No puede llegar aquí!\n";
36: }
```

**SALIDA**

Escriba un número entre 0 y 10,000 para duplicarlo: 9000

Antes de llamar al duplicador...  
entrada: 9000 duplicada: 0

Regresando del duplicador...

entrada: 9000      duplicada: 18000

Escriba un número entre 0 y 10,000 para duplicarlo: 11000

Antes de llamar al duplicador...  
entrada: 11000 duplicada: 0

Regresando del duplicador...  
entrada: 11000 duplicada: -1

### ANÁLISIS

En las líneas 14 y 15 se pide un número y se imprime en la línea 18, junto con el resultado de la variable local. En la línea 20 se llama a la función `Duplicador()`, y el valor de entrada se pasa como parámetro. El resultado será asignado a la variable local llamada `resultado`, y los valores se imprimirán de nuevo en la línea 23.

En la línea 31, dentro de la función `Duplicador()`, el parámetro se prueba para ver si es mayor que 10,000. Si no lo es, la función regresa el doble del número original. Si es mayor que 10,000, la función regresa -1 como valor de error.

Nunca se llega a la instrucción de la línea 35 porque el valor sea o no mayor que 10,000, la función regresa ya sea en la línea 32 o en la 34, antes de llegar a la línea 35. Un buen compilador advertirá que esta instrucción no se puede compilar, ¡y un buen programador la quitaría!

### Preguntas frecuentes

**FAQ:** ¿Cuál es la diferencia entre `int main()` y `void main();` cuál debo usar? He utilizado ambas y funcionan bien, así que ¿por qué necesito utilizar `int main()` `{return 0;}`?

**Respuesta:** Ambas funcionarán en los compiladores GNU y en la mayoría de los demás, pero sólo `int main()` se apega a las normas ANSI, y por consecuencia sólo `int main()` está garantizada para seguir funcionando en el futuro.

La diferencia es que `int main()` regresa un valor al sistema operativo. Por ejemplo, al terminar su programa, ese valor puede ser capturado por archivos de secuencias de comandos.

Nosotros no utilizamos el valor de retorno de `main` en nuestros ejemplos (usted podría comprobarlo en una de sus secuencias de comandos de shell si lo desea), pero el estándar ANSI lo requiere.

## Parámetros predeterminados

Para cualquier parámetro que se declare en un prototipo y en una definición de función, la función que hace la llamada debe pasar un valor. El valor pasado debe ser del tipo declarado. Por lo tanto, si tiene una función declarada de la siguiente manera:

```
long miFuncion(int);
```

la función debe, en efecto, tomar una variable entera. Si la definición de la función es distinta, o si no se pasa un valor entero, se obtendrá un error de compilación.

La única excepción a esta regla es si el prototipo de la función declara un valor predeterminado para el parámetro. Un valor predeterminado es un valor que se utiliza en caso de no proporcionar uno. La declaración anterior se podría escribir de la siguiente manera:

```
long miFuncion (int x = 50);
```

Este prototipo dice: “`miFuncion()` regresa un valor entero largo y toma un parámetro entero. Si no se proporciona un argumento, utilizar el valor predeterminado `50`”. Debido a que no se requieren los nombres de los parámetros en los prototipos de funciones, esta declaración se hubiera podido escribir de la siguiente manera:

```
long miFuncion (int = 50);
```

La definición de la función no cambia al declarar un parámetro predeterminado. El encabezado de la definición para esta función sería

```
long miFuncion (int x)
```

Si la función que hace la llamada no incluye un argumento, el compilador daría a `x` el valor predeterminado `50`. El nombre del parámetro predeterminado del prototipo no necesita ser el mismo que el del encabezado de la función; el valor predeterminado se asigna por posición, no por nombre.

Los valores predeterminados se pueden asignar a cualquiera o a todos los parámetros de la función. La única restricción es ésta: si alguno de los parámetros no tiene un valor predeterminado, ningún parámetro anterior puede tener un valor predeterminado.

Si el prototipo de la función se ve así:

```
long miFunción (int Param1, int Param2, int Param3);
```

podrá asignar un valor predeterminado a `Param2` sólo si ha asignado un valor predeterminado a `Param3`. Puede asignar un valor predeterminado a `Param1` sólo si asigna valores predeterminados a `Param2` y a `Param3`. El listado 5.7 muestra el uso de valores predeterminados.

5

#### ENTRADA LISTADO 5.7 Una muestra de valores predeterminados de parámetros

```
1: // Listado 5.7 - muestra el uso
2: // de los valores predeterminados de parámetros
3:
4: #include <iostream.h>
5:
6: int VolumenCaja(int longitud, int ancho = 25, int altura = 1);
7:
```

continúa

**LISTADO 5.7** CONTINUACIÓN

```

8: int main()
9: {
10:     int longitud = 100;
11:     int ancho = 50;
12:     int altura = 2;
13:     int volumen;
14:
15:     volumen = VolumenCaja(longitud, ancho, altura);
16:     cout << "La primera vez el volumen es igual a: " << volumen << "\n";
17:
18:     volumen = VolumenCaja(longitud, ancho);
19:     cout << "La segunda vez el volumen es igual a: " << volumen << "\n";
20:
21:     volumen = VolumenCaja(longitud);
22:     cout << "La tercera vez el volumen es igual a: " << volumen << "\n";
23:     return 0;
24: }
25:
26: int VolumenCaja(int longitud, int ancho, int altura)
27: {
28:
29:     return (longitud * ancho * altura);
30: }
```

**SALIDA**

La primera vez el volumen es igual a: 10000  
 La segunda vez el volumen es igual a: 5000  
 La tercera vez el volumen es igual a: 2500

**ANÁLISIS**

En la línea 6, el prototipo de `VolumenCaja()` especifica que esta función toma tres parámetros de tipo entero. Los últimos dos tienen valores predeterminados.

Esta función calcula el volumen de una caja con las dimensiones que se le pasan al momento de ser invocada. Si se escriben dos parámetros, el primero corresponderá a la longitud y el segundo al ancho, mientras que la altura tendrá el valor predeterminado de 1. Si se escribe un sólo parámetro, éste corresponderá a la longitud y se utilizará un ancho de 25 y una altura de 1. No es posible pasar la altura sin pasar el ancho.

En las líneas 10 a 12 se inicializan las dimensiones `longitud`, `altura` y `ancho`, y se pasan a la función `VolumenCaja()` de la línea 15. Se calculan los valores, y el resultado se imprime en la línea 16.

La ejecución regresa a la línea 18, en donde se llama otra vez a `VolumenCaja()`, pero sin dar un valor para la altura. Se utiliza el valor predeterminado, y se vuelven a calcular y a imprimir las dimensiones.

La ejecución regresa a la línea 21, y esta vez no se pasan ni el ancho ni la altura. La ejecución se ramifica por tercera ocasión a la línea 27. Se utilizan los valores predeterminados. Se calcula el volumen y luego se imprime.

DEBE	NO DEBE
DEBE recordar que los parámetros de una función actúan como variables locales dentro de la función.	<p>NO DEBE tratar de crear un valor predeterminado para el primer parámetro si no existe un valor predeterminado para el segundo. Esto también se aplica al segundo parámetro y a los demás. No dé a un parámetro un valor predeterminado si el parámetro que se encuentra a su derecha no tiene uno.</p> <p>NO DEBE olvidar que los argumentos que se pasan por valor no pueden afectar a las variables de la función que hace la llamada.</p> <p>NO DEBE olvidar que los cambios que se hacen a una variable global de una función cambian a esa variable para todas las funciones.</p>

## Sobrecarga de funciones

C++ le permite crear más de una función con el mismo nombre. Esto se conoce como *sobrecarga de funciones*. Las funciones deben ser diferentes en su lista de parámetros, es decir, ésta debe tener un tipo distinto de parámetro, un número distinto de parámetros, o ambos. He aquí un ejemplo:

```
int miFuncion (int, int);
int miFuncion (long, long);
int miFuncion (long);
```

*miFuncion()* está sobrecargada con tres listas de parámetros. La primera y segunda versiones difieren en los tipos de los parámetros, y la tercera difiere en el número de parámetros.

Los tipos de valores de retorno pueden ser iguales o diferentes en funciones sobrecargadas. Debe tener en cuenta que dos funciones que tengan el mismo nombre y la misma lista de parámetros, pero diferentes tipos de valor de retorno, generarán un error de compilación.

La sobrecarga de funciones también se conoce como *polimorfismo de funciones*. *Poli* significa muchos, y *morfismo* significa forma: una función polimorfa tiene muchas formas.

El polimorfismo de funciones se refiere a la capacidad de “sobrecargar” una función con más de un significado. Al cambiar el número o el tipo de los parámetros, puede dar el mismo nombre a dos o más funciones, y se llamará a la función adecuada mediante la comparación de los parámetros utilizados. Esto le permite crear una función que pueda sacar el promedio de enteros, flotantes y otros valores sin tener que crear nombres individuales para cada función, como *PromedioEnteros()*, *PromedioFlotantes()*, etc.

Suponga que escribe una función que duplica cualquier entrada que le dé. Usted quiere pasar un **int**, un **long**, un **float** o un **double**. Sin sobrecarga de funciones, tendría que crear cuatro nombres de función:

```
int DuplicarEntero(int);
long DuplicarLargo(long);
float DuplicarFlotante(float);
double DuplicarDoble(double);
```

Con la sobrecarga de funciones, haría la siguiente declaración:

```
int Duplicar(int);
long Duplicar(long);
float Duplicar(float);
double Duplicar(double);
```

La sobrecarga es más fácil de leer y de utilizar. Usted no tiene que preocuparse por cuál función llamar; sólo necesita pasar una variable, y se llama automáticamente a la función apropiada. El listado 5.8 muestra el uso de la sobrecarga de funciones.

#### ENTRADA

#### **LISTADO 5.8 Una muestra del polimorfismo de funciones**

```
1: // Listado 5.8 - muestra el
2: // polimorfismo de funciones
3:
4: #include <iostream.h>
5:
6: int Duplicar(int);
7: long Duplicar(long);
8: float Duplicar(float);
9: double Duplicar(double);
10:
11: int main()
12: {
13:     int      miEntero = 6500;
14:     long     miLargo = 65000;
15:     float    miFlotante = 6.5F;
16:     double   miDoble = 6.5e20;
17:
18:     int      enteroDuplicado;
19:     long     largoDuplicado;
20:     float    flotanteDuplicado;
21:     double   dobleDuplicado;
22:
23:     cout << "miEntero: " << miEntero << "\n";
24:     cout << "miLargo: " << miLargo << "\n";
25:     cout << "miFlotante: " << miFlotante << "\n";
26:     cout << "miDoble: " << miDoble << "\n";
27:
```

```
28:     enteroDuplicado = Duplicar(miEntero);
29:     largoDuplicado = Duplicar(miLargo);
30:     flotanteDuplicado = Duplicar(miFlotante);
31:     dobleDuplicado = Duplicar(miDoble);
32:
33:     cout << "enteroDuplicado: " << enteroDuplicado << "\n";
34:     cout << "largoDuplicado: " << largoDuplicado << "\n";
35:     cout << "flotanteDuplicado: " << flotanteDuplicado << "\n";
36:     cout << "dobleDuplicado: " << dobleDuplicado << "\n";
37:
38:     return 0;
39: }
40:
41: int Duplicar(int original)
42: {
43:     cout << "En Duplicar(int)\n";
44:     return original * 2;
45: }
46:
47: long Duplicar(long original)
48: {
49:     cout << "En Duplicar(long)\n";
50:     return original * 2;
51: }
52:
53: float Duplicar(float original)
54: {
55:     cout << "En Duplicar(float)\n";
56:     return original * 2;
57: }
58:
59: double Duplicar(double original)
60: {
61:     cout << "En Duplicar(double)\n";
62:     return original * 2;
63: }
```

5

**SALIDA**

```
miEntero: 6500
miLargo: 65000
miFlotante: 6.5
miDoble: 6.5e+20
En Duplicar(int)
En Duplicar(long)
En Duplicar(float)
En Duplicar(double)
enteroDuplicado: 13000
largoDuplicado: 130000
flotanteDuplicado: 13
dobleDuplicado: 1.3e+21
```

**ANÁLISIS**

La función `Duplicar()` se sobrecarga con `int`, `long`, `float` y `double`. Los prototipos están en las líneas 6 a 9, y las definiciones están en las líneas 41 a 63.

En el cuerpo del programa principal se declaran ocho variables locales. En las líneas 13 a 16 se inicializan cuatro de los valores, y en las líneas 28 a 31 se asignan a los otros cuatro los resultados obtenidos al pasar los primeros cuatro a la función `Duplicar()`. Observe que cuando se llama a la función `Duplicar()`, la función que hace la llamada no decide a cuál llamar; sólo pasa un argumento, y se invoca a la función correcta.

El compilador examina los argumentos y elige a cuál de las cuatro funciones `Duplicar()` debe llamar. La salida revela que se llamó a cada una de las cuatro, como era de esperarse.

## Temas especiales sobre funciones

Debido a que las funciones son tan fundamentales para la programación, surgen unos cuantos temas especiales que podrían ser de su interés cuando confronte problemas poco usuales. Si las utiliza sabiamente, las funciones en línea pueden ayudarlo a sacar el máximo rendimiento. La recursión de funciones es una de esas partes maravillosas y enigmáticas de la programación, con la que, de vez en cuando, se puede resolver un problema que de otra manera sería muy difícil de solucionar.

### Funciones en línea

Al definir una función, normalmente el compilador crea sólo un conjunto de instrucciones en la memoria. Al llamar a la función, la ejecución del programa se dirige hacia esas instrucciones, y cuando la función termina, la ejecución regresa a la siguiente línea de la función que hizo la llamada. Si llama 10 veces a la función, el programa salta al mismo conjunto de instrucciones cada vez. Esto significa que sólo existe una copia de la función, no 10.

Hay una disminución en el rendimiento al saltar hacia las funciones y regresar de ellas. Resulta que algunas funciones son muy pequeñas (sólo una o dos líneas de código), y se puede obtener algo de eficiencia si el programa puede evitar hacer estos saltos sólo para ejecutar una o dos instrucciones. Cuando los programadores hablan de eficiencia, por lo general se refieren a la velocidad; el programa se ejecuta más rápido si se puede evitar la llamada a la función.

Si se declara una función con la palabra reservada `inline`, el compilador no creará una función real; copiará el código de esa función directamente en la función que hace la llamada. No se hace ningún salto; es como si usted hubiera escrito las instrucciones de la función dentro de la función que hace la llamada.

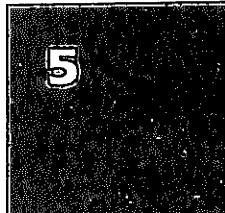
Observe que las funciones en línea pueden tener serias desventajas. Si se llama 10 veces a la función, cada una de esas 10 veces el código en línea se copia en las funciones que hacen las llamadas. El pequeño aumento de velocidad que se puede lograr queda más que sepultado por el aumento de tamaño del programa ejecutable. Por lo tanto, el aumento

de velocidad podría ser ilusorio. En primer lugar, los compiladores optimizadores de la actualidad hacen un magnífico trabajo por sí solos, y declarar una función en línea casi nunca produce un gran aumento de velocidad. Lo que es más importante, el aumento de tamaño produce una disminución en el rendimiento.

¿Cuál es la regla empírica? Si tiene una función pequeña (una o dos instrucciones), es una buena candidata para hacerla una función en línea. No obstante, cuando tenga duda, no la use como función en línea. El listado 5.9 muestra el uso de una función en línea.

**ENTRADA** **LISTADO 5.9** Muestra de una función en línea

```
1: // Listado 5.9 - muestra las funciones en línea
2:
3: #include <iostream.h>
4:
5: inline int Duplicar(int);
6:
7: int main()
8: {
9:     int numero;
10:
11:    cout << "Escriba un número con el que quiera trabajar: ";
12:    cin >> numero;
13:    cout << "\n";
14:
15:    numero = Duplicar(numero);
16:    cout << "Número: " << numero << endl;
17:
18:    numero = Duplicar(numero);
19:    cout << "Número: " << numero << endl;
20:
21:
22:    numero = Duplicar(numero);
23:    cout << "Número: " << numero << endl;
24:    return 0;
25: }
26:
27: int Duplicar(int numero)
28: {
29:     return numero * 2;
30: }
```

**SALIDA**

Escriba un número con el que quiera trabajar: 20

Número: 40  
Número: 80  
Número: 160

**ANÁLISIS**

En la línea 5 se declara `Duplicar()` como función en línea, la cual toma un parámetro de tipo `int` y regresando un `int`. La declaración es como la de cualquier otro prototipo, excepto que se antepone la palabra reservada `inline` antes del valor de retorno.

Al compilarse esto, produce el mismo código que si hubiera escrito lo siguiente:

`numero = numero * 2;`

en cualquier lugar en el que lo hubiera escrito

`numero = Duplicar(numero);`

Al momento en que se ejecuta el programa, las instrucciones ya están en su lugar, compiladas en el archivo ejecutable. Esto ahorra un salto en la ejecución del código, pero a costa de tener un programa más grande.

**Nota**

Inline le indica al compilador que usted quiere que se ponga en línea a la función. El compilador tiene la libertad de ignorar la indicación y de hacer una llamada real a la función.

## Recursión

Una función puede llamarse a sí misma. Esto se conoce como *recursión*, y puede ser directa o indirecta. La recursión es directa cuando una función se llama a sí misma; es indirecta cuando una función llama a otra función, que a su vez llama a la primera función.

Algunos problemas se resuelven con más facilidad mediante la recursión, generalmente aquellos en los que se trabaja sobre los datos, y luego se trabaja de la misma manera sobre el resultado. Ambos tipos de recursión, directa e indirecta, vienen en dos variedades: las que finalmente terminan y producen una respuesta, y las que nunca terminan y producen un error en tiempo de ejecución. Los programadores piensan que estas últimas son algo graciosas (cuando le ocurre a otra persona).

Es importante tener en cuenta que cuando una función se llama a sí misma, se ejecuta una nueva copia de esa función. Las variables locales de la segunda versión son independientes de las variables locales de la primera, y no se pueden afectar unas a otras directamente, así como las variables locales de `main()` no pueden afectar a las variables locales de ninguna función que ésta llame, como se mostró en el listado 5.4.

Para tener un ejemplo de la solución de un problema por medio de la recursión, considere la serie de Fibonacci:

1,1,2,3,5,8,13,21,34...

Cada número después del segundo es la suma de los dos números anteriores. Un problema podría ser determinar cuál es el duodécimo número de la serie.

Una manera de solucionar este problema es examinar cuidadosamente la serie. Los dos primeros números son 1. Cada número subsecuente es la suma de los dos números anteriores. Por lo tanto, el séptimo número es la suma de los números quinto y sexto. Viéndolo en forma más general: el enésimo número es la suma de  $n-2$  y  $n-1$ , siempre y cuando  $n > 2$ .

Las funciones recursivas necesitan una condición para detenerse. Algo debe ocurrir para que el programa detenga la recursión, o ésta nunca terminará. En la serie de Fibonacci,  $n < 3$  es una condición de alto.

El algoritmo a utilizar es el siguiente:

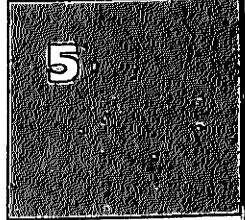
1. Pedir al usuario una posición en la serie.
2. Llamar a la función `fib()` con esa posición, pasando el valor que escribió el usuario.
3. La función `fib()` examina el argumento ( $n$ ). Si  $n < 3$ , regresa 1; de no ser así, `fib()` se llama a sí misma (en forma recursiva) pasando el valor  $n-2$ , se llama a sí misma otra vez pasando  $n-1$ , y regresa la suma.

Si llama a `fib(1)`, ésta regresa 1. Si llama a `fib(2)`, regresa 1. Si llama a `fib(3)`, regresa la suma de llamar a `fib(2)` y `fib(1)`. Como `fib(2)` regresa 1 y `fib(1)` regresa 1, `fib(3)` regresará 2.

Si se llama a `fib(4)`, ésta regresa la suma de llamar a `fib(3)` y a `fib(2)`. Ya establecimos que `fib(3)` regresa 2 (al llamar a `fib(2)` y a `fib(1)`) y que `fib(2)` regresa 1, por lo que `fib(4)` sumará estos números y regresará 3, que es el cuarto número de la serie.

Llevando a cabo un paso más, si se llama a `fib(5)`, ésta regresará la suma de `fib(4)` y `fib(3)`. Ya establecimos que `fib(4)` regresa 3 y que `fib(3)` regresa 2, por lo tanto, la suma regresada será 5.

Este método no es la forma más eficiente para solucionar este problema (¡en `fib(20)` la función `fib()` se llama 13,529 veces!), pero sí funciona. Tenga cuidado; si proporciona un número demasiado grande, se acabará la memoria. Cada vez que se llama a `fib()` se reserva parte de la memoria. Al regresar se libera esa memoria. Con la recursión se sigue reservando la memoria antes de liberarla, y este sistema puede agotar la memoria rápidamente. El listado 5.10 implementa la función `fib()`.



### Precaución

Al ejecutar el listado 5.10, utilice un número pequeño (menor que 15). Debido a que aquí se utilizan la recursión y cout para cada llamada de la función, ésta produce mucha salida y puede consumir mucha memoria.

**ENTRADA****LISTADO 5.10** Muestra de la recursión implementando la serie de Fibonacci

```
1: #include <iostream.h>
2:
3: int fib (int n);
4:
5:     int main()
6:     {
7:
8:         int n, respuesta;
9:         cout << "Escriba el número a encontrar: ";
10:        cin >> n;
11:        cout << "\n\n";
12:
13:        respuesta = fib(n);
14:
15:        cout << respuesta << " es el número " << n << " en la serie de
16:        ↪Fibonacci\n";
17:        return 0;
18:    }
19:    int fib (int n)
20:    {
21:        cout << "Procesando fib(" << n << ")... ";
22:        if (n < 3)
23:        {
24:            cout << "iRegresa 1!\n";
25:            return 1;
26:        }
27:        else
28:        {
29:            cout << "Llama a fib(" << n-2 << ") y a fib(" << n-1 << ").\n";
30:            return (fib(n-2) + fib(n-1));
31:        }
32:    }
```

---

**SALIDA**

Escriba el número a encontrar: 6

```
Procesando fib(6)... Llama a fib(4) y a fib(5).
Procesando fib(4)... Llama a fib(2) y a fib(3).
Procesando fib(2)... iRegresa 1!
Procesando fib(3)... Llama a fib(1) y a fib(2).
Procesando fib(1)... iRegresa 1!
Procesando fib(2)... iRegresa 1!
Procesando fib(5)... Llama a fib(3) y a fib(4).
Procesando fib(3)... Llama a fib(1) y a fib(2).
Procesando fib(1)... iRegresa 1!
Procesando fib(2)... iRegresa 1!
Procesando fib(4)... Llama a fib(2) y a fib(3).
Procesando fib(2)... iRegresa 1!
Procesando fib(3)... Llama a fib(1) y a fib(2).
```

Procesando fib(1)... ¡Regresa 1!  
 Procesando fib(2)... ¡Regresa 1!  
 8 es el número 6 en la serie de Fibonacci

### Nota

Algunos compiladores (que no son de GNU) tienen dificultades al utilizar operadores en una instrucción cout. Si recibe una advertencia en la línea 28, coloque paréntesis alrededor de la operación de resta para que dicha línea se vea así:

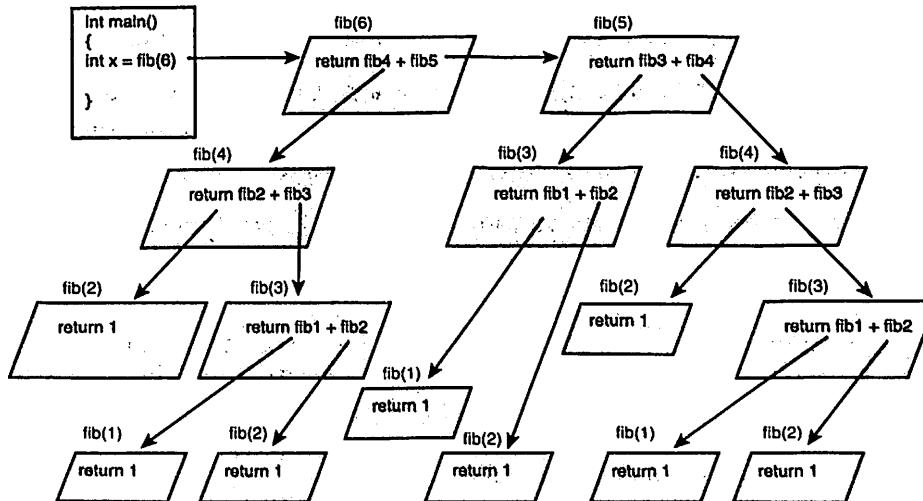
```
28:     cout << "Llama a fib("<< (n-2) << ") y a
      fib(" << (n-1) << ").\n";
```

En la línea 9, el programa pide que se encuentre un número, y asigna ese número a n. Luego llama a fib() con n. En la línea 20, la ejecución se ramifica hacia la función fib(), que imprime su argumento.

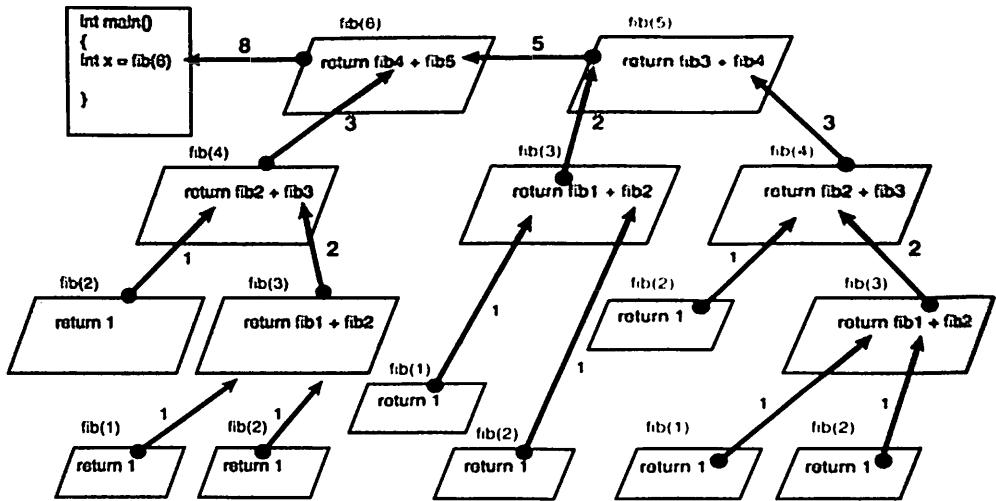
En la línea 21 se prueba el argumento n para ver si es menor que 3; de ser así, fib() regresa el valor 1. De no ser así, regresa la suma de los valores regresados al llamar a fib() con n-2 y n-1.

La función no puede regresar estos valores sino hasta que se resuelva cada llamada (a fib()). Por lo tanto, usted puede imaginar al programa dividiéndose en fib() una y otra vez, hasta llegar a una llamada a fib() que regrese un valor. Las únicas llamadas que regresan un valor inmediatamente son las llamadas a fib(2) y a fib(1). Estos valores de retorno se pasan entonces a las funciones que hicieron las llamadas y que están en espera, las que a su vez suman el valor de retorno al suyo, y luego regresan ese valor. Las figuras 5.4 y 5.5 muestran esta recursión de fib().

**FIGURA 5.4**  
*Uso de la recursión.*



**FIGURA 5.5**  
Retorno de la recursión.



En el ejemplo,  $n$  vale 6, por lo que se llama a `fib(6)` desde `main()`. La ejecución salta hasta la función `fib()`, y en la línea 21 se prueba  $n$  para ver si tiene un valor menor que 3. La prueba falla, por lo que `fib(6)` regresa la suma de los valores que regresan `fib(4)` y `fib(5)`.

29:       `return(fib(n-2) + fib(n-1));`

Esto significa que se hace una llamada a `fib(4)` (debido a que  $n == 6$ , `fib(n-2)` es igual que `fib(4)`) y se hace otra llamada a `fib(5)` (`fib(n-1)`), y luego la función en la que usted está (`fib(6)`) *espera* hasta que estas llamadas regresen un valor. Cuando esto ocurre, esta función puede regresar el resultado de la suma de esos dos valores.

Como `fib(5)` pasa un argumento que no es menor que 3, se llamará a `fib()` de nuevo, esta vez con 4 y 3. A su vez, `fib(4)` llamará a `fib(3)` y a `fib(2)`.

La salida rastrea estas llamadas y los valores de retorno. Compile, enlace y ejecute este programa, escribiendo primero 1, luego 2, luego 3, y así hasta llegar a 6, y observe cuidadosamente la salida.

Esta sería una buena oportunidad para empezar a experimentar con su depurador. Coloque un punto de interrupción en la línea 20 y luego rastree *internamente* cada llamada a `fib()`, llevando un registro del valor de  $n$  al ir rastreando cada llamada recursiva a `fib()`.

La recursión no se utiliza muy seguido en la programación en C++, pero puede ser una herramienta poderosa y elegante para ciertas necesidades.

### Nota

La recursión es una parte engañosa de la programación avanzada. Se presenta aquí porque puede ser útil para comprender los fundamentos de su funcionamiento, pero no se preocupe mucho si no entiende completamente todos los detalles.

## Cómo trabajan las funciones: un vistazo a su interior

Al llamar a una función, el código se ramifica a la función llamada, se pasan los parámetros y se ejecuta el cuerpo de la función. Al terminar de ejecutarse la función, se regresa un valor (a menos que la función regrese void), y el control regresa a la función que hizo la llamada.

¿Cómo se logra esto? ¿Cómo sabe el código hacia donde ramificarse? ¿En dónde se guardan los valores cuando se pasan? ¿Qué ocurre con las variables que se declaran en el cuerpo de la función? ¿Cómo se pasa el valor de retorno? ¿Cómo sabe el código en dónde continuar?

La mayoría de los libros introductorios no trata de contestar estas preguntas, pero sin entender esta información, le parecerá que la programación sigue siendo un misterio. El tema requiere de una breve explicación sobre la memoria de la computadora.

### Niveles de abstracción

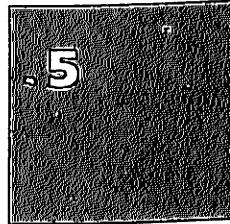
Uno de los principales obstáculos para los nuevos programadores es tratar con los varios niveles de abstracción intelectual. Desde luego que las computadoras son sólo equipos electrónicos. No saben nada acerca de ventanas y menús, ni de programas ni instrucciones, y ni siquiera saben nada sobre unos y ceros. Todo lo que está ocurriendo realmente es que se está midiendo voltaje en varios lugares de un circuito integrado. Incluso esto es una abstracción: la electricidad en sí es sólo un concepto intelectual que representa el comportamiento de partículas subatómicas.

Pocos programadores se preocupan por saber que hay más allá de los valores en la RAM. Después de todo, usted no necesita comprender la física de partículas para conducir un auto, hacer pan tostado o pegarle a una pelota de béisbol, y no necesita comprender la electrónica de una computadora para programarla.

Lo que sí necesita es comprender cómo está organizada la memoria. Sin tener una imagen mental razonablemente sólida de en dónde se encuentran sus variables cuando son creadas y de cómo se pasan los valores entre las funciones, todo seguirá siendo un misterio indescifrable.

### Partición de la RAM

Cuando usted inicia su programa, Linux configura varias áreas de memoria con base en los requerimientos del programa compilado. Como programador de C++, a menudo se preocupará por el espacio de nombres global, el heap, los registros, el espacio de código y la pila.



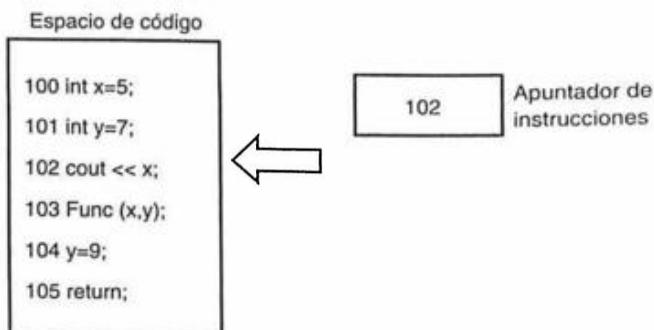
Las variables globales se encuentran en el espacio de nombres global. Hablaremos más acerca del espacio de nombres global y del heap en los días que siguen, pero por ahora nos enfocaremos en los registros, el espacio de código y la pila.

Los registros son un área especial de memoria construida dentro de la CPU (unidad central de procesamiento). Se encargan del mantenimiento interno. Mucho de lo que ocurre en los registros está más allá del alcance de este libro, pero lo que nos interesa es el conjunto de registros responsables de apuntar, en cualquier momento dado, a la siguiente línea de código. Llamamos a estos registros en conjunto el apuntador de instrucciones. El trabajo del apuntador de instrucciones es mantener un registro de cuál línea de código se debe ejecutar a continuación.

El código en sí se encuentra en el *espacio de código*, que es parte de la memoria que se reserva para guardar el archivo binario de las instrucciones que usted creó en su programa (ese archivo binario es su programa ejecutable). Cada línea de código fuente se traduce en una serie de instrucciones en lenguaje de máquina, y cada una de estas instrucciones se encuentra en una dirección específica de memoria. El apuntador de instrucciones tiene la dirección de la siguiente instrucción que se va a ejecutar. La figura 5.6 ilustra esta idea.

**FIGURA 5.6**

*El apuntador de instrucciones.*



La *pila* es un área especial de memoria asignada para que su programa guarde la información requerida por cada una de sus funciones. Se llama pila porque es una lista en la que el último elemento en entrar es el primero que sale, algo muy parecido a un montón de platos apilados en una cafetería, como se muestra en la figura 5.7.

**FIGURA 5.7**

*Una pila.*



En esta pila lo que se agregue al último será lo primero que se pueda sacar. La mayoría de las listas son como la fila de personas esperando a comprar boletos en un cine. El primero de la fila es el primero que sale. Una pila es más parecida a un montón de monedas apiladas. Si apila 10 monedas en una mesa y luego quita algunas, las últimas tres que haya puesto serán las primeras tres que quite.

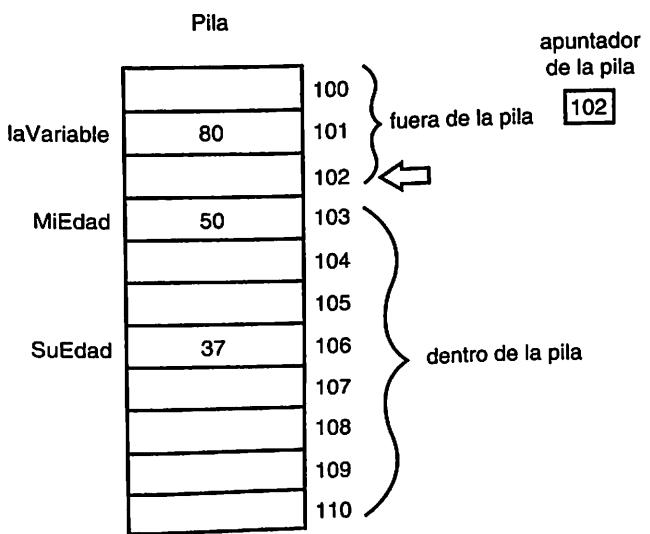
Cuando los datos se *introducen* en la pila, ésta crece; al *sacar* datos de la pila, ésta decrece. No es posible sacar un plato de la pila sin sacar primero todos los platos que estén sobre ese plato.

Una pila de platos es la analogía más común, sólo que está mal en cierta manera fundamental. Una imagen mental más precisa es una serie de casillas alineadas de arriba hacia abajo. La parte superior de la pila es cualquier casilla a la que el apuntador de la pila (que es otro registro) esté apuntando en ese momento.

Cada una de las casillas tiene una dirección secuencial, y una de esas direcciones se mantiene en el registro del apuntador de la pila. Todo lo que haya debajo de esa dirección mágica, conocida como la parte superior de la pila, se considera que está dentro de la pila. Todo lo que esté encima de la parte superior de la pila se considera que está fuera de la pila, y no es válido. La figura 5.8 ilustra esta idea.

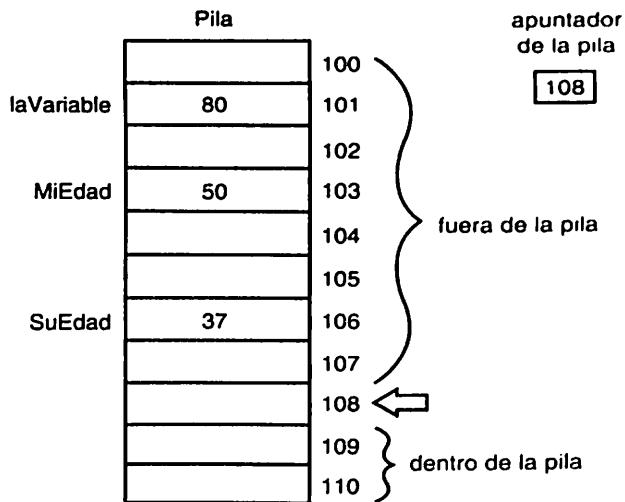
**FIGURA 5.8**

*El apuntador de la pila.*



Cuando se colocan datos en la pila, se colocan en una casilla que esté encima del apuntador de la pila, y luego se mueve el apuntador de la pila hacia los nuevos datos. Cuando se sacan datos de la pila, lo que realmente ocurre es que se cambia la dirección del apuntador de la pila moviéndolo hacia abajo una posición. La figura 5.9 aclara esta regla.

**FIGURA 5.9**  
*Movimiento del apuntador de la pila.*



## La pila y las funciones

Lo siguiente es lo que ocurre cuando un programa, que se ejecute en la mayoría de los sistemas, se ramifica hacia una función:

1. La dirección del apuntador de instrucciones se incrementa a la siguiente instrucción después de la llamada a la función. Esa dirección se coloca a continuación en la pila, y será la dirección de retorno cuando la función regrese.
2. Se hace espacio en la pila para el tipo de valor de retorno que usted declaró. Si en un sistema con enteros de 2 bytes se declara como `int` el tipo de valor de retorno, entonces se agregan otros dos bytes a la pila, pero no se coloca ningún valor en estos bytes.
3. La dirección de la función llamada, que también se encuentra en el área de código, se carga en el apuntador de instrucciones, por lo que la siguiente instrucción que se ejecute estará en la función llamada.
4. La parte superior actual de la pila se anota y se guarda en un apuntador especial llamado borde de la pila. En adelante, todo lo que se agregue a la pila hasta que la función regrese, se considera “local” para la función.
5. Todos los argumentos para la función se colocan en la pila.
6. La instrucción que se encuentra ahora en la dirección del apuntador de instrucciones se ejecuta, lo cual viene siendo la primera instrucción de la función.
7. Las variables locales se introducen en la pila a medida que se definen.

Cuando una función está lista para regresar, ocurre lo siguiente:

1. El valor de retorno se coloca en el área de la pila reservada en el paso 2 de arriba.
2. A continuación, la pila se desplaza hacia abajo hasta llegar al apuntador del borde de la pila, lo que efectivamente descarta todas las variables locales y los argumentos para la función.

3. El valor de retorno se saca de la pila y se asigna como el valor de la llamada de la función en sí.
4. La dirección que se guardó en el paso 1 de la llamada se recupera y se coloca en el apuntador de instrucciones. El programa continúa inmediatamente después de la llamada a la función, con el valor de la función recuperado.

Algunos de los detalles de este proceso cambian de un compilador a otro, o de una computadora a otra, pero las ideas esenciales son consistentes en todos los entornos. En general, cuando se llama a una función, se colocan en la pila la dirección de retorno y los parámetros. Durante la vida de la función, las variables locales se agregan a la pila. Cuando la función regresa, las variables locales se eliminan al vaciar la pila.

En los días siguientes verá otras ubicaciones en memoria que se utilizan para guardar datos que deben continuar más allá de la vida de la función.

## Programas de archivos fuente múltiples (bibliotecas de funciones creadas por el programador)

Uno de los puntos más fuertes de C y de C++ es la habilidad para reutilizar el código. Ahora todos los lenguajes soportan esta habilidad. Sin embargo, para muchos de ellos el mecanismo es cortar y pegar (también conocido como clonación); cuando usted quiere utilizar un código que ya existe, copia el código fuente en el programa nuevo y parte de ahí. El problema con este método es la dificultad en el mantenimiento. Cuando la regla empresarial incluida en el código cambie (y todas lo harán alguna vez), todas las copias del código tienen que cambiar. Pero cada versión es un poco diferente y se tiene que revisar cuidadosamente antes de poder cambiarla.

C y C++ proporcionan un mejor mecanismo para la reutilización del código: bibliotecas de funciones y de objetos. Éstas son colecciones de código ya compilado en formato de archivo objeto (.o o .obj). Con una disciplina apropiada, hay solamente una versión que todos los programadores utilizan. Sólo se cambia esa versión del código fuente, y cualquier programa que se vea afectado se vuelve a compilar o a enlazar.

Además del archivo objeto, usted debe crear un archivo de encabezado que contenga los prototipos para esas funciones.

Linux también soporta un archivo de "biblioteca" en forma de archivo de paquetes. Un archivo de paquetes contiene varios archivos objeto. El enlazador sabe cómo leer archivos en formato objeto simple o cómo extraerlos de un archivo de paquetes. El comando `ar` se utiliza para manipular archivos de paquetes. Revise su manual o el archivo de información para más detalles.

Además, puede crear archivos de proyecto (conocidos como archivos make, en relación con el comando que los utiliza: `make`). Estos archivos de proyecto controlan la compilación y el enlace con base en reglas que usted crea. En la forma más simple, su programa se volverá a compilar y enlazar si cualquiera de sus partes cambia (código fuente principal, archivos de encabezado y bibliotecas).

La siguiente sección le muestra cómo crear y utilizar bibliotecas simples de funciones; la sección que sigue después de ésa presenta el comando `make` de GNU (`gmake`).

## Cómo crear y utilizar bibliotecas de funciones con g++

Una biblioteca de funciones simple consta de tres partes: el código fuente de las funciones (las funciones en sí), los prototipos de las funciones de un archivo de encabezado y las funciones compiladas.

Puede sacar la función del programa de la serie de Fibonacci que se muestra en el listado 5.10. El listado 5.11a contiene el archivo de encabezado y se debe guardar como `lst05-11.h`.

**ENTRADA** **LISTADO 5.11a** Archivo de encabezado (`lst05-11.h`) para mostrar las bibliotecas de funciones

---

```

1: // archivo de encabezado lst05-11.h
2: #ifndef __LST05-11_H
3: #define __LST05-11_H
4: int fib (int n);
5: #endif

```

---

El listado 5.11b contiene la función `fib()`. Ésta se compila en forma separada del programa principal y se enlaza posteriormente.

**ENTRADA** **LISTADO 5.11b** Biblioteca de funciones

---

```

1: #include <iostream.h>
2: #include "lst05-11.h"
3: // archivo fuente de biblioteca lst05-11.cxx
4: int fib (int n)
5: {
6:     cout << "Procesando fib(" << n << "... ";
7:     if (n < 3)
8:     {
9:         cout << "iRegresa 1!\n";
10:        return 1;
11:    }
12:    else
13:    {
14:        cout << "Llama a fib(" << n-2 << ") y a fib(" << n-1 << ").\n";
15:        return(fib(n-2) + fib(n-1));
16:    }
17: }

```

---

El listado 5.12 contiene el programa principal que llama a la función `fib()`. Esto es lo mismo que el programa del listado 5.10, excepto que la función se ha sacado y el prototipo de la función se encuentra en un archivo de encabezado (en lugar de estar en el mismo archivo que `main()`).

**ENTRADA** **LISTADO 5.12** Muestra de la recursión implementando la serie de Fibonacci—Uso de bibliotecas

```
1: #include <iostream.h>
2: #include "lst05-11.h"
3: // Listado 5.12 - Una muestra del uso de bibliotecas
4:
5: int main()
6: {
7:
8:     int n, respuesta;
9:     cout << "Escriba el número a encontrar: ";
10:    cin >> n;
11:    cout << "\n\n";
12:
13:    respuesta = fib(n);
14:
15:    cout << respuesta << " es el número " << n << " en la serie
16:    de Fibonacci\n";
17:    return 0;
18: }
```

Hay dos maneras de compilar las funciones `fib()` y `main()` juntas; la primera es al mismo tiempo:

```
g++ lst05-12.cxx lst05-11.cxx -o lst05-12
```

o puede utilizar dos comandos separados:

```
g++ -c lst05-11.cxx
g++ lst05-12.cxx lst05-11.o -o lst05-12
```

La ventaja del segundo método es que puede compilar `lst05-11.cxx` una vez en un archivo objeto y proporcionar el archivo objeto y el de encabezado a otros programadores. Eso permite a los programadores utilizar su función sin poder cambiarla (porque no tienen el código fuente).

5

## Creación de archivos de proyecto (para make)

En su forma más simple, el comando `make` en Linux le permite preparar un proyecto y dejar que ciertas reglas controlen lo que se compila y cuándo se compila. El código se cambiará sólo cuando sea necesario (cuando algo cambie).

Estas reglas se guardan en un archivo conocido como *make*, (que también es el nombre predeterminado para el archivo), aunque se le puede dar cualquier nombre.

El formato general de un archivo make consiste en dos tipos distintos de líneas. La primera es el destino/dependencia, la cual define el destino a crear junto con cualquier archivo del que dependa. Las otras líneas son los comandos requeridos para producir ese destino. El archivo make para los listados 5.11a, 5.11b y 5.12 se muestra en el listado 5.13.

**ENTRADA** **LISTADO 5.13** Archivo make que se utiliza en los listados 5.11a, 5.11b y 5.12 (lst05-13.mak)

```
1: lst05-11.o: lst05-11.cxx lst05-11.h
2:         g++ -c lst05-11.cxx
3: #
4: lst05-12: lst05-12.cxx lst05-11.o lst05-11.h
5:         g++ lst05-12.cxx lst05-11.o -o lst05-12
```

La primera línea muestra que *lst05-11.o* (la biblioteca de funciones, nuestro destino) depende de *lst05-11.cxx* y de *lst05-11.h*. La línea 2 es el comando para producir ese destino. La línea 3 es un comentario.

La línea 4 muestra que *lst05-12* (el archivo ejecutable, nuestro destino de este paso) depende de *lst05-12.cxx*, *lst05-11.o* y de *lst05-11.h*. La línea 5 es el comando para producir ese destino.

**Nota**

Las líneas 2 y 5 tienen un carácter de tabulación antes del comando *g++*. Éste debe ser un carácter de tabulación. Si utiliza cualquier otro carácter aquí (incluyendo espacios), obtendrá el siguiente mensaje de error del comando *make*:

suarchivo.mak:2: \*\*\* missing separator. Stop.

Cuando *lst05-11.h* cambie y usted trate de hacer el archivo ejecutable usando el siguiente comando, creará todas las partes necesarias:

*make -f lst05-13.mak lst05-12*

**SALIDA**

```
g++ -c lst05-11.cxx
gcc: -lgpp: linker input file unused since linking not done
gcc: -lstdcx: linker input file unused since linking not done
gcc: -lm: linker input file unused since linking not done
g++ lst05-12.cxx lst05-11.o -o lst05-12
```

**ANÁLISIS**

El destino especificado es *lst05-12*. Éste necesita a *lst05-11.o*, y a *lst05-11.h*. *lst05-11.o* necesita a *lst05-11.h*. Yo cambié a *lst05-11.h* (agregué un espacio adicional al final de una línea) antes de emitir el comando *make*. Este comando es lo suficientemente inteligente como para determinar si hay otras dependencias o

no. Determinó que como `lst05-12` necesita a `lst05-11.o`, y como `lst05-11.h` cambió desde la última vez que se creó `lst05-11.o`, era necesario volver a compilar a `lst05-11.cxx` antes que a `lst05-12.cxx`.

El comando `make` es mucho más sofisticado de lo que muestra este sencillo ejemplo. Dé un vistazo a las páginas del manual para obtener mayor información; `make` soporta un lenguaje de programación propio.

Usted puede crear archivos de proyecto sencillos como éste para sus programas. Así no necesita recordar cuáles son las dependencias.

## Funciones de bibliotecas estándar de C++ (`libg++`)

Todos los compiladores suscritos al estándar ANSI vienen con una serie de funciones para que usted las utilice. Se dice que estas funciones son *integradas* ya que vienen con el compilador; no las escribe usted. El compilador GNU no es la excepción. Existe una biblioteca estándar de funciones para programas de C y un superconjunto de esa biblioteca de funciones y objetos para los programadores de C++.

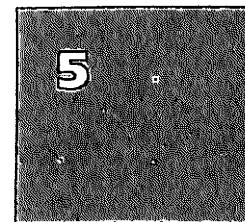
Aprenderá acerca de muchos de los objetos estándar más adelante. Siempre puede utilizar el comando `info` para obtener mayor información:

```
info libg++
```

Puede obtener detalles acerca de las funciones estándar de C mediante el comando `info`:

```
info libc
```

Las funciones estándar para C y C++ son muy importantes. Puede utilizarlas en lugar de escribir sus propias funciones. Esto facilita la creación de los programas (menos cosas por hacer) y simplifica enormemente su mantenimiento debido a que los fabricantes de compiladores (Colaboradores del Proyecto GNU/Fundación para el Software Libre) se preocupan por mantener esas funciones. Éstas se clasifican en las siguientes secciones: matemáticas, de caracteres y funciones generales.



### Funciones matemáticas

La biblioteca estándar de funciones matemáticas le permite realizar operaciones matemáticas comunes (y algunas no tan comunes). Esto hace que usted se preocupe por la forma de utilizar los resultados en lugar de preocuparse por cómo calcular los resultados. Por ejemplo, hay una función que calcula la raíz cuadrada de un número positivo. Como programador, usted no necesita saber cómo codificar esto (otro programador ya lo ha hecho). Sólo necesita saber qué hacer con los resultados (o por qué quiere obtener la raíz cuadrada de un número).

Algunas de las funciones matemáticas que se utilizan con más frecuencia se muestran en la tabla 5.1.

**TABLA 5.1** Funciones matemáticas comunes

Función	Uso
<code>ceil(f)</code>	Redondea el número $f$ al entero más pequeño que no sea menor que $f$ : <code>ceil(3.3)</code> es 4 y <code>ceil(-3.3)</code> es -3.
<code>cos(f)</code>	Calcula el coseno de $f$ (donde $f$ se da en radianes): <code>cos(0.0)</code> es 1 y <code>cos(3.1415926)</code> es -1.
<code>exp(f)</code>	Función exponencial en la forma de $e^f$ : <code>exp(0.0)</code> es 1 y <code>exp(1.0)</code> es 2.718(e).
<code>fabs(f)</code>	Valor absoluto en punto flotante de $f$ : <code>fabs(-1.0)</code> es 1, <code>fabs(0.0)</code> es 0 y <code>fabs(-1.0)</code> es 1.
<code>floor(f)</code>	Redondea $f$ al entero más grande que no sea mayor que $f$ : <code>floor(3.3)</code> es 3 y <code>floor(-3.3)</code> es -4; compárese con la función <code>ceil()</code> .
<code>fmod (f, g)</code>	Calcula el residuo de $f/g$ como <code>float</code> .
<code>log (f)</code>	Logaritmo natural de $f$ usando como base e: <code>log(1.0)</code> es 0.0 y <code>log(2.718)</code> es 1.0.
<code>log10 (f)</code>	Logaritmo en base 10 de $f$ : <code>log10(0.0)</code> no está definido, <code>log10(1.0)</code> es 0.0, <code>log10(10.0)</code> es 1 y <code>log10(1000.0)</code> es 3.
<code>pow (f, g)</code>	Eleva $f$ a la $g$ -ésima potencia ( $f^g$ ): <code>pow(2.0, 3.0)</code> es 8.0 y <code>pow(3.0, 2.0)</code> es 9.0.
<code>sin (f)</code>	Calcula el seno de $f$ (donde $f$ se da en radianes): <code>sin(0.0)</code> es 0 y <code>sin(3.1415926 / 2.0)</code> es 1.
<code>sqrt (f)</code>	Raíz cuadrada de $f$ : <code>sqrt(4.0)</code> es 2 y <code>sqrt(9.0)</code> es 3.
<code>tan (f)</code>	Calcula la tangente de $f$ (donde $f$ se da en radianes): <code>tan(0.0)</code> es 0 y <code>tan(3.1415926 / 4.0)</code> es 1.
<code>acos, asin, atan</code>	Arco coseno, arco seno y arco tangente (funciones inversas del coseno, seno y tangente).
<code>acosh, asinh, atanh</code>	Arco coseno hiperbólico, arco seno hiperbólico y arco tangente hiperbólica (funciones inversas del coseno, seno y tangente hiperbólicos).
<code>cosh, sinh, tanh</code>	Coseno hiperbólico, seno hiperbólico y tangente hiperbólica.

Para todas estas funciones se debe incluir el archivo de encabezado `math.h` (para obtener los prototipos funcionales).

## Funciones de caracteres y de cadenas de caracteres.

La manipulación de caracteres y cadenas es una tarea común en la mayoría de los programas. Como resultado, se ha construido una gran biblioteca de funciones a través de los años, la cual se ha convertido en parte del estándar.

Al igual que las funciones matemáticas, ¡estas funciones existen para facilitarle la vida!

En la tabla 5.2 se muestran algunas de las funciones de cadenas y de caracteres que se utilizan con más frecuencia.

**TABLA 5.2** Funciones comunes de cadenas y de caracteres

Función	Uso
<code>strcpy (s1, s2)</code>	Copia s2 en s1 carácter por carácter hasta copiar el carácter nulo de terminación.
<code>strncpy (s1, s2, n)</code>	Copia s2 en s1 carácter por carácter hasta copiar el carácter nulo de terminación, o hasta que se copian n caracteres; s1 no termina con carácter nulo si se terminó la copia por llegar a la cuenta de n caracteres.
<code>strcat (s1, s2)</code>	Agrega s2 al final de s1. (Concatena s1 y s2.)
<code>strncat (s1, s2, n)</code>	Agrega, como máximo, n caracteres de s2 al final de s1.
<code>strcmp (s1, s2)</code>	Compara s1 con s2 carácter por carácter y regresa la diferencia. Si las cadenas son iguales, se regresa 0; de no ser así, un valor positivo o uno negativo muestra la diferencia entre las dos.
<code>strncmp (s1, s2, n)</code>	Compara, como máximo, n caracteres de s1 con s2 carácter por carácter y regresa la diferencia. Si las cadenas son iguales, se regresa 0; de no ser así, un valor positivo o uno negativo muestra la diferencia entre ambas.
<code>strlen(s1)</code>	Regresa la longitud (en caracteres, sin contar el carácter nulo de terminación) de s1.
<code>strlwr (s1)</code>	Convierte a s1 en minúsculas.
<code>strtok (s1, s2)</code>	Divide s1 en tokens con base en los valores contenidos en s2. Esto divide y analiza sintácticamente a s1 en piezas delimitadas por los caracteres contenidos en s2. La primera vez que se llama a strtok, se proporciona s2; cada vez subsecuente, utilice NULL en lugar de s2.
<code>strtod (s1), strtol (s1)</code>	Convierte a s1 en un valor numérico entero o entero largo.
<code>strupr (s1)</code>	Convierte a s1 en mayúsculas.

Debe incluir el archivo de encabezado `string.h` para todas estas funciones (para obtener los prototipos funcionales).

## Funciones generales

Existen muchas otras funciones que son parte de la biblioteca estándar. Algunas de las funciones más comunes que no son matemáticas ni de caracteres se muestran en la tabla 5.3.

**TABLA 5.3** Otras funciones utilizadas con frecuencia

Función	Uso
<code>system (s1)</code>	Envía a <code>s1</code> como comando al sistema operativo o al intérprete de comandos.
<code>malloc (n)</code>	Asigna <code>n</code> bytes de memoria y regresa un apuntador a la dirección. Regresa <code>NULL</code> en caso de fallar.
<code>free (ptr)</code>	Libera la memoria asignada por <code>malloc</code> o <code>realloc</code> .
<code>realloc (ptr, n)</code>	Asigna <code>n</code> bytes que aumentan o disminuyen el espacio asignado previamente (por <code>malloc</code> o <code>realloc</code> ). Permite la implementación de arreglos "dinámicos".
<code>abs (n)</code>	Regresa el valor absoluto entero de <code>n</code> : <code>abs (-1)</code> es 1, <code>abs (0)</code> es 0 y <code>abs (1)</code> es 1.
<code>asctime (t)</code>	Convierte el tiempo de formato interno <code>t</code> en un formato imprimible.
<code>bsearch ()</code>	Realiza una búsqueda binaria en una estructura de datos ordenada.
<code>qsort ()</code>	Ordena una estructura de datos con el método rápido (quick sort).
<code>isalpha (c)</code>	Regresa <code>true</code> si el carácter <code>c</code> es alfabético.
<code>isdigit (c)</code>	Regresa <code>true</code> si el carácter <code>c</code> es un dígito numérico.
<code>islower (c)</code>	Regresa <code>true</code> si el carácter <code>c</code> es alfabético y está en minúscula.
<code>isupper (c)</code>	Regresa <code>true</code> si el carácter <code>c</code> es alfabético y está en mayúscula.

Existen muchas, pero muchas otras funciones disponibles en la biblioteca estándar. Revise las páginas del manual o el archivo `info` para obtener mayor información acerca de estas y otras funciones. Algunas de las funciones son muy complicadas (como `bsearch()` y `qsort()`) y requieren de mucha más explicación, la cual se puede hallar en la documentación.

## Mucho más

Recuerde, la reutilización de código es uno de los aspectos básicos de C y de C++. Es también una de las bases de la ingeniería de software. Cualquier código que pueda volver a utilizar es código que no tiene que escribir como si fuera nuevo, ni le tiene que dar mantenimiento. Puede ser mucho más resistente y se puede probar con más intensidad que si fuera escrito por programadores individuales. Y si se requieren cambios, se pueden hacer en un solo lugar (en la biblioteca), en lugar de tener que rastrear todo el código.

Revise las bibliotecas estándar (de C y C++) antes de escribir sus propias funciones (tal vez ya exista una que pueda resolver su problema). Tal vez necesite varias funciones de la biblioteca estándar para realizar la tarea que requiere, pero esto le ahorrará tiempo y esfuerzo.

Recuerdo un proyecto durante el cual salí de la ciudad para asistir a una conferencia. Había un consultor joven en el proyecto. Era un buen codificador y entendía bien el lenguaje C, pero no estaba muy familiarizado con la biblioteca. Necesitaba ordenar cierta información

dentro de su programa (no era mucha, por lo que el algoritmo de ordenamiento no era importante). Debido a que muy pocas personas memorizan la manera de escribir rutinas para ordenar datos, él tuvo que investigar a fondo en sus apuntes de la universidad, codificar, probar y ajustar el código. Esto le tomó varias horas durante un par de días. Yo regresé de la conferencia, y al enterarme de lo que estaba batallando con la codificación, le mostré la página de la función `qsort()` de la biblioteca estándar.

En una o dos horas en el mismo día, usando una función estándar (y una función especial para determinar el ordenamiento de los datos), él habría podido ordenar sus datos, pero le tomó más tiempo, tuvo que trabajar más duro, y creó más código que mantener. Utilicé el género masculino en esta descripción porque la persona con la que trabajé era hombre. Se puede cometer el mismo error sin importar cuál sea el género.

## Resumen

En este capítulo se presentaron las funciones. Una función es, en efecto, un subprograma al que usted puede pasar parámetros y del que puede regresar un valor. Todo programa de C++ empieza en la función `main()`, la que a su vez puede llamar a otras funciones.

Una función se declara con un prototipo de función, el cual describe el valor de retorno, el nombre de la función y los tipos de sus parámetros. Como una opción, puede declarar una función en línea. Un prototipo de función también puede declarar variables con valores predeterminados para uno o más de sus parámetros.

La definición de la función debe concordar con el prototipo de la función en el tipo de valor de retorno, nombre y la lista de parámetros. Los nombres de funciones se pueden sobrecargar cambiando el número o el tipo de los parámetros; el compilador encuentra la función correcta basándose en la lista de argumentos.

Las variables locales de las funciones, junto con los argumentos que se pasan a la función, son locales para el bloque en el que se declaran. Los parámetros que se pasan por valor son copias y no pueden afectar el valor de las variables de la función que hace la llamada.

5

## Preguntas y respuestas

**P** ¿Por qué no hacer todas las variables globales?

**R** Durante un tiempo, ésta era exactamente la manera en que se programaba. Sin embargo, a medida que los programas se fueron haciendo más complejos, se hizo muy difícil encontrar errores en los programas porque cualquiera de las funciones podía alterar los datos (los datos globales se pueden cambiar en cualquier parte del programa). Años de experiencia han convencido a los programadores de que los datos se deben mantener tan locales como sea posible, y el acceso para cambiar esos datos se debe definir estrechamente.

- P ¿Cuándo se debe utilizar la palabra reservada `inline` en un prototipo de función?**
- R Si la función es muy pequeña, no mayor de una o dos líneas, y no se llamará desde muchos lugares en su programa, se puede declarar como función en línea.**
- P ¿Por qué los cambios al valor de los argumentos de la función no se reflejan en la función que hace la llamada?**
- R Los argumentos que se pasan a una función se pasan por valor. Esto significa que el argumento de la función es en realidad una copia de la variable original. Este concepto se explica a profundidad en la sección “Cómo trabajan las funciones: un vistazo a su interior”.**
- P Si se pasan los argumentos por valor, ¿qué hago si necesito reflejar los cambios en la función que hace la llamada?**
- R En el día 8 hablaremos sobre los apuntadores. El uso de apuntadores solucionará este problema, además de proporcionar una forma para resolver la limitación de regresar un solo valor de una función.**
- P ¿Qué pasa si tengo las siguientes dos funciones?:**
- ```
int Area (int ancho, int longitud = 1); int Area (int tamanio);
```
- ¿Se sobrecargarían? Existe un número distinto de parámetros, pero el primero tiene un valor predeterminado.**
- R Las declaraciones sí compilarán, pero si invoca a la función `Area` con un parámetro, recibirá un error en tiempo de compilación: ambigüedad entre `Area(int, int)` y `Area(int)`.**

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Cuáles son las diferencias entre el prototipo de la función y la definición de la función?
2. ¿Tienen que concordar los nombres de los parámetros en el prototipo, la definición y la llamada a la función?
3. ¿Cómo se declara una función si no regresa un valor?
4. Si no declara un valor de retorno, ¿qué tipo de valor de retorno se asume?
5. ¿Qué es una variable local?

6. ¿Qué es el alcance?
7. ¿Qué es la recursión?
8. ¿Cuándo debe utilizar variables globales?
9. ¿Qué es la sobrecarga de funciones?
10. ¿Qué es el polimorfismo?

## Ejercicios

1. Escriba el prototipo para una función llamada `Perimetro()`, la cual regresa un valor de tipo `unsigned long int` y acepta dos parámetros, ambos de tipo `unsigned short int`.
2. Escriba la definición de la función `Perimetro()` como se describe en el ejercicio 1. Los dos parámetros representan la longitud y el ancho de un rectángulo. Haga que la función regrese el perímetro (dos veces la longitud más dos veces el ancho).
3. **CAZA ERRORES:** ¿Qué está mal en la función del siguiente código?

```
#include <iostream.h>
void miFunc(unsigned short int x);
int main()
{
    unsigned short int x, y;
    y = miFunc(int);
    cout << "x: " << x << " y: " << y << "\n";
}

void miFunc(unsigned short int x);
{
    return (4*x);
}
```

5

4. **CAZA ERRORES:** ¿Qué está mal en la función del siguiente código?

```
#include <iostream.h>;
int miFunc(unsigned short int x);
int main()
{
    unsigned short int x, y;
    y = miFunc(x);
    cout << "x: " << x << " y: " << y << "\n";
}

int miFunc(unsigned short int x);
{
    return (4*x);
}
```

5. Escriba una función que acepte dos argumentos de tipo `unsigned short int` y que regrese el resultado de la división del primero entre el segundo. No haga la división si el segundo número es igual a cero, pero regrese el valor de -1.

6. Escriba un programa que pida dos números al usuario y que llame a la función que escribió en el ejercicio 5. Imprima la respuesta o imprima un mensaje de error si obtiene -1.
7. Escriba un programa que pida un número y una potencia. Escriba una función recursiva que eleve el número a esa potencia. Por ejemplo, si el número es 2 y la potencia es 4, la función debe regresar 16.

# SEMANA 1

## DÍA 6

### Clases base

Las clases extienden las capacidades integradas de C++ para ayudarlo a representar y resolver problemas complejos del mundo real. Hoy aprenderá lo siguiente:

- Qué son las clases y los objetos
- Cómo definir una nueva clase y crear objetos de esa clase
- Qué son las funciones miembro y los datos miembro
- Qué son los constructores y cómo utilizarlos

### Creación de nuevos tipos

Ya ha visto varios tipos de variables, incluyendo de enteros sin signo y de caracteres. El tipo de una variable le da mucha información acerca de ella. Por ejemplo, si declara `Altura` y `Ancho` como tipo entero corto sin signo (`unsigned short int`), sabe que cada una puede guardar un número entre 0 y 65,535, si el entero corto sin signo es de 2 bytes. Esto es lo que se quiere dar a entender al decir que son enteros sin signo; tratar de guardar cualquier otra cosa en estas variables produce un error. No puede guardar su nombre en un entero corto sin signo, y ni siquiera debería tratar de hacerlo.

Con sólo declarar estas variables como enteros cortos sin signo, usted sabe que es posible sumar `Altura` y `Ancho` y asignar ese número a otra variable.

El tipo de estas variables le indica lo siguiente:

- Su tamaño en memoria
- Qué información pueden guardar
- Qué acciones se pueden realizar sobre ellas

Visto en forma más general, un tipo es una categoría. Entre los tipos conocidos del mundo real se encuentran auto, casa, persona, fruta y forma. En C++, el programador puede crear cualquier tipo que necesite, y cada uno de estos nuevos tipos puede tener toda la funcionalidad y poder de los tipos integrados.

### ¿Por qué crear un nuevo tipo?

Por lo general, los programas se escriben para resolver problemas del mundo real, como mantener el registro de los empleados o simular el funcionamiento de un sistema de calefacción. Aunque puede resolver problemas complejos utilizando programas escritos sólo con enteros y caracteres, le será mucho más fácil combatir problemas grandes y complejos si crea representaciones de los objetos de los que está hablando.

En otras palabras, simular el funcionamiento de un sistema de calefacción es más fácil si crea variables que representen cuartos, sensores térmicos, termostatos y calentadores. Entre más cercanas estén estas variables a la realidad, será más sencillo escribir el programa.

## Introducción a las clases y miembros

Usted crea un nuevo tipo al declarar una clase. Una clase es sólo una colección de variables, por lo general de tipos distintos, combinadas con un conjunto de funciones relacionadas.

Una forma de pensar en un auto es como si fuera una colección de ruedas, puertas, asientos, ventanas etc. Otra forma es pensar en lo que un auto puede hacer: se puede mover, acelerar, desacelerar, detener, estacionar, etc. Una clase le permite conjuntar, o reunir en un solo paquete, todas estas partes y funciones en una sola entidad, lo que se conoce como objeto.

Una ventaja de reunir un conjunto de características y funciones en una entidad, es la forma en que se puede definir su interacción. En el ejemplo del auto existe una interacción entre el carburador, el múltiple, las válvulas y los pistones. Sin embargo, usted no necesita interactuar con ellos mientras maneja; simplemente oprime el acelerador y el motor hará el resto. Debido a que el funcionamiento interno del motor tiene poca importancia para casi todos los automovilistas, el motor se oculta bajo el cofre y uno se olvida de los detalles técnicos. Este concepto corresponde a la encapsulación de datos y funciones en una clase. Encapsular en una clase todo lo que sepa acerca de un auto tiene varias ventajas para un programador. Todo está en un solo lugar, lo que facilita la referencia, el copiado y la manipulación de los datos. Asimismo, los clientes de la clase (es decir, las partes del programa que utilizan esa clase) pueden utilizar el objeto sin preocuparse por lo que hay dentro de él o por la forma en que funciona.

Una clase se puede componer de cualquier combinación de los tipos de variables y también de otros tipos de clases. Las variables que están en la clase se conocen como *variables miembro* o *datos miembro*. Una clase llamada Auto podría tener variables miembro que representen los asientos, el tipo de radio, las llantas, y así sucesivamente.

Las variables miembro, también conocidas como datos miembro, son las variables de la clase. Las variables miembro son parte de la clase, así como las ruedas y el motor son parte del auto.

Por lo general, las funciones de la clase manipulan a las variables miembro. Estas funciones se conocen como *funciones miembro* o *métodos* de la clase. Entre los métodos de la clase Auto se podrían incluir Arrancar() y Frenar(). Una clase llamada Gato podría tener datos miembro que representen la edad y el peso; entre sus métodos se podrían incluir Dormir(), Maullar() y PerseguirRatones().

Las funciones miembro, también conocidas como métodos, son las funciones de la clase. Las funciones miembro son parte de una clase de la misma manera que las variables miembro. Ellas determinan lo que la clase puede hacer.

Algunos autores llaman funciones miembro a todas las funciones incluidas en una clase y reservan el término *método* para las funciones miembro públicas.

## Declaración de una clase

Para declarar una clase, utilice la palabra reservada class seguida de una llave de apertura, y luego ponga en una lista los datos miembro y métodos de esa clase. Termine la declaración con una llave de cierre y un punto y coma. La declaración de una clase llamada Gato sería así:

```
class Gato
{
    unsigned int suEdad;
    unsigned int suPeso;
    void Maullar();
};
```

Al declarar esta clase no se asigna memoria para un Gato. Sólo se le indica al compilador cómo es un Gato, qué datos contiene (suEdad y suPeso), y qué puede hacer (Maullar()). También se le indica al compilador qué tan grande es un Gato (es decir, cuánto espacio debe reservar el compilador para cada Gato que se vaya a crear). En este ejemplo, si un entero es de 4 bytes, el tamaño de un Gato sería de 8 bytes: suEdad es de 4 bytes y suPeso ocupa otros 4 bytes. Maullar() no ocupa espacio porque no se reserva espacio de almacenamiento para las funciones miembro (métodos).

## Unas palabras sobre las convenciones de denominación

Como programador, usted debe nombrar todas sus variables miembro, funciones miembro y clases. Como aprendió en el día 3, “Variables y constantes”, los nombres deben ser fáciles de entender y significativos. Gato, Rectangulo y Empleado son buenos nombres de clases. Maullar(), PerseguirRatones() y DetenerMotor() son buenos nombres de funciones porque indican lo que éstas hacen. Muchos programadores denominan sus variables miembro con el prefijo su, como en suEdad y suVelocidad. Esto ayuda a distinguir las variables miembro de las que no son variables miembro.

C++ es sensible al uso de mayúsculas, y todos los nombres de las clases deben seguir el mismo patrón. Establezca un patrón de denominación para clases, variables y funciones, de esta forma, nunca tendrá que revisar la manera de deletrear el nombre de una clase: ¿era Rectangulo, rectangulo o RECTANGULO? A algunos programadores les gusta colocar un prefijo en cada nombre de clase, formado por una letra específica (por ejemplo, cGato o cPersona), mientras que otros utilizan sólo mayúsculas o sólo minúsculas para el nombre. La convención utilizada en este libro es denominar todas las clases con la primer letra en mayúscula, como en Gato y Persona.

De la misma manera, muchos programadores empiezan todas las funciones con mayúscula y todas las variables con minúscula. Las palabras por lo general se separan con un guión bajo, como en Perseguir\_Raton, o poniendo en mayúscula la primer letra de cada palabra, por ejemplo, PerseguirRaton o DibujarCirculo.

Como mencionamos hace unos días al hablar sobre las variables, lo importante es que usted escoja un estilo y lo mantenga en todos sus programas. Con el tiempo, su estilo evolucionará para incluir no sólo convenciones de denominación, sino también sangrías, alineación de las llaves y estilo de comentarios.

### Nota

Es muy común que las compañías de desarrollo tengan estándares internos para muchas cuestiones relacionadas con los estilos. Esto asegura que los desarrolladores puedan leer fácilmente el código de otros desarrolladores.

Aun cuando no hay estándares formales, los grupos desarrollan sus métodos preferidos.

## Declaración de un objeto

Usted declara un objeto de su nuevo tipo de la misma forma en que declara una variable de tipo entero:

```
unsigned int PesoNeto;           // declaración de un entero sin signo  
Gato Pelusa;                   // declaración de un Gato
```

Este código declara una variable llamada PesoNeto, cuyo tipo es entero sin signo. También declara a Pelusa, que es un objeto cuya clase (o tipo) es Gato.

## Comparación de clases y objetos

Usted no tiene como mascota la definición de un gato; tiene como mascota uno o más gatos individuales. Usted establece la diferencia entre la idea de un gato y el gato que se encuentra justo ahora paseando por toda la sala. De la misma manera, C++ establece la diferencia entre la clase Gato, que es la idea de un gato, y cada objeto individual del tipo Gato. Por lo tanto, Pelusa es un objeto de tipo Gato, así como PesoNeto es una variable de tipo unsigned int.

Un objeto es una instancia individual de una clase. El término técnico para crear un objeto es *instanciación*.

**Nota**

Algunas veces la terminología se reutiliza y se utiliza en forma errónea. El significado de términos individuales, o qué término se debe utilizar y en dónde, a menudo son tema de discusión y convención. Como resultado, hoy una palabra es correcta, pero mañana puede ser incorrecta.

En este libro, la palabra *declarar* se utiliza para indicar que se está especificando o creando un objeto o variable. En las clases pasa algo similar, al describir los miembros (datos y funciones) en una clase de objetos, se está declarando esa clase.

Al trabajar con variables normales (como int o float), no existe diferencia entre el proceso de describir y crear. Con las clases de objetos sí hay una diferencia.

Usted describe (define y declara) la clase de objeto en una ubicación del programa (por lo regular, en un archivo .hpp) y posteriormente crea un objeto específico.

Con las clases de objetos, cuando asigna la memoria, lo que hace en realidad es crear un objeto específico. Algunos autores utilizan la palabra *especificar* para este propósito.

## Cómo acceder a los miembros de las clases

Después de definir un objeto Gato (por ejemplo, Pelusa) se utiliza el operador de punto (.) para tener acceso a los miembros de ese objeto. Por lo tanto, para asignar 50 a la variable miembro suPeso del objeto Pelusa, se escribiría lo siguiente:

```
Pelusa.suPeso = 50;
```

De la misma forma, para llamar a la función Maullar(), se escribiría lo siguiente:

```
Pelusa.Maullar();
```

Cuando utiliza el método de una clase, llama al método. En este ejemplo, usted llama a Maullar(), la cual se encuentra en Pelusa.

## Asignar a objetos, no a clases

En C++ usted no asigna valores a los tipos; asigna valores a las variables. Por ejemplo, nunca escribiría

```
int = 5; // incorrecto
```

El compilador marcaría esto como un error porque no puede asignar 5 a un entero. En lugar de eso, debe definir una variable de tipo entero y asignar 5 a esa variable. Por ejemplo,

```
int x; // declarar x para que sea int
x = 5; // asignar a x el valor 5
```

Ésta es una manera corta de decir "Asignar 5 a la variable x, que es de tipo int". De la misma manera, usted no escribiría

```
Gato.suEdad=5;           // incorrecto
```

El compilador marcaría esto como error porque no puede asignar 5 a la parte de la edad de un Gato. En lugar de eso, debe definir un objeto Gato y asignar 5 a ese objeto, por ejemplo:

```
Gato Pelusa;           // igual que int x;
Pelusa.suEdad = 5;     // igual que x = 5;
```

## **Si no lo declara, su clase no lo tendrá**

Pruebe este experimento: camine hacia un niño de tres años y muéstrelle un gato. Luego diga: "Éste es Pelusa. Pelusa sabe un truco. Pelusa, ladra". El niño reirá y dirá: "No, tonto, los gatos no pueden ladrar".

Si escribiera

```
Gato Pelusa;           // hacer un Gato llamado Pelusa
Pelusa.Ladrar()        // decir a Pelusa que ladre
```

el compilador diría: "No, tonto, los gatos no pueden ladrar". En realidad, el compilador GNU le dirá:

```
ejemplo.cxx: In function 'int main()':
ejemplo.cxx:25: no member function 'Gato::Ladrar()' defined
```

El compilador sabe que Pelusa no puede ladrar porque la clase Gato no tiene una función Ladrar(). El compilador ni siquiera dejaría que Pelusa maullara si usted no definiera una función Maullar().

### **DEBE**

**DEBE** utilizar la palabra reservada class para declarar una clase.

**DEBE** utilizar el operador de punto (.) para tener acceso a los miembros y a las funciones de la clase.

### **No BEBE**

**NO DEBE** confundir una declaración con una definición. Una declaración de clase dice lo que es la clase. Una definición es la implementación de los métodos y funciones que se encuentran en la declaración de clase. Una declaración de objeto reserva memoria para un objeto.

**NO DEBE** confundir una clase con un objeto.

**NO DEBE** asignar valores a una clase. Asigne valores a los datos miembro de un objeto.

## Definición del alcance público en comparación con la del privado

En la declaración de una clase se pueden utilizar otras palabras reservadas. Dos de las más importantes son: `public` y `private`.

Todos los miembros de una clase (datos y métodos) son privados de manera predeterminada. Puede acceder a los miembros privados sólo dentro de métodos que se encuentren en la clase misma. Puede acceder a los miembros públicos a través de cualquier objeto que esté dentro del programa. Esta distinción es tan importante como confusa. Para hacerla un poco más clara, considere un ejemplo presentado anteriormente en este capítulo:

```
class Gato
{
    unsigned int    suEdad;
    unsigned int    suPeso;
    void Maullar();
};
```

En esta declaración, `suEdad`, `suPeso` y `Maullar()` son privados debido a que todos los miembros de una clase son privados de manera predeterminada. Esto significa que, a menos que se especifique de otra forma, son privados.

No obstante, si escribe lo siguiente en la función `main()` (por ejemplo):

```
Gato Botas;
Botas.suEdad=5;           // ¡error! ¡No puede acceder a datos privados!
```

el compilador marca esto como error. En efecto, usted dijo al compilador: "Voy a acceder a `suEdad`, `suPeso` y `Maullar()` sólo dentro de funciones miembro de la clase `Gato`". Y aquí accedió a la variable miembro `suEdad` del objeto `Botas` desde afuera de un método de la clase `Gato`. El hecho de que `Botas` sea un objeto de la clase `Gato` no significa que usted pueda acceder a las partes de `Botas` que sean privadas.

Esto produce mucha confusión en los programadores novatos de C++. Casi puedo escucharlo gritar: "¡Hey! Acabo de decir que `Botas` es un `Gato`. ¿Por qué `Botas` no puede acceder a su propia edad?" La respuesta es que `Botas` sí puede, pero usted no. `Botas`, dentro de sus propios métodos, puede acceder a todas sus partes, públicas y privadas. Que usted haya creado una clase `Gato` no significa que pueda ver o cambiar las partes de ella que sean privadas.

La manera de utilizar Gato para que pueda acceder a los datos miembro es la siguiente:

```
class Gato
{
public:
    unsigned int suEdad;
    unsigned int suPeso;
    void Maullar();
};
```

Ahora suEdad, suPeso y Maullar() son públicos. Botas.suEdad=5 se compila sin problemas.

El listado 6.1 muestra la declaración de una clase Gato con variables miembro públicas.

**ENTRADA** **LISTADO 6.1** Cómo acceder a los miembros públicos de una clase simple

```
1: // Muestra de la declaración de una clase y
2: // de la declaración de un objeto de esa clase,
3:
4: include <iostream.h> // para cout
5:
6: class Gato           // declarar el objeto clase
7: {
8:     public:          // los siguientes miembros son públicos
9:         int suEdad;
10:        int suPeso;
11:    };
12:
13:
14: int main()
15: {
16:     Gato Pelusa;
17:     Pelusa.suEdad = 5; // asignar a la variable miembro
18:     cout << "Pelusa es un gato que tiene ";
19:     cout << Pelusa.suEdad << " años de edad.\n";
20:     return 0;
21: }
```

**SALIDA** Pelusa es un gato que tiene 5 años de edad.

**ANÁLISIS** La línea 6 contiene la palabra reservada **class**. Esto le indica al compilador que lo que sigue es una declaración. El nombre de la nueva clase viene después de la palabra reservada **class**. En este caso, es **Gato**.

El cuerpo de la declaración empieza en la línea 7 con la llave de apertura, y termina en la línea 11 con una llave de cierre y un punto y coma. La línea 8 contiene la palabra reservada **public**, que indica que todo lo que sigue a continuación es público hasta llegar a la palabra reservada **private** o al final de la declaración de la clase.

Las líneas 9 y 10 contienen las declaraciones de los miembros de la clase, es decir, de suEdad y suPeso.

La función principal del programa empieza en la línea 14. Pelusa se declara en la línea 16 como una instancia de Gato (es decir, un objeto Gato). En la línea 17, la edad de Pelusa se establece en 5. En las líneas 18 y 19 se utiliza la variable miembro suEdad para imprimir un mensaje acerca de Pelusa.

### Nota

Trate de convertir la línea 8 en comentario e intente volver a compilar. Recibirá un mensaje de error en las líneas 17 y 19 debido a que suEdad ya no tendrá acceso público. El acceso predeterminado para las clases es el privado. El compilador GNU le indicará esto de la siguiente manera:

```
1st06-01.cxx: In function 'int main()':  
1st06-01.cxx:17: member 'suEdad' is a private member of class 'Gato'  
1st06-01.cxx:19: member 'suEdad' is a private member of class 'Gato'
```

## Debe hacer que los datos miembro sean privados

Como regla de diseño general, debe mantener privados los datos miembro de una clase. Por lo tanto, debe crear funciones públicas, conocidas como métodos de acceso, para establecer y obtener los valores de las variables miembro privadas. Estos *métodos de acceso* son las funciones miembro llamadas por otras partes del programa para establecer y obtener valores de las variables miembro privadas.

Un *método de acceso público* es una función miembro de la clase que se utiliza para leer el valor de una variable miembro privada de la clase, o para darle un valor.

¿Por qué batallar con este nivel adicional de acceso indirecto? Después de todo, es más simple y sencillo utilizar los datos en lugar de trabajar mediante funciones de acceso.

Las funciones de acceso le permiten separar los detalles tanto de la forma en que se guardan los datos como de la forma en que se utilizan. Esto le permite cambiar la forma en que se guardan los datos sin tener que volver a escribir funciones que utilicen esos datos.

Si una función que necesita saber la edad de un Gato accede directamente a la variable miembro suEdad, sería necesario volver a escribir esa función si usted, como autor de la clase Gato, decidiera cambiar la forma en que se guarda ese dato. Al hacer que la función llame a ObtenerEdad(), su clase Gato puede fácilmente regresar el valor correcto sin importar cómo llegue a la edad. La función que hace la llamada no necesita saber si usted la está guardando como entero sin signo o como entero largo, o si la está calculando a medida que la necesita.

Esta técnica facilita el mantenimiento de los programas. Le proporciona una vida más larga a su código debido a que los cambios en el diseño no hacen que su programa sea obsoleto. Es una buena táctica de ingeniería de software y de programación defensiva debido a que esconde los detalles del elemento que hace la llamada.

El listado 6.2 muestra la clase **Gato** modificada para incluir datos miembro privados y métodos de acceso públicos. Observe que éste no es un listado ejecutable.

**ENTRADA****LISTADO 6.2 Una clase con métodos de acceso**

```

1:      // Declaración de la clase Gato
2:      // Los datos miembro son privados, los métodos de acceso públicos
3:      // se encargan de asignar y obtener los valores de los datos privados
4:
5: class Gato
6: {
7: public:
8:     // elementos de acceso públicos
9:     unsigned int ObtenerEdad();
10:    void AsignarEdad(unsigned int Edad);
11:
12:    unsigned int ObtenerPeso();
13:    void AsignarPeso(unsigned int Peso);
14:
15:    // funciones miembro públicas
16:    void Maullar();
17:
18:    // datos miembro privados
19: private:
20:    unsigned int suEdad;
21:    unsigned int suPeso;
22:
23: };

```

**ANÁLISIS**

Esta clase tiene cinco métodos públicos. Las líneas 9 y 10 contienen los métodos de acceso para `suEdad`. En las líneas 12 y 13 están los métodos de acceso para `suPeso`. Estas funciones de acceso asignan valores a las variables miembro y regresan sus valores.

La función miembro pública `Maullar()` se declara en la línea 16. `Maullar()` no es una función de acceso. No asigna ni obtiene el valor de una variable miembro; realiza otro servicio para la clase: imprimir la palabra `Miau`.

Las variables miembro se declaran en las líneas 20 y 21.

Para establecer la edad de Pelusa, se pasaría el valor al método `AsignarEdad()`, como en el siguiente ejemplo:

```

Gato Pelusa;
Pelusa.AsignarEdad(5); // asignar la edad de Pelusa mediante el método de
                      // acceso público

```

## Distinción entre privacidad y seguridad

La declaración de métodos o datos privados permite al compilador encontrar equivocaciones en la programación antes de que se conviertan en errores. Cualquier programador que valga lo que cobra puede encontrar una manera de evadir la privacidad, si lo desea. Stroustrup, el inventor de C++, dijo: "Los mecanismos de control de acceso de C++ proporcionan protección contra los accidentes, no contra los fraude". (ARM, 1990.)

### La palabra reservada class

La sintaxis para la palabra reservada class es la siguiente:

```
class nombre_clase
{
    // aquí van las palabras reservadas de control de acceso
    // aquí se declaran las variables y los métodos de la clase
};
```

La palabra reservada class se utiliza para declarar nuevos tipos. Una clase es una colección de datos miembro que son variables de diversos tipos, incluyendo a otras clases. La clase también contiene funciones (o métodos) que se utilizan para manipular los datos de la clase y para realizar otros servicios para la clase.

Usted crea objetos del nuevo tipo en forma muy parecida a la que declara cualquier variable. Declara el tipo (class) y luego el nombre de la variable (el objeto). Accede a los miembros y a las funciones de la clase mediante el operador de punto (.).

Utiliza palabras reservadas de control de acceso para declarar secciones de la clase como públicas o privadas. El control de acceso predeterminado es privado. Cada palabra reservada cambia el control de acceso desde ese punto hasta el final de la clase o hasta la siguiente palabra reservada de control de acceso. Las declaraciones de clases terminan con una llave de cierre y un punto y coma.

#### Ejemplo 1

```
class Gato;
{
    public:
        unsigned int suEdad;
        unsigned int suPeso;
        void Maullar();
};

Gato Pelusa;
Pelusa.suEdad = 8;
Pelusa.suPeso = 18;
Pelusa.Maullar();
```

**Ejemplo 2**

```

class Auto
{
public:                                // los siguientes cinco son
   // públicos

    void Encender();
    void Acelerar();
    void Frenar();
    void AsignarAnio(int anio);
    int ObtenerAnio();

private:                                 // el resto es privado

    int Anio;
    char Modelo [255];
};

Auto ViejoFiel;                         // hacer una instancia de un auto
int comprado;                           // una variable local de tipo int
ViejoFiel.AsignarAnio(84);              // asignar 84 al año
comprado = ViejoFiel.ObtenerAnio();     // asignar 84 a comprado
ViejoFiel.Encender();                   // llamar al método para encender
   // el auto

```

**DEBE**

- DEBE** declarar las variables miembro como privadas.
- DEBE** utilizar métodos de acceso públicos.
- DEBE** acceder a las variables miembro privadas desde el interior de las funciones miembro de la clase.

**NO DEBE**

- NO DEBE** tratar de utilizar variables miembro privadas desde afuera de la clase.

## Implementación de los métodos de una clase

Como ha visto, un método de acceso proporciona una interfaz pública para los datos miembro privados de la clase. Cada método de acceso, así como cualquier otro método de la clase que declare, debe tener una implementación. A esto se le conoce como *definición de métodos*.

La definición de un método empieza con el nombre de la clase, seguido por dos signos de dos puntos (::), el nombre del método y sus parámetros. El listado 6.3 muestra la declaración completa de una clase Gato simple y la implementación de sus métodos de acceso y un método general de la clase.

**ENTRADA****LISTADO 6.3** Implementación de los métodos de una clase simple

```

1: // Muestra de la declaración de una clase y
2: // la definición de los métodos de la clase,
3:
4: #include <iostream.h>           // para cout
5:
6: class Gato                     // empieza la declaración de la clase
7: {
8:     public:                   // empieza la sección pública
9:         int ObtenerEdad();    // método de acceso
10:        void AsignarEdad (int edad); // método de acceso
11:        void Maullar();       // método general
12:    private:                  // empieza la sección privada
13:        int suEdad;          // variable miembro
14:    };
15:
16: // ObtenerEdad, método de acceso público
17: // regresa el valor de la propiedad suEdad
18: int Gato::ObtenerEdad()
19: {
20:     return suEdad;
21: }
22:
23: // definición de AsignarEdad, método
24: // de acceso público
25: // da un valor a la propiedad suEdad
26: void Gato::AsignarEdad(int edad)
27: {
28:     // dar a la variable miembro suEdad el
29:     // valor pasado por el parámetro edad
30:     suEdad = edad;
31: }
32:
33: // definición del método Maullar
34: // regresa: void
35: // parámetros: Ninguno
36: // acción: Imprime "miau" en la pantalla
37: void Gato::Maullar()
38: {
39:     cout << "Miau.\n";
40: }
41:
42: // crear un gato, asignar un valor a su edad, hacer que
43: // maúlle, que nos diga su edad y que vuelva a maullar.
44: int main()
45: {

```

**LISTADO 6.3** CONTINUACIÓN

---

```

46:     Gato Pelusa;
47:     Pelusa.AsignarEdad(5);
48:     Pelusa.Maullar();
49:     cout << "Pelusa es un gato que tiene ";
50:     cout << Pelusa.ObtenerEdad() << " años de edad.\n";
51:     Pelusa.Maullar();
52:     return 0;
53: }
```

---

**SALIDA**

Miau.  
Pelusa es un gato que tiene 5 años de edad.  
Miau.

**ANÁLISIS**

Las líneas 6 a 14 contienen la declaración de la clase Gato. La línea 8 contiene la palabra reservada `public`, que le indica al compilador que lo que sigue a continuación es un conjunto de miembros públicos. La línea 9 tiene la declaración del método de acceso público llamado `ObtenerEdad()`. Este método proporciona acceso a la variable miembro privada `suEdad`, la cual se declara en la línea 13. La línea 10 contiene el método de acceso público llamado `AsignarEdad()`. Este método toma un entero como argumento y asigna a `suEdad` el valor de ese argumento.

La línea 11 tiene la declaración del método `Maullar()` de la clase. Este método no es una función de acceso. Es un método general que imprime la palabra `Miau` en la pantalla.

En la línea 12 comienza la sección privada, que incluye sólo la declaración en la línea 13 de la variable miembro privada `suEdad`. La declaración de la clase termina en la línea 14 con una llave de cierre y un punto y coma.

Las líneas 18 a 21 contienen la definición del método `ObtenerEdad()`. Este método no toma parámetros; regresa un entero. Observe que los métodos de clases incluyen el nombre de la clase seguido por dos signos de dos puntos (::) y el nombre del método (línea 18). Esta sintaxis le indica al compilador que el método `ObtenerEdad()` que está definiendo es el que declaró en la clase `Gato`. Con excepción de esta línea de encabezado, el método `ObtenerEdad()` se crea de la misma manera que cualquier otro método.

El método `ObtenerEdad()` ocupa sólo una línea; regresa el valor de `suEdad`. Observe que la función `main()` no puede tener acceso a `suEdad` debido a que ésta es privada para la clase `Gato`. La función `main()` tiene acceso al método público `ObtenerEdad()`. Como este método es una función miembro de la clase `Gato`, tiene acceso completo a la variable `suEdad`. Este acceso permite que `ObtenerEdad()` regrese el valor de `suEdad` a `main()`.

La línea 26 contiene la definición del método de acceso `AsignarEdad()`. En la línea 30, este método toma un parámetro entero y asigna a la variable `suEdad` el valor de ese parámetro. Como es un miembro de la clase `Gato`, `AsignarEdad()` tiene acceso directo a la variable miembro `suEdad`.

En la línea 37 empieza la definición, o implementación, del método `Maullar()` de la clase `Gato`. Este método es una función de una sola línea que imprime la palabra `Miau` en la pantalla, seguida de una nueva línea. Recuerde que el carácter `\n` imprime una nueva línea en la pantalla.

En la línea 44 empieza el cuerpo del programa con la conocida función `main()`. En este caso, no toma argumentos. En la línea 46, `main()` declara un objeto del tipo `Gato` llamado `Pelusa`. En la línea 47 se asigna el valor 5 a la variable miembro `suEdad` mediante el método de acceso `AsignarEdad()`. Observe que se llama al método mediante el uso del nombre del objeto (`Pelusa`) seguido del operador miembro `(.)` y del nombre del método `(AsignarEdad())`. Cualquier otro método de una clase se puede llamar de la misma manera.

En la línea 48 llama a la función miembro `Maullar()`, y en las líneas 49 y 50 se imprime un mensaje usando el método de acceso `ObtenerEdad()`. En la línea 51 se vuelve a llamar a `Maullar()`.

## Comprensión de los constructores y destructores

Existen dos maneras de declarar una variable de tipo entero. Puede declarar la variable y asignarle un valor posteriormente en el programa, por ejemplo:

```
int Peso;           // declarar una variable
...                 // aquí puede ir más código
Peso = 7;          // asignarle un valor
```

O puede declarar el entero e inicializarlo inmediatamente. Por ejemplo:

```
int Peso = 7;      // declarar e inicializar en 7
```

La inicialización combina la declaración de la variable con su asignación inicial. Nada le impide cambiar ese valor más adelante. La inicialización asegura que su variable nunca tenga un valor sin significado.

Las clases tienen una función miembro especial llamada *constructor*, el cual inicializa los datos miembro de esa clase. El constructor puede tomar los parámetros que necesite, pero no puede tener un valor de retorno, ni siquiera `void`. El constructor es un método de clase que tiene el mismo nombre que ésta.

Siempre que declare un constructor, también debe declarar un destructor. Así como los constructores crean e inicializan objetos de la clase, los destructores se encargan de limpiar todo cuando usted ya no va a utilizar el objeto y liberan la memoria que haya asignado. Un destructor siempre tiene el nombre de la clase, antecedido por una tilde (~). Los destructores no toman argumentos y no tienen valor de retorno. Por lo tanto, la declaración de `Gato` incluye lo siguiente:

```
-Gato();
```

**Nota**

Puede tener múltiples constructores por medio de la sobrecarga de funciones. El compilador selecciona la función constructora que va a utilizar por medio de los parámetros que usted proporcione en la llamada o en la creación. Pero debe tener mucho cuidado al combinar múltiples constructores y argumentos predeterminados; el compilador necesita tener la capacidad de elegir el constructor que usted quiere.

## Constructores y destructores predeterminados

Si no declara un constructor o un destructor, el compilador crea uno por usted. Los constructores y destructores predeterminados no llevan argumentos y no hacen nada.

### Preguntas frecuentes

**FAQ:** ¿Se llama constructor predeterminado porque no tiene argumentos, o porque lo proporciona el compilador si yo no declaro uno?

**Respuesta:** Un constructor que no toma argumentos se llama constructor predeterminado, ya sea que el compilador lo cree por usted, o que lo cree usted mismo. Usted recibe un constructor predeterminado de forma predeterminada.

Aunque suene desconcertante, el destructor predeterminado es el destructor proporcionado por el compilador. Como ningún destructor toma parámetros, lo que distingue al destructor predeterminado es que no realiza ninguna acción (tiene un cuerpo de función vacío).

## Uso del constructor predeterminado

¿De qué sirve un constructor que no hace nada? En parte, es cuestión de forma. Todos los objetos deben ser construidos y destruidos, y estas funciones que no hacen nada se llaman en el momento adecuado. Sin embargo, para declarar un objeto sin pasarle parámetros, como:

Gato Silvestre; // Silvestre no tiene parámetros

debe tener un constructor de la siguiente manera:

Gato();

Cuando se define un objeto de una clase, se llama al constructor. Si el constructor Gato() tuviera dos parámetros, usted podría definir a un objeto Gato escribiendo lo siguiente:

Gato Pelusa (5, 7);

Si el constructor tuviera un parámetro, escribiría

Gato Pelusa (3);

En caso de que el constructor no tuviera parámetros (es decir, que fuera un constructor *predeterminado*), omitiría los paréntesis y escribiría

Gato Pelusa;

Ésta es una excepción a la regla que declara que todas las funciones requieren paréntesis, aunque no lleven parámetros. Ésta es la razón por la cual usted puede escribir

Gato Pelusa;

Esto se interpreta como una llamada al constructor predeterminado. No proporciona parámetros, y omite los paréntesis.

Observe que no tiene que usar el constructor predeterminado proporcionado por el compilador. Siempre es libre de escribir su propio constructor predeterminado (es decir, un constructor sin parámetros). Usted puede dar a su constructor predeterminado un cuerpo de función en el que podría inicializar la clase.

Tenga en cuenta que, por cuestión de forma, si declara un constructor, debe declarar un destructor, aunque su destructor no haga nada. Aunque es cierto que el destructor predeterminado funcionaría correctamente, nunca está de más declarar su propio constructor. Ayuda a que su código sea más claro.

El listado 6.4 vuelve a escribir la clase Gato para utilizar un constructor que inicialice el objeto Gato, asignando a su edad el valor que usted proporcione, y muestra dónde se llama al destructor.

#### ENTRADA LISTADO 6.4 Uso de constructores y destructores

```
1: // Muestra de la declaración de un constructor y
2: // un destructor para la clase Gato
3:
4: #include <iostream.h>           // para cout
5:
6: class Gato                     // empieza la declaración de la clase
7: {
8:     public:                   // empieza la sección pública
9:         Gato(int edadInicial); // constructor
10:        ~Gato();              // destructor
11:        int ObtenerEdad();    // método de acceso
12:        void AsignarEdad(int edad); // método de acceso
13:        void Maullar();
14:    private:                  // empieza la sección privada
15:        int suEdad;            // variable miembro
16:    };
17:
18: // constructor de Gato,
```



**LISTADO 6.4** CONTINUACIÓN

---

```
19:     Gato::Gato(int edadInicial)
20:     {
21:         suEdad = edadInicial;
22:     }
23:
24:     Gato::~Gato()           // destructor, no realiza ninguna acción
25:     {
26:     }
27:
28:     // ObtenerEdad, método de acceso público
29:     // regresa el valor de su miembro suEdad
30:     int Gato::ObtenerEdad()
31:     {
32:         return suEdad;
33:     }
34:
35:     // Definición de AsignarEdad, método
36:     // de acceso público
37:
38:     void Gato::AsignarEdad(int edad)
39:     {
40:         // asignar a la variable miembro suEdad el
41:         // valor pasado por el parámetro edad
42:         suEdad = edad;
43:     }
44:
45:     // definición del método Maullar
46:     // regresa: void
47:     // parámetros: Ninguno
48:     // acción: Imprime "miau" en la pantalla
49:     void Gato::Maullar()
50:     {
51:         cout << "Miau.\n";
52:     }
53:
54:     // crear un gato, asignar un valor a su edad, hacer que
55:     // maúlle, que nos diga su edad, y que vuelva a maullar.
56:     int main()
57:     {
58:         Gato Pelusa(5);
59:         Pelusa.Maullar();
60:         cout << "Pelusa es un gato que tiene " ;
61:         cout << Pelusa.ObtenerEdad() << " años de edad.\n";
62:         Pelusa.Maullar();
63:         Pelusa.AsignarEdad(7);
64:         cout << "Ahora Pelusa tiene " ;
65:         cout << Pelusa.ObtenerEdad() << " años de edad.\n";
66:         return 0;
67:     }
```

---

**SALIDA**

Miau.  
 Pelusa es un gato que tiene 5 años de edad.  
 Miau.  
 Ahora Pelusa tiene 7 años de edad.

**ANÁLISIS**

El listado 6.4 es similar al listado 6.3, excepto que la línea 9 agrega un constructor que toma un entero como parámetro. En la línea 10 se declara el destructor, el cual no toma parámetros. Los destructores nunca llevan parámetros, y ni los constructores ni los destructores regresan valores, ni siquiera void.

Las líneas 19 a 22 muestran la implementación del constructor. Ésta es similar a la implementación del método de acceso `AsignarEdad()`. No hay valor de retorno.

Las líneas 24 a 26 muestran la implementación del destructor `~Gato`. Esta función no hace nada, pero usted debe incluir la definición de la función si lo especifica en la declaración de la clase.

La línea 58 contiene la declaración de un objeto de la clase `Gato`, llamado `Pelusa`. El valor 5 se pasa al constructor de `Pelusa`. No se necesita llamar a `AsignarEdad()` porque `Pelusa` se creó con el valor 5 en su variable miembro `suEdad`, como se muestra en la línea 61. En la línea 63 se asigna 7 a la variable `suEdad` de `Pelusa`. La línea 65 imprime el nuevo valor.

| <b>DEBE</b>                                                      | <b>No DEBE</b>                                                                                                                  |
|------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <b>DEBE</b> utilizar constructores para inicializar sus objetos. | <b>NO DEBE</b> dar a los constructores o destructores un valor de retorno.<br><b>NO DEBE</b> dar parámetros a los destructores. |

## Uso de funciones miembro const

Si declara un método de clase como `const`, está prometiendo que el método no cambiará el valor de ninguno de los miembros de la clase. Para declarar un método de una clase como constante, coloque la palabra reservada `const` después de los paréntesis y antes del punto y coma. La declaración de la función miembro constante llamada `UnaFuncion()` no toma argumentos y regresa `void`. Se ve así:

```
void UnaFuncion() const;
```

Los métodos de acceso a menudo se declaran como funciones constantes mediante el uso del modificador `const`. La clase `Gato` tiene dos métodos de acceso:

```
void AsignarEdad(int edad);
int ObtenerEdad();
```

`AsignarEdad()` no puede ser `const` porque cambia el valor de la variable miembro `suEdad`. Por otro lado, `ObtenerEdad()` puede y debe ser `const` porque no cambia la clase para nada.

`ObtenerEdad()` simplemente regresa el valor actual de la variable miembro `suEdad`. Por lo tanto, la declaración de estos métodos se debe escribir de la siguiente manera:

```
void AsignarEdad(int edad);
int ObtenerEdad() const;
```

Si declara una función como `const`, y la implementación de esa función cambia el objeto al cambiar el valor de cualquiera de sus miembros, el compilador marcará un error. Por ejemplo, si escribiera `ObtenerEdad()` de tal forma que llevara la cuenta del número de veces que se pregunta la edad a `Gato`, generaría un error de compilación. Esto se debe a que estaría cambiando al objeto `Gato` por medio de este método.



### Nota

Utilice `const` siempre que sea posible. Declare funciones miembro como `const` cuando no deban cambiar el objeto. Esto permite que el compilador le ayude a encontrar errores; es más rápido y menos costoso que hacerlo usted mismo.

## Distinción entre interfaz e implementación

Como vio anteriormente, los clientes son la parte del programa que crean y utilizan objetos de una clase. Imagine que la interfaz pública para su clase (la declaración de la clase) es como un contrato con esos clientes. El contrato le indica la forma en que la clase debe comportarse.

Por ejemplo, en la declaración de la clase `Gato` usted crea un contrato que establece que la edad de cualquier clase `Gato` se puede inicializar en su constructor, se le puede asignar un valor mediante el método de acceso `AsignarEdad()`, y se puede leer por medio del método de acceso `ObtenerEdad()`. También promete que cada clase `Gato` sabrá `Maullar()`. Observe que no está diciendo nada en la interfaz pública acerca de la variable miembro `suEdad`; ése es un detalle de implementación que no es parte del contrato. Usted proporcionará una edad (`ObtenerEdad()`) y asignará un valor a esa edad (`AsignarEdad()`), pero el mecanismo (`suEdad`) es invisible.

Si hace de `ObtenerEdad()` una función `const` (como debe hacerlo), el contrato también promete que este método no cambiará la clase `Gato` en la que lo llame.

C++ está fuertemente tipificado, lo que significa que el compilador se asegurará de que cumpla estos contratos, emitiendo un error de compilación cuando los viole. El listado 6.5 muestra un programa que no compilará debido a las violaciones incurridas en estos contratos.

**Precaución**

¡El listado 6.5 no compila!

**ENTRADA****LISTADO 6.5** Una muestra de las violaciones a la interfaz

```
1: // Este listado **NO** compilará
2: // Muestra los errores de compilación
3:
4: #include <iostream.h>           // para cout
5:
6: class Gato
7: {
8:     public:
9:         Gato(int edadInicial);
10:        ~Gato();
11:        int ObtenerEdad() const;    // función const de acceso
12:        void AsignarEdad (int edad);
13:        void Maullar();
14:    private:
15:        int suEdad;
16:    };
17:
18: // constructor de Gato,
19: Gato::Gato(int edadInicial)
20: {
21:     suEdad = edadInicial;
22:     cout << "Constructor de Gato\n";
23: }
24:
25: Gato::~Gato()                  // destructor, no realiza ninguna acción
26: {
27:     cout << "Destructor de Gato\n";
28: }
29: // ObtenerEdad, función const
30: // ipero estamos violando el uso de const!
31: int Gato::ObtenerEdad() const
32: {
33:     return (suEdad++);          // viola el uso de const!
34: }
35:
36: // definición de AsignarEdad, método
37: // de acceso público
38:
39: void Gato::AsignarEdad(int edad)
40: {
41:     // asignar a la variable miembro suEdad el
```

**LISTADO 6.5** CONTINUACIÓN

```

42:      // valor pasado por el parámetro edad
43:      suEdad = edad;
44:  }
45:
46:  // definición del método Maullar
47:  // regresa: void
48:  // parámetros: Ninguno
49:  // acción: Imprime "miau" en la pantalla
50: void Gato::Miau()
51: {
52:     cout << "Miau.\n";
53: }
54:
55: // muestra diversas violaciones a la
56: // interfaz, así como los errores de compilación resultantes
57: int main()
58: {
59:     Gato Pelusa;           // no concuerda con la declaración
60:     Pelusa.Maullar();
61:     Pelusa.Ladrar();      // No, tonto, los gatos no pueden ladrar.
62:     Pelusa.suEdad = 7;    // suEdad es privada
63:     return 0;
64: }
```

**ANÁLISIS** Así como está escrito, este programa no compilará. Por lo tanto, no hay salida.

La línea 11 declara `ObtenerEdad()` como función `const` de acceso (como debe ser). Sin embargo, en la línea 33, en el cuerpo de `ObtenerEdad()`, se incrementa la variable miembro `suEdad`. Como este método está declarado como `const`, no debe cambiar el valor de `suEdad`. Por lo tanto, se marca como error al compilar el programa.

En la línea 13, `Maullar()` no se declara como `const`. Aunque éste no es un error, es una mala práctica de programación. En un mejor diseño se tomaría en consideración que este método no cambia las variables miembro de `Gato`. Por lo tanto, `Maullar()` debería ser `const`.

La línea 59 muestra la declaración de un objeto de la clase `Gato` llamado `Pelusa`. Ahora `Gato` tiene un constructor, el cual toma un entero como parámetro. Esto significa que usted debe pasarle un parámetro. Como no existe ningún parámetro en la línea 59, se marcará como un error.



**Tip**

Si usted proporciona *un* constructor, el compilador no lo proporcionará. Por lo tanto, si crea un constructor que tome un parámetro, entonces no tendrá un constructor predeterminado, a menos que usted mismo escriba uno.

La línea 61 muestra una llamada a un método de clase, llamado `Ladrar()`. Este método nunca fue declarado. Por lo tanto, es ilegal.

La línea 62 muestra que se asigna el valor 7 a `suEdad`. Como `suEdad` es un dato miembro privado, esto se marca como un error al compilar el programa.

#### ¿Por qué utilizar el compilador para atrapar errores?

Aunque sería maravilloso escribir código 100% libre de errores, pocos programadores han sido capaces de hacerlo. Sin embargo, muchos programadores han desarrollado un sistema para ayudar a minimizar los errores al atraparlos y solucionarlos con prontitud en el proceso.

Aunque los errores de compilación son exasperantes y son la ruina en la existencia de un programador, son menos nocivos que la alternativa. Un lenguaje débilmente tipificado le permite violar sus contratos sin que el compilador haga nada, pero su programa fallará en tiempo de ejecución (por ejemplo, cuando su jefe esté observando).

Los errores en tiempo de compilación (es decir, errores encontrados al momento de la compilación) son menos nocivos que los errores en tiempo de ejecución (es decir, los errores que se encuentran al estar ejecutando el programa). Esto se debe a que los errores en tiempo de compilación se pueden encontrar con mucha más confiabilidad. Es posible ejecutar un programa muchas veces sin pasar por cada posible ruta del código. Por lo tanto, un error en tiempo de ejecución puede permanecer oculto por bastante tiempo. Los errores en tiempo de compilación se encuentran cada vez que se compila. Por lo tanto, son más fáciles de identificar y de solucionar. La meta de la programación de calidad es asegurar que el código no tenga errores en tiempo de ejecución. Una técnica comprobada para lograr esto es utilizar el compilador para que atrape los errores lo más pronto posible en el proceso de desarrollo.

## Dónde colocar declaraciones de clases y definiciones de métodos

Cada función que usted declare para su clase debe tener una definición. La definición también se conoce como la *implementación del método*. Al igual que otras funciones, la definición del método de una clase tiene un encabezado y un cuerpo de función.

La definición se debe ubicar en un archivo que el compilador pueda encontrar. La mayoría de los compiladores de C++ requiere que ese archivo tenga la extensión .c o .cpp. Los compiladores GNU manejan las extensiones .c, .cpp, .cxx o .c++. Los archivos de listados utilizan .cxx para que usted pueda utilizarlos en Linux o en un sistema de archivos Microsoft Windows DOS/FAT. Si utiliza un compilador distinto, tendrá que determinar cuál extensión prefiere. Tenga cuidado con la sensibilidad a mayúsculas y minúsculas en Linux (.cxx no es lo mismo que .cxx).

 Nota

Los compiladores GNU dan por hecho que los archivos que terminan con .c son programas de C, y que los archivos de programas de C++ terminan con .cpp, .cxx o .c++. Puede utilizar cualquier extensión, pero .cpp minimizará la confusión.

También puede colocar la declaración en este archivo, pero ésa no es una buena práctica de programación. La convención que adopta la mayoría de los programadores es colocar la declaración en lo que se conoce como archivo de encabezado, generalmente con el mismo nombre pero con extensión .h, .hp o .hpp. En este libro los nombres de los archivos de encabezado terminan con .hpp, pero verifique qué terminación prefiere su compilador.

Por ejemplo, usted coloca la declaración de la clase Gato en un archivo llamado gato.hpp, y coloca la definición de los métodos de la clase en un archivo llamado gato.cxx. Luego adjunta el archivo de encabezado al archivo fuente de C++ colocando el siguiente código al principio de gato.cxx:

```
#include "gato.hpp"
```

Esto le indica al compilador que lea gato.hpp y que lo inserte en el archivo fuente, como si usted lo hubiera escrito directamente. Nota: Algunos compiladores insisten en que el uso de mayúsculas y minúsculas debe ser igual entre la instrucción #include y su sistema de archivos. Si usted declara #include "gato.hpp" y guarda el archivo de encabezado con el nombre GATO.hpp, posiblemente recibirá un error durante la compilación.

¿Por qué tomarse la molestia de separar el contenido de su archivo .hpp y su archivo .cxx si va a volver a insertar el contenido del archivo .hpp dentro del archivo .cxx? La mayor parte del tiempo, los clientes de su clase no se preocupan por los detalles específicos de la implementación. Con sólo leer el archivo de encabezado obtienen todo lo que necesitan saber, así que pueden ignorar los archivos de implementación. Además, usted podría necesitar incluir el archivo .hpp en más de un archivo .cxx.

El uso de archivos include (de encabezado) es un ejemplo de una buena ingeniería de software (es más sencillo reutilizar su código).

 Nota

La declaración de una clase le indica al compilador lo que hace la clase, qué datos guarda y qué funciones tiene. La declaración de la clase se conoce como su interfaz porque le indica al usuario cómo interactuar con la clase. Por lo regular, la interfaz se guarda en un archivo .hpp, el cual se conoce como archivo de encabezado.

La definición de la función le indica al compilador la forma en que ésta trabaja. La definición de la función se conoce como la implementación del método de la clase, y se guarda en un archivo .cxx. Los detalles de la implementación de la clase conciernen sólo al autor de la clase. Los clientes de la clase (es decir, los componentes del programa que utilizan la clase) tampoco necesitan, ni les preocupa, saber cómo se implementan las funciones.

## Aplicación de la implementación en línea

Así como puede pedir al compilador que convierta a una función normal en función en línea, también puede hacer que los métodos de una clase estén en línea. La palabra reservada `inline` aparece antes del tipo de valor de retorno. Por ejemplo, la implementación en línea de la función `ObtenerPeso()` se vería así:

```
inline int Gato::ObtenerPeso()
{
    return suPeso;           // regresar el dato miembro llamado suPeso
}
```

También puede colocar la definición de una función dentro de la declaración de la clase, lo que automáticamente hace que la función esté en línea, por ejemplo:

```
class Gato
{
public:
    int ObtenerPeso() { return suPeso; }    // en linea
    void AsignarPeso(int peso);
};
```

Observe la sintaxis de la definición de `ObtenerPeso()`. El cuerpo de la función en línea empieza inmediatamente después de la declaración del método de la clase; no se utiliza punto y coma después de los paréntesis. Al igual que con cualquier función, la definición empieza con una llave de apertura y termina con una llave de cierre. Como siempre, el espacio en blanco no importa; usted habría podido escribir la declaración de la siguiente manera:

```
class Gato
{
public:
    int ObtenerPeso() const
    {
        return suPeso;
    }                      // en linea
    void AsignarPeso(int peso);
};
```

Los listados 6.6a y 6.6b vuelven a crear la clase `Gato`, pero colocan la declaración en `lst06-06.hpp` y la implementación de las funciones en `lst06-06.cxx`). El listado 6.6b también convierte las funciones de acceso y la función `Maullar()` en funciones en línea.

6

### ENTRADA LISTADO 6.6a Declaración de la clase Gato en `lst06-06.hpp`

```
1:  // lst06-06.hpp
2:  #include <iostream.h>
3:  class Gato
```

continúa

**LISTADO 6.6a** CONTINUACIÓN

```

4:  {
5:  public:
6:    Gato (int edadInicial);
7:    -Gato();
8:    int ObtenerEdad() const { return suEdad; }
9:    void AsignarEdad (int edad) { suEdad = edad; }           // ien linea!
10:   void Maullar() const { cout << "Miau.\n"; }             // ien linea!
11: private:
12:    int suEdad;
13: };

```

**ENTRADA****LISTADO 6.6b** Implementación de Gato en lst06-06.cxx

```

1:  // Muestra de las funciones en línea
2:  // y la inclusión de archivos de encabezado
3:
4:  #include "lst06-06.hpp"          // iasegúrese de incluir los archivos de
5:                                // encabezado!
6:
7:  Gato::Gato(int edadInicial)    //constructor
8:  {
9:    suEdad = edadInicial;
10: }
11:
12: Gato::~Gato()                 //destructor, no realiza ninguna acción
13: {
14: }
15:
16: // Crear un gato, asignar un valor a su edad, hacer que maúlle,
17: // que nos diga su edad y que maúlle nuevamente.
18: int main()
19: {
20:   Gato Pelusa(5);
21:   Pelusa.Maullar();
22:   cout << "Pelusa es un gato que tiene " ;
23:   cout << Pelusa.ObtenerEdad() << " años de edad.\n";
24:   Pelusa.Maullar();
25:   Pelusa.AsignarEdad(7);
26:   cout << "Ahora Pelusa tiene " ;
27:   cout << Pelusa.ObtenerEdad() << " años de edad.\n";
28:   return 0;
29: }

```

**SALIDA**

Miau.  
Pelusa es un gato que tiene 5 años de edad.  
Miau.  
Ahora Pelusa tiene 7 años de edad.

**ANÁLISIS**

El código presentado en los listados 6.6a y 6.6b es similar al código del listado 6.4, excepto que tres de los métodos se escriben en línea en el archivo de declaración, y la declaración se ha separado en `1st06-06.hpp`.

En la línea 8 se declara `ObtenerEdad()`, y se proporciona su implementación en línea. Las líneas 9 y 10 proporcionan más funciones en línea, pero la funcionalidad de estas funciones permanece sin cambio a partir de las anteriores implementaciones “fuera de línea”.

La línea 4 del listado 6.6b muestra `#include "1st06-06.hpp"`, lo que hace que se incluya el listado de `1st06-06.hpp`. Al incluir `1st06-06.hpp`, ha indicado al precompilador que lea el archivo `1st06-06.hpp` y lo inserte en el archivo como si hubiera sido escrito directamente ahí, empezando en la línea 5.

Esta técnica le permite colocar sus declaraciones en un archivo distinto al de su implementación, y aún así tener esa declaración disponible cuando el compilador la necesite. Ésta es una técnica muy común en la programación en C++. Normalmente, las declaraciones de clases se encuentran en un archivo `.hpp` que luego se incluye, mediante la instrucción `#include`, en el archivo `.cxx` asociado.

Las líneas 18 a 29 repiten la función `main()` del listado 6.4. Esto muestra que convertir los métodos en métodos en línea no cambia su rendimiento.

## Uso de clases con otras clases como datos miembro

No es poco común formar una clase compleja mediante la declaración de clases más simples y luego incluirlas en la declaración de la clase más complicada. Por ejemplo, usted podría declarar una clase llamada rueda, una clase motor, una clase transmisión, entre otras, y luego combinarlas en una clase llamada auto. Esto establece una relación. Un auto tiene un motor, ruedas y una transmisión.

Considere otro ejemplo. Un rectángulo se compone de líneas. Una línea se define como dos puntos. Un punto se define como una coordenada x y una coordenada y. El listado 6.7b muestra la declaración completa de la clase `Rectangulo`, como podría aparecer en `rect.hpp` (abreviación de `rectangulo.hpp`). Debido a que un rectángulo se define como cuatro líneas que conectan cuatro puntos, y cada punto se refiere a una coordenada en una gráfica, primero se declara una clase llamada `Punto` para guardar las coordenadas (x, y) de cada punto. El listado 6.7a muestra una declaración completa de ambas clases.

**ENTRADA****LISTADO 6.7a** Declaración de una clase completa

```
1: // Inicio de lst06-07.hpp
2: #include <iostream.h>
3: class Punto      // guarda las coordenadas (x, y)
4: {
5:     // no hay constructor, usar el predeterminado
6:     public:
7:         void AsignarX(int x) { suX = x; }
8:         void AsignarY(int y) { suY = y; }
9:         int ObtenerX() const { return suX; }
10:        int ObtenerY() const { return suY; }
11:    private:
12:        int suX;
13:        int suY;
14:    }; // fin de la declaración de la clase Punto
15:
16:
17: class Rectangulo
18: {
19:     public:
20:         Rectangulo (int superior, int izquierdo, int inferior, int derecho);
21:         ~Rectangulo () {}
22:
23:         int ObtenerSuperior() const { return suSuperior; }
24:         int ObtenerIzquierdo() const { return suIzquierdo; }
25:         int ObtenerInferior() const { return suInferior; }
26:         int ObtenerDerecho() const { return suDerecho; }
27:
28:         Punto ObtenerSupIzq() const { return suSupIzq; }
29:         Punto ObtenerInfIzq() const { return suInfIzq; }
30:         Punto ObtenerSupDer() const { return suSupDer; }
31:         Punto ObtenerInfDer() const { return suInfDer; }
32:
33:         void AsignarSupIzq(Punto Ubicacion) { suSupIzq = Ubicacion; }
34:         void AsignarInfIzq(Punto Ubicacion) { suInfIzq = Ubicacion; }
35:         void AsignarSupDer(Punto Ubicacion) { suSupDer = Ubicacion; }
36:         void AsignarInfDer(Punto Ubicacion) { suInfDer = Ubicacion; }
37:
38:         void AsignarSuperior(int superior) { suSuperior = superior; }
39:         void AsignarIzquierdo (int izquierdo) { suIzquierdo = izquierdo; }
40:         void AsignarInferior (int inferior) { suInferior = inferior; }
41:         void AsignarDerecho (int derecho) { suDerecho = derecho; }
42:
43:         int ObtenerArea() const;
44:
45:     private:
46:         Punto suSupIzq;
47:         Punto suSupDer;
48:         Punto suInfIzq;
```

```

49:     Punto suInfDer;
50:     int suSuperior;
51:     int suIzquierdo;
52:     int suInferior;
53:     int suDerecho;
54: };

```

**ENTRADA LISTADO 6.7b lst06-07.cxx**

```

1: // Inicio de lst06-07.cxx
2: #include "lst06-07.hpp"
3: Rectangulo::Rectangulo(int superior, int izquierdo, int inferior,
int derecho)
4: {
5:     suSuperior = superior;
6:     suIzquierdo = izquierdo;
7:     suInferior = inferior;
8:     suDerecho = derecho;
9:
10:    suSupIzq.AsignarX(izquierdo);
11:    suSupIzq.AsignarY(superior);
12:
13:    suSupDer.AsignarX(derecho);
14:    suSupDer.AsignarY(superior);
15:
16:    suInfIzq.AsignarX(izquierdo);
17:    suInfIzq.AsignarY(inferior);
18:
19:    suInfDer.AsignarX(derecho);
20:    suInfDer.AsignarY(inferior);
21: }
22:
23:
24: // calcular área del rectángulo encontrando los lados,
25: // establecer ancho y altura y luego multiplicar
26: int Rectangulo::ObtenerArea() const
27: {
28:     int Ancho = suDerecho - suIzquierdo;
29:     int Altura = suSuperior - suInferior;
30:     return (Ancho * Altura);
31: }
32:
33: int main()
34: {
35:     //inicializar una variable Rectangulo local
36:     Rectangulo MiRectangulo (100, 20, 50, 80);
37:
38:     int area = MiRectangulo.ObtenerArea();
39:

```

**LISTADO 6.7b** CONTINUACIÓN

---

```

40:     cout << "Área: " << area << "\n";
41:     cout << "Coordenada X superior izquierda: ";
42:     cout << MiRectangulo.ObtenerSupIzq().ObtenerX() << endl;
43:     return 0;
44: }
```

---

**SALIDA**

Área: 3000  
Coordenada X superior izquierda: 20

**ANÁLISIS**

En las líneas 3 a 14 del listado 6.7a se declara la clase Punto, la cual se utiliza para guardar las coordenadas (x, y) específicas en una gráfica. Así como está escrito, este programa no utiliza mucho la clase Punto. Sin embargo, otros métodos de dibujo requieren del uso de Punto.

En las líneas 12 y 13, dentro de la declaración de la clase Punto, se declaran dos variables miembro (suX y suY). Estas variables guardarán los valores de las coordenadas. Al incrementarse la coordenada x, usted se mueve hacia la derecha en la gráfica. Al incrementarse la coordenada y, se mueve hacia arriba en la gráfica. Otras gráficas utilizan diferentes sistemas. Por ejemplo, algunos programas que utilizan ventanas incrementan la coordenada y a medida que se avanza hacia abajo en la ventana.

La clase Punto utiliza funciones de acceso en línea para obtener y asignar un valor a las coordenadas X y Y declaradas en las líneas 7 a 10. Los objetos Punto utilizarán el constructor y el destructor predeterminados. Por lo tanto, usted debe establecer sus coordenadas en forma explícita.

En la línea 17 empieza la declaración de la clase Rectangulo. Un Rectangulo consta de cuatro puntos que representan sus esquinas (no los confunda con los cuatro parámetros que representan los lados).

El constructor para Rectangulo (línea 20) toma cuatro enteros como parámetros, llamados superior, izquierdo, inferior y derecho. Los cuatro parámetros para el constructor se copian en cuatro variables miembro (listado 6.7b), y luego se establecen los cuatro objetos Punto.

Además de las funciones de acceso usuales, Rectangulo tiene una función llamada ObtenerArea() declarada en la línea 43. En lugar de guardar el área como variable, la función ObtenerArea() calcula el área en las líneas 28 a 30 del listado 6.7b. Para hacer esto, calcula el ancho y la altura del rectángulo, y luego multiplica estos dos valores.

Para obtener la coordenada x de la esquina superior izquierda del rectángulo, necesita acceder al punto suSupIzq y pedir a ese punto el valor de suX. Como ObtenerSupIzq() es un método de Rectangulo, puede acceder directamente a los datos privados de Rectangulo, incluyendo suSupIzq. Como suSupIzq es un Punto y el valor suX de Punto es privado, ObtenerSupIzq() no puede tener acceso directo a este dato. En vez de eso, debe utilizar el método de acceso público llamado ObtenerX() para obtener ese valor.

La línea 33 del listado 6.7b es el comienzo del cuerpo del programa en sí. Hasta la línea 36, no se ha asignado memoria, y no ha pasado realmente nada. Lo único que usted hizo fue decir al compilador cómo hacer un punto y cómo hacer un rectángulo, en caso de que se necesite uno.

En la línea 36 se define un `Rectangulo` pasando valores para `Superior`, `Izquierdo`, `Inferior` y `Derecho`.

En la línea 38 se crea una variable local de tipo entero llamada `area`. Esta variable guarda el área del `Rectangulo` creado anteriormente. `Area` se inicializa con el valor regresado por la función `ObtenerArea()` del objeto `MiRectangulo`.

Un cliente de `Rectangulo` podría crear un objeto de la clase `Rectangulo` y obtener su área sin siquiera ver la implementación de `ObtenerArea()`.

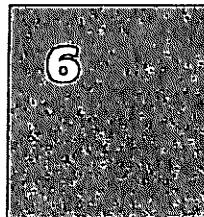
`1st06-07.hpp` se muestra en el listado 6.7a. Con sólo mirar el archivo de encabezado, el cual contiene la declaración de la clase `Rectangulo`, el programador sabe que `ObtenerArea()` regresa un entero. La forma en que `ObtenerArea()` realiza su trabajo no es de importancia para el usuario de objetos de la clase `Rectangulo`. De hecho, el autor de la clase `Rectangulo` podría cambiar el método `ObtenerArea()` sin afectar los programas que utilizan esta clase.

#### Preguntas frecuentes

**FAQ: ¿Cuál es la diferencia entre declarar y definir?**

**Respuesta:** Una declaración presenta el nombre de algo y las acciones que ese algo puede realizar. Una definición, además de presentarnos el nombre, nos dice qué hace y cómo lo hace; es la implementación de lo que se presenta en la declaración.

Con unas cuantas excepciones, todas las definiciones son también declaraciones. Las excepciones más importantes son la declaración de una función global (un prototipo) y la declaración de una clase (por lo general en un archivo de encabezado).



## Uso de estructuras

Un parente muy cercano de la palabra reservada `class` es la palabra reservada `struct`, que se utiliza para declarar una estructura. En C++, una estructura es lo mismo que una clase, con la excepción de que sus miembros son públicos de manera predeterminada. Puede declarar una estructura exactamente igual que como declara una clase, y puede darle los mismos datos miembro y funciones. De hecho, si sigue la buena práctica de programación de siempre declarar explícitamente las secciones públicas y privadas de su clase, no existirá ninguna diferencia.

Vuelva a escribir el listado 6.7a con estos cambios:

- En la línea 3, cambie class Punto por struct Punto.
- En la línea 17, cambie class Rectangulo por struct Rectangulo.

Ahora ejecute el programa de nuevo y compare la salida. No debe haber ningún cambio.

## Por qué dos palabras reservadas hacen lo mismo

Probablemente se esté preguntando por qué dos palabras reservadas hacen lo mismo. Éste es un accidente de la historia. Cuando se desarrolló C++, se creó como una extensión del lenguaje C. C tiene estructuras, aunque las estructuras de C no tienen métodos de clases. Bjarne Stroustrup, el creador de C++, hizo su creación a partir de las estructuras (struct), pero cambió el nombre a clases (class) para representar la nueva funcionalidad expandida.

| DEBE                                                                                                                               | NO DEBE |
|------------------------------------------------------------------------------------------------------------------------------------|---------|
| <b>DEBE</b> colocar su declaración de la clase en un archivo .hpp y la implementación de sus funciones miembro en un archivo .cxx. |         |
| <b>DEBE</b> utilizar const siempre que pueda.                                                                                      |         |
| <b>DEBE</b> comprender las clases antes de avanzar a otros temas.                                                                  |         |

## Resumen

Hoy aprendió cómo crear nuevos tipos de datos llamados clases. Aprendió cómo definir variables de estos nuevos tipos, las cuales se conocen como objetos.

Una clase tiene datos miembro, que son variables de diversos tipos, incluyendo a otras clases. Una clase también incluye funciones miembro (también conocidas como métodos). Estas funciones miembro se utilizan para manipular los datos miembro y para realizar otros servicios.

Los miembros de las clases, tanto datos como funciones, pueden ser públicos o privados. Los miembros públicos son accesibles para cualquier parte del programa. Los miembros privados son accesibles sólo para las funciones miembro de la clase.

Es una buena práctica de programación aislar la interfaz, o declaración, de la clase en un archivo de encabezado. Por lo general, esto se hace en un archivo con extensión .hpp. La implementación de los métodos de la clase se escribe en un archivo con extensión .cxx.

Los constructores de clases inicializan objetos. Los destructores de clases destruyen objetos y se utilizan normalmente para liberar memoria asignada por los métodos de la clase.

## Preguntas y respuestas

**P ¿Qué tan grande es el objeto de una clase?**

**R** El tamaño en memoria del objeto de una clase se determina por medio de la suma de los tamaños de sus variables miembro. Los métodos de las clases no ocupan espacio como parte de la memoria reservada para el objeto.

Algunos compiladores alinean variables en memoria de tal forma que las variables de 2 bytes en realidad consumen un poco más de 2 bytes. Revise el manual de su compilador para estar seguro, pero en este momento no necesita preocuparse por estos detalles.

**P Si declaro una clase Gato con un miembro privado llamado suEdad, y luego defino dos objetos Gato, Pelusa y Silvestre, ¿puede Silvestre acceder a la variable miembro suEdad de Pelusa?**

**R** Sí. Los datos privados están disponibles para las funciones miembro de una clase, e instancias diferentes de una clase pueden acceder a los datos entre sí. En otras palabras, si Pelusa y Silvestre son instancias de Gato, las funciones miembro de Pelusa pueden acceder a los datos de Pelusa y también a los datos de Silvestre.

**P ¿Por qué no debo hacer públicos todos los datos miembro?**

**R** Hacer los datos miembro privados le permite al cliente de la clase utilizar los datos sin preocuparse por la forma en que se guardan o se calculan. Por ejemplo, si la clase Gato tiene un método llamado ObtenerEdad(), los clientes de la clase Gato pueden preguntar la edad del gato sin saber ni preocuparse si el gato guarda su edad en una variable miembro o si calcula su edad en el momento.

**P Si usar una función const para cambiar la clase produce un error de compilación, ¿por qué no omitir la palabra const para estar seguro de evitar errores?**

**R** Si su función miembro no debe cambiar la clase, usar la palabra reservada const es una buena forma de hacer que el compilador lo ayude a encontrar errores típicos. Por ejemplo, tal vez ObtenerEdad() no tenga motivo para cambiar la clase Gato, pero su implementación podría tener la siguiente línea:

```
if (Pelusa. suEdad = 100) cout << "¡Hey! Pelusa tiene 100 años de edad\n";
```

Declarar ObtenerEdad() como const ocasiona que este código se marque como error. Usted quiso comprobar si suEdad era igual a 100, pero en vez de eso asignó sin querer 100 a suEdad. Debido a que esta asignación cambia la clase, y usted dijo que este método no cambiaría la clase, el compilador puede encontrar el error.

Este tipo de error puede ser difícil de encontrar si sólo se examina el código. A menudo, el ojo sólo ve lo que espera ver. Lo que es más importante, puede parecer que el programa se ejecuta correctamente, pero ahora se le ha asignado un número raro a suEdad. Esto ocasionará problemas tarde o temprano.

**P ¿Existe alguna razón para utilizar una estructura en un programa de C++?**

**R** Muchos programadores de C++ reservan la palabra reservada `struct` para clases que no tienen funciones. Esto es un recordatorio de las viejas estructuras de C, las cuales no podían tener funciones. Francamente, esto me parece confuso y una mala práctica de programación. La estructura sin métodos de hoy podría necesitar métodos en el futuro. Entonces usted estaría obligado a cambiar el tipo a clase, o a quebrantar su regla y terminar con una estructura que tenga métodos. Por supuesto, C++ debe tener compatibilidad con el código de C. Por lo general, el uso de `struct` se reserva para programas de C que se compilan y se mantienen en compiladores de C++.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Qué es el operador de punto, y para qué se utiliza?
2. ¿Cuál de las dos acciones reserva memoria, la declaración o la creación?
3. ¿Qué es la declaración de una clase, su interfaz o su implementación?
4. ¿Cuál es la diferencia entre datos miembro públicos y privados?
5. ¿Se pueden establecer métodos privados?
6. ¿Se pueden establecer datos miembro públicos?
7. Si declara dos objetos Gato, ¿pueden éstos tener distintos valores en sus datos miembro `suEdad`?
8. ¿Terminan con un punto y coma las declaraciones de clases? ¿Y las definiciones de los métodos de clases?
9. ¿Cuál sería el encabezado para un método de la clase Gato, llamado `Maullar`, que no toma parámetros y regresa `void`?
10. ¿Qué función se llama para inicializar una clase?

### Ejercicios

1. Escriba el código que declare una clase llamada `Empleado` con estos datos miembro: `edad`, `aniosDeServicio` y `salario`.
2. Vuelva a escribir la clase `Empleado` para hacer los datos miembro privados, y proporcione métodos de acceso públicos para obtener y asignar un valor para cada uno de los datos miembro.

3. Escriba un programa con la clase `Empleado` que cree dos empleados; que asigne un valor a los datos miembro `edad`, `aniosDeServicio` y `salario`, y que imprima sus valores.
4. Como continuación del ejercicio 3, proporcione un método de la clase `Empleado` que reporte cuántos miles de pesos gana el empleado, redondeados al múltiplo de 1000 más cercano.
5. Cambie la clase `Empleado` de forma que pueda inicializar `edad`, `aniosDeServicio` y `salario` al crear al empleado.
6. **CAZA ERRORES:** ¿Qué está mal en la siguiente declaración?

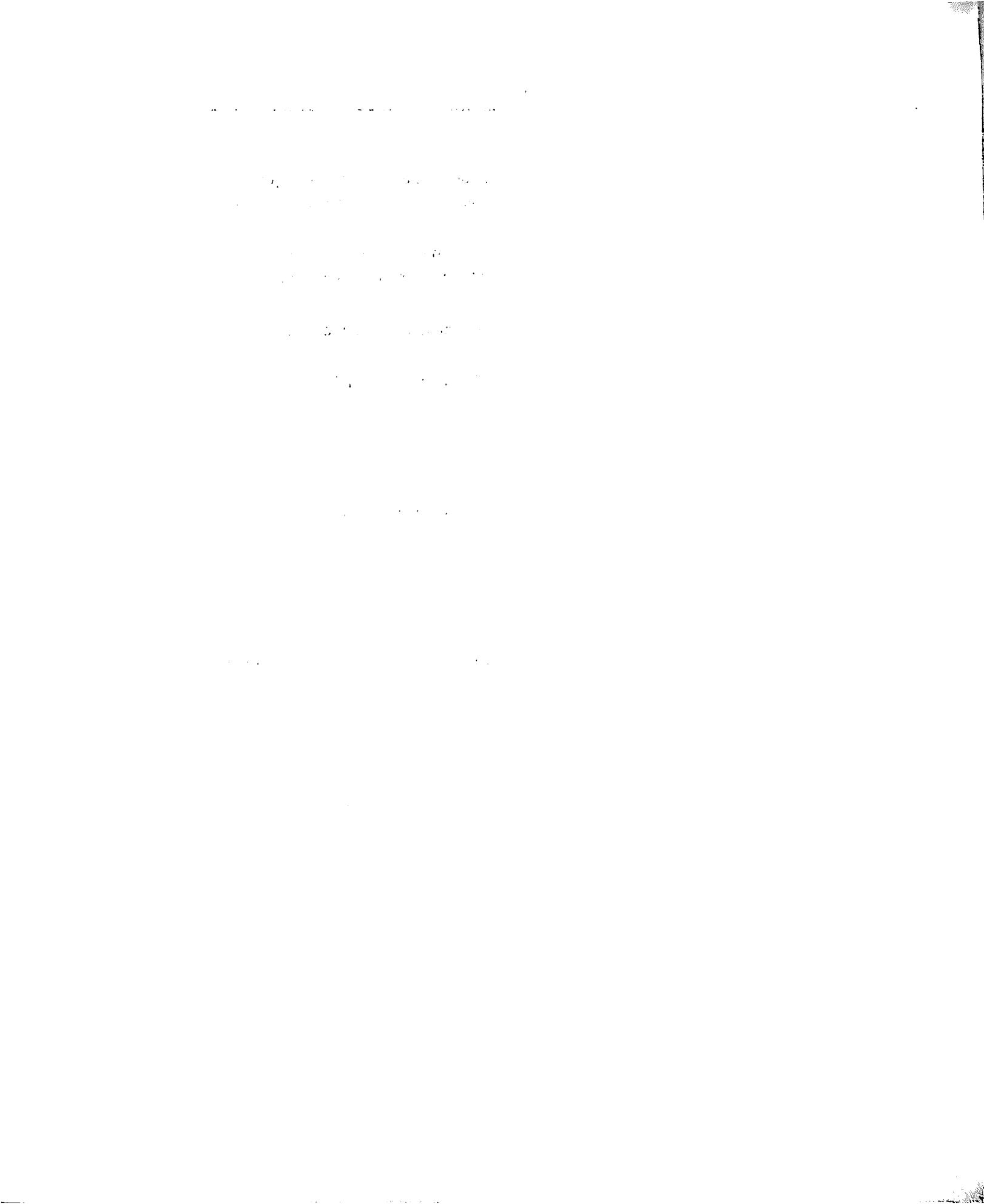
```
class Cuadrado
{
public:
    int Lado;
}
```

7. **CAZA ERRORES:** ¿Por qué no es muy útil la siguiente declaración de clase?

```
class Gato
{
    int ObtenerEdad()const;
private:
    int suEdad;
};
```

8. **CAZA ERRORES:** ¿Cuáles son los tres errores que encontrará el compilador en este código?

```
class TV
{
public:
    void AsignarEstacion(int estacion);
    int ObtenerEstacion() const;
private:
    int suEstacion;
};
int main()
{
    TV miTV;
    miTV.suEstacion = 9;
    TV.AsignarEstacion(10);
    TV miOtraTV(2);
    return 0;
}
```



# SEMANA 1

Día 7

## Más flujo de programa

Los programas realizan la mayor parte de su trabajo mediante la ramificación y los ciclos. En el día 4, “Expresiones e instrucciones”, aprendió cómo ramificar su programa por medio de la instrucción `if`. Hoy aprenderá lo siguiente:

- Qué son los ciclos y cómo se utilizan
- Cómo construir varios ciclos
- Una alternativa para las instrucciones `if/else` complejas

### Uso de los ciclos

Muchos problemas de programación se solucionan al actuar en forma repetida sobre los mismos datos. Dos maneras de hacer esto son la recursión (que se explicó en el día 5, “Funciones”) y la iteración. Iteración significa hacer lo mismo una y otra vez. El principal método de iteración es el ciclo.

#### Las raíces del uso de ciclos: la instrucción `goto`

En los días primitivos de la ciencia computacional, los programas eran péudos, torpes y cortos. Los ciclos consistían en una etiqueta, algunas instrucciones y un salto.

En C++, una etiqueta es sólo un nombre seguido de un signo de dos punto (:). La etiqueta se coloca a la izquierda de una instrucción válida de C++ o en su propia línea, y un salto se logra al escribir `goto` seguido del nombre de la etiqueta. El listado 7.1 muestra esto.


**Precaución**

Es una mala práctica utilizar la instrucción `goto`. Ésta se incluye aquí sólo para cubrir todas las cuestiones relacionadas con los ciclos, y por razones históricas. ¡Los buenos programadores evitan el uso de esta instrucción!

**ENTRADA****LISTADO 7.1** Uso de ciclos con la palabra reservada `goto`

```

1: // Listado 7.1
2: // Uso de ciclos con goto
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:     int contador = 0;           // inicializar contador
9:     ciclo: contador++;        // principio del ciclo
10:    cout << "contador: " << contador << "\n";
11:    if (contador < 5)          // evaluar el valor
12:        goto ciclo;           // saltar al principio del ciclo
13:
14:    cout << "Completo. Contador: " << contador << ".\n";
15:
16: }
```

**SALIDA**

```

contador: 1
contador: 2
contador: 3
contador: 4
contador: 5
Completo. Contador: 5.
```

**ANÁLISIS**

En la línea 8 se inicializa `contador` en 0. En la línea 9, la etiqueta `ciclo` marca el inicio del ciclo. Se incrementa `contador` y se imprime su nuevo valor. El valor de `contador` se evalúa en la línea 11. Si es menor que 5, la instrucción `if` es `true` (verdadera) y se ejecuta la instrucción `goto`. Esto ocasiona que la ejecución del programa regrese a la línea 9. El programa continuará en el ciclo hasta que `contador` sea igual a 5, momento en el que “saldrá” del ciclo y se imprimirá la salida final.

## Por qué se evita el uso de goto

La instrucción goto ha recibido muchas críticas negativas últimamente, y son bien merecidas. Esta instrucción puede originar un salto hacia cualquier ubicación del código fuente, hacia atrás o hacia adelante. Su uso indiscriminado produce programas enredados, malos e imposibles de leer, lo que se conoce como "código espagueti". Debido a esto, los maestros de la ciencia computacional han pasado los últimos 20 años tratando de meter una lección en la cabeza de sus estudiantes: "¡Nunca, pero nunca, utilicen goto!"

Uno de los mayores problemas de depuración con goto es que usted no sabe cómo llega a esa etiqueta. Tiene que buscar instrucciones goto con esa etiqueta en todo el código y luego tiene que descubrir cuál de ellas lo envió allá. ¡Esto es mucho trabajo!

Para evitar el uso de goto, se han introducido comandos para uso de ciclos más sofisticados y estrechamente controlados: `for`, `while` y `do...while`. El uso de estos comandos hace que los programas sean más fáciles de comprender, y por lo general se evita el uso de goto, pero alguien podría argumentar que esto es un poco exagerado. Igual que cualquier herramienta en las manos apropiadas y utilizada cuidadosamente, goto puede ser una instrucción útil, y el comité ANSI decidió mantenerla en el lenguaje porque tiene sus usos legítimos. Pero como dicen: "Niños, no intenten esto en casa".

### La instrucción goto

Para utilizar la instrucción goto, se escribe la palabra reservada `goto` seguida de un nombre de etiqueta. Esto produce un salto incondicional hacia la etiqueta.

Ejemplo:

```
if (valor > 10)      goto end;if (valor < 10)
goto end;cout << "El valor es 10!";
end: cout << "listo";
```

### Precaución

El uso de goto es casi siempre un signo de un mal diseño. El mejor consejo es evitar su uso. En nuestros casi 20 años combinados de programación, la hemos utilizado sólo unas cuantas veces.

## Ciclos while

Un ciclo while ocasiona que su programa repita una secuencia de instrucciones siempre y cuando la condición de inicio permanezca verdadera (`true`). En el ejemplo de goto del listado 7.1, contador se incrementaba hasta llegar a 5. El listado 7.2 muestra el mismo programa reescrito para aprovechar las ventajas que ofrece un ciclo while.

**ENTRADA****LISTADO 7.2 Ciclos while**

```

1: // Listado 7.2
2: // Uso de ciclos con while
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:     int contador = 0;           // inicializar la condición
9:
10:    while(contador < 5)        // evaluar que la condición aún sea verdadera
11:    {
12:        contador++;          // cuerpo del ciclo
13:        cout << "contador: " << contador << "\n";
14:    }
15:
16:    cout << "Completo. Contador: " << contador << ".\n";
17:    return 0;
18: }
```

**SALIDA**

```

contador: 1
contador: 2
contador: 3
contador: 4
contador: 5
Completo. Contador: 5.
```

**ANÁLISIS**

Este programa sencillo muestra los fundamentos del ciclo `while`. Se evalúa una condición, y si es verdadera, se ejecuta el cuerpo del ciclo `while`. En este caso, la condición que se evalúa en la línea 10 es si el contenido de la variable `contador` es menor que 5. Si la condición es verdadera, se ejecuta el cuerpo del ciclo; en la línea 12 se incrementa `contador`, y en la línea 13 se imprime su valor. Cuando falle la instrucción condicional de la línea 10 (cuando `contador` ya no sea menor que 5), no se ejecutará más el cuerpo del ciclo `while` (líneas 11 a 14). La ejecución del programa se irá hasta la línea 15.

**La instrucción while**

La sintaxis para la instrucción `while` es la siguiente:

```
while ( condición )
instrucción;
```

`condición` es cualquier expresión de C++, e `instrucción` es cualquier instrucción o bloque de instrucciones válido de C++. Cuando `condición` es verdadera (1), se ejecuta

instrucción, y luego se vuelve a evaluar condición. Esto continúa hasta que condición sea falsa (0), momento en el que termina el ciclo while y la ejecución continúa en la siguiente línea después de instrucción.

#### Ejemplo

```
// contar hasta 10
int x = 0;
while (x < 10)
cout << "X: " << x++;
```

## Instrucciones while más complicadas

La condición evaluada por un ciclo while puede ser tan compleja como cualquier expresión válida de C++. Esto incluye a las expresiones producidas mediante el uso de los operadores lógicos && (AND), || (OR) y ! (NOT). El listado 7.3 muestra una instrucción while un poco más complicada.

### ENTRADA LISTADO 7.3 Ciclos while complejos

```
1: // Listado 7.3
2: // Instrucciones while complejas
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:     unsigned short chico;
9:     unsigned long grande;
10:    const unsigned short MAXCHICO=65535;
11:
12:    cout << "Escriba un número chico: ";
13:    cin >> chico;
14:    cout << "Escriba un número grande: ";
15:    cin >> grande;
16:
17:    cout << "chico: " << chico << "...";
18:
19:    // para cada iteración, evaluar tres condiciones
20:    while (chico < grande && grande > 0 && chico < MAXCHICO)
21:    {
22:        if (chico % 5000 == 0) // escribir un punto cada 5 mil líneas
23:            cout << ".";
24:
25:        chico++;
26:
27:        grande-=2;
28:    }
```

**LISTADO 7.3** CONTINUACIÓN

```

29:
30:     cout << "\nChico: " << chico << " Grande: " << grande << endl;
31:     return 0;
32: }
```

**SALIDA**

```

Escriba un número chico: 2
Escriba un número grande: 100000
chico: 2.....
Chico: 33335 Grande: 33334
```

**ANÁLISIS** Este programa es un juego. Escriba dos números, uno chico y uno grande. El número más chico contará en incrementos de uno, y el número más grande contará en decrementos de dos. El objetivo del juego es adivinar cuándo se encontrarán los dos números.

En las líneas 12 a 15 se escriben los números. En la línea 20 empieza un ciclo `while`, que continuará sólo mientras se cumplan tres condiciones:

1. chico no debe ser mayor que grande.
2. grande no debe ser negativo ni cero.
3. chico no debe sobrepasar el valor de un entero chico (`MAXCHICO`).

En la línea 22 se calcula el valor de chico en 5,000 por medio del operador de residuo. Esto no cambia el valor de chico; sin embargo, sólo regresa el valor 0 cuando `chico` sea un múltiplo exacto de 5,000. Cada vez que sea así, se imprime un punto (.) en pantalla para mostrar el progreso, chico se incrementa en la línea 25, y grande se decremente en 2 en la línea 27.

Cuando falle cualquiera de las tres condiciones del ciclo `while`, éste terminará, y la ejecución del programa continuará después de la llave de cierre del ciclo `while` en la línea 29.


**Nota**

El operador de residuo (%) y las condiciones compuestas se describen en el día 4.

## Uso de `continue` y `break` en ciclos

Algunas veces necesita regresar al principio de un ciclo `while` antes de que se ejecute todo el conjunto de instrucciones que está dentro del ciclo `while`. La instrucción `continue` salta hacia el principio del ciclo.

Otras veces, tal vez necesite salir del ciclo antes de que se cumplan las condiciones de salida. La instrucción `break` sale inmediatamente del ciclo `while`, y la ejecución del programa continúa después de la llave de cierre.

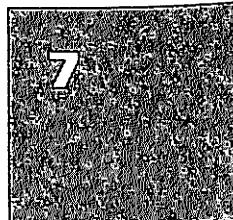
El listado 7.4 muestra el uso de estas instrucciones. Esta vez el juego se ha vuelto más complicado. Se invita al usuario a que escriba un número chico y un número grande, un valor de salto y un número de destino. El número chico se incrementará en uno, y el número grande se decrementará en 2. Cada vez que el número chico sea un múltiplo de salto, no se realizará el decremento. El juego termina si chico se hace mayor que grande. Si el número grande llega al destino en forma exacta, se imprimirá una instrucción y el juego terminará.

El objetivo del usuario es colocar un número de destino para el número grande que haga que se detenga el juego.

**ENTRADA LISTADO 7.4 break y continue**

```
1: // Listado 7.4
2: // Muestra de break y continue
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:     unsigned short chico;
9:     unsigned long grande;
10:    unsigned long salto;
11:    unsigned long destino;
12:    const unsigned short MAXCHICO=65535;
13:
14:    cout << "Escriba un número chico: ";
15:    cin >> chico;
16:    cout << "Escriba un número grande: ";
17:    cin >> grande;
18:    cout << "Escriba un valor de salto: ";
19:    cin >> salto;
20:    cout << "Escriba un número de destino: ";
21:    cin >> destino;
22:
23:    cout << "\n";
24:
25:    // establece las 3 condiciones para detener el ciclo
26:    while (chico < grande && grande > 0 && chico < MAXCHICO)
27:
28:    {
29:
30:        chico++;
31:
32:        if (chico % salto == 0) // ¿saltar el decremento?
33:        {
34:            cout << "salto en " << chico << endl;
```

continúa



**LISTADO 7.4** CONTINUACIÓN

```

35:         continue;
36:     }
37:
38:     if (grande == destino) // & grande concuerda exactamente con destino?
39:     {
40:         cout << "¡Destino alcanzado!";
41:         break;
42:     }
43:
44:     grande-=2;
45: }                     // fin del ciclo while
46:
47: cout << "\nChico: " << chico << " Grande: " << grande << endl;
48: return 0;
49: }
```

**SALIDA**

Escriba un número chico: 2  
 Escriba un número grande: 20  
 Escriba un valor de salto: 4  
 Escriba un número de destino: 6  
 salto en 4  
 salto en 8  
 Chico: 10 Grande: 8

**ANÁLISIS**

En este juego, el usuario perdió; chico se hizo mayor que grande antes de alcanzar el número seis.

En la línea 26 se prueban las condiciones del ciclo while. Si chico aún es menor que grande, grande es mayor que 0, y chico no ha sobrepasado el valor de MAXCHICO, entonces se entra al cuerpo del ciclo while.

En la línea 32 se realiza la operación de módulo del valor de chico con el valor de salto. Si chico es múltiplo de salto, se llega a la instrucción continue y la ejecución del programa salta hacia el principio del ciclo en la línea 26. Esto efectivamente elimina la evaluación para el destino y el decremento de grande.

En la línea 38 se compara destino con el valor de grande. Si son iguales, el usuario ha ganado. Entonces se imprime un mensaje y se llega a la instrucción break. Esto ocasiona una salida inmediata del ciclo while, y la ejecución del programa continúa en la línea 46.

**Nota**

Tanto `continue` como `break` se deben utilizar con cuidado. Son los comandos más peligrosos después de `goto`, casi por las mismas razones. Los programas que cambian repentinamente de dirección son más difíciles de comprender, y el uso indiscriminado de `continue` y `break` puede ocasionar que hasta un ciclo `while` pequeño sea ilegible.

**La instrucción `continue`**

`continue;` ocasiona que los ciclos `while` o `for` empiecen de nuevo al principio del ciclo.  
Vea el listado 7.4 para un ejemplo del uso de `continue`.

**La instrucción `break`**

`break;` ocasiona la terminación inmediata de los ciclos `while` o `for`. La ejecución salta hasta la siguiente instrucción después de la llave de cierre.

**Ejemplo**

```
while (condición)
{
    if (condición2)
        break;
    // instrucciones;
}
```

## Ciclos `while` (`true`)

La condición evaluada en un ciclo `while` puede ser cualquier expresión válida de C++. Mientras que esa condición sea verdadera (`true`), el ciclo `while` continuará. Usted puede crear un ciclo que nunca termine usando el valor `true` para la condición que se va a evaluar. El listado 7.5 muestra el conteo hasta 10 usando esta instrucción.

**ENTRADA** **LISTADO 7.5** Ciclos `while`

```
1: // Listado 7.5
2: // Muestra de un ciclo while (true)
3:
4: #include <iostream.h>
```

*continua*

**LISTADO 7.5** CONTINUACIÓN

```

5:
6:     int main()
7:     {
8:         int contador = 0;
9:
10:        while (true)
11:        {
12:            contador++;
13:            if (contador > 10)
14:                break;
15:        }
16:        cout << "Contador: " << contador << "\n";
17:        return 0;
18:    }

```

**SALIDA**

Contador: 11

**ANÁLISIS**

En la línea 10 hay un ciclo `while` con una condición que nunca podrá ser falsa.

El ciclo incrementa la variable `contador` en la línea 12, y luego en la línea 13 prueba si `contador` pasa de 10. Si no pasa, el ciclo `while` hace una iteración. Si `contador` es mayor que 10, la instrucción `break` de la línea 14 termina el ciclo `while`, y la ejecución del programa se va hasta la línea 16, en donde se imprimen los resultados.

Este programa funciona, pero no es muy elegante. Éste es un buen ejemplo del uso de la herramienta incorrecta para el trabajo. Se puede lograr lo mismo colocando la prueba del valor de `contador` en donde pertenece (en la condición `while`).

Otro nombre para esto es “While eterno” o “ciclo infinito”.

**Precaución**

Los ciclos eternos como `while (true)` pueden ocasionar que su computadora se paralice si nunca se llega a la condición de salida. Utilice estos ciclos con precaución y pruébelos exhaustivamente.

C++ le proporciona muchas maneras de lograr la misma tarea. El verdadero truco es elegir la herramienta adecuada para un trabajo específico.

| DEBE                                                                              | NO DEBE                                      |
|-----------------------------------------------------------------------------------|----------------------------------------------|
| <b>DEBE</b> utilizar ciclos while para iterar cuando una condición sea verdadera. | <b>NO DEBE</b> utilizar la instrucción goto. |
| <b>DEBE</b> tener cuidado al usar instrucciones continue y break.                 |                                              |
| <b>DEBE</b> asegurarse que su ciclo tenga fin.                                    |                                              |

## Limitaciones del ciclo while

Es posible que el cuerpo de un ciclo while nunca se ejecute. La instrucción while prueba su condición antes de ejecutar cualquiera de sus instrucciones, y si la condición es falsa (`false`), se salta todo el cuerpo del ciclo while. El listado 7.6 muestra esto.

### ENTRADA **LISTADO 7.6** Cómo saltar el cuerpo del ciclo while

```

1: // Listado 7.6
2: // Muestra cómo se salta el cuerpo del
3: // ciclo while cuando la condición es falsa.
4:
5: #include <iostream.h>
6:
7: int main()
8: {
9:     int contador;
10:
11:    cout << "¿Cuántos holas?: ";
12:    cin >> contador;
13:    while (contador > 0)
14:    {
15:        cout << "¡Hola!\n";
16:        contador--;
17:    }
18:    cout << "Contador vale: " << contador << endl;
19:    return 0;
20: }
```

### SALIDA

```

¿Cuántos holas?: 2
¡Hola!
¡Hola!
Contador vale: 0
```

Si se ejecuta el programa por segunda vez:

```

¿Cuántos holas?: 0
Contador vale: 0
```

**ANÁLISIS**

En la línea 11 se pide al usuario un valor de inicio. Este valor de inicio se guarda en la variable entera contador. El valor de contador se evalúa en la línea 13 y se decrementa en el cuerpo del ciclo `while`. La primera vez que la ejecución del programa pasó por el cuerpo del ciclo, contador tenía el valor 2, y por consecuencia el cuerpo del ciclo `while` se ejecutó dos veces. Sin embargo, la segunda vez el usuario escribió un 0. El valor de contador se probó en la línea 13 y la condición fue falsa; contador no era mayor que 0. Todo el cuerpo del ciclo `while` se saltó y nunca se imprimió Hola.

¿Qué pasa si usted quiere asegurar que Hola se imprima por lo menos una vez? El ciclo `while` no puede lograr esto debido a que la condición se prueba antes de que se realice cualquier impresión. Puede forzar esto con una instrucción `if` justo antes de entrar al ciclo `while`, como en el siguiente ejemplo:

```
if (contador < 1) // forzar un valor mínimo
    contador = 1;
```

pero eso es lo que los programadores llaman un "parche", una solución nada elegante. Existe una mejor solución: el ciclo `do...while`, el cual se describe en la siguiente sección.

## Ciclos `do...while`

El ciclo `do...while` ejecuta el cuerpo del ciclo antes de que se evalúe su condición y asegura que el cuerpo siempre se execute por lo menos una vez. El listado 7.7 vuelve a escribir el listado 7.6, esta vez utilizando un ciclo `do...while`.

**ENTRADA****LISTADO 7.7** Muestra del ciclo `do...while`

```
1:      // Listado 7.7
2:      // Muestra del ciclo do...while
3:
4:      #include <iostream.h>
5:
6:      int main()
7:      {
8:          int contador;
9:
10:         cout << "¿Cuántos holas?: ";
11:         cin >> contador;
12:         do
13:         {
14:             cout << "Hola\n";
15:             contador--;
16:         } while (contador >0);
17:         cout << "Contador vale: " << contador << endl;
18:
19:     }
```

**SALIDA**

```
¿Cuántos holas?: 2
Hola
Hola
Contador vale: 0
```

**ANÁLISIS**

En la línea 10 se pide al usuario un valor de inicio, el cual se guarda en la variable de tipo entero contador. En el ciclo `do...while` se entra al cuerpo del ciclo antes de evaluar la condición y, por lo tanto, se garantiza que el cuerpo del ciclo se ejecute por lo menos una vez. En la línea 14 se imprime el mensaje, en la línea 15 se decremente el contador, y en la línea 16 se prueba la condición. Si la condición es verdadera (`true`), la ejecución salta hasta el principio del ciclo en la línea 14; de no ser así, se va hasta la línea 17.

Las instrucciones `continue` y `break` funcionan en el ciclo `do...while` exactamente igual que en el ciclo `while`. La única diferencia entre un ciclo `while` y un ciclo `do...while` es el momento en el que se prueba la condición.

**La instrucción `do...while`**

La sintaxis para la instrucción `do...while` es la siguiente:

```
do
    instrucción
while (condición);
```

Primero se ejecuta `instrucción`, y luego se evalúa `condición`. Si `condición` es verdadera, el ciclo se repite; de no ser así, el ciclo termina. En todo lo demás, las instrucciones y condiciones son idénticas a las del ciclo `while`.

**Ejemplo 1**

```
// contar hasta 10
int x = 0;
do
cout << "X: " << x++;
while (x < 10)
```

**Ejemplo 2**

```
// imprimir alfabeto en minúsculas.
char ch = 'a';
do
{
cout << ch << ' ';
ch++;
} while ( ch <= 'z' );
```

**DEBE****No DEBE**

**DEBE** utilizar do...while cuando quiera asegurar que el ciclo se ejecute por lo menos una vez.

**DEBE** utilizar ciclos while cuando quiera saltar el ciclo si la condición es falsa.

**DEBE** probar todos los ciclos para asegurarse de que hagan lo que usted espera.

## Ciclos for

Al programar con ciclos while, a menudo se encontrará estableciendo una condición de inicio, evaluando si la condición es verdadera, e incrementando o cambiando una variable cada vez que pase por el ciclo. El listado 7.8 muestra esto.

**ENTRADA****LISTADO 7.8** while examinado nuevamente

```

1: // Listado 7.8
2: // Uso de ciclos con while
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:     int contador = 0;
9:
10:    while(contador < 5)
11:    {
12:        contador++;
13:        cout << "¡Haciendo un ciclo!  ";
14:    }
15:    cout << "\nContador: " << contador << ".\n";
16:    return 0;
17: }
```

**SALIDA**

```

¡Haciendo un ciclo! ¡Haciendo un ciclo! ¡Haciendo un ciclo!
¡Haciendo un ciclo! ¡Haciendo un ciclo!
Contador: 5.
```

**ANÁLISIS**

La asignación se establece en la línea 8: contador se inicializa en 0. En la línea 10 se prueba contador para ver si es menor que 5. En la línea 12 se incrementa contador. En la línea 13 se imprime un mensaje sencillo, pero podemos imaginar que se podrían realizar trabajos más importantes para cada incremento del contador. Finalmente, en la línea 15 se imprime el valor de contador cuando ha terminado el ciclo.

Un ciclo `for` combina tres pasos en una instrucción. Los tres pasos son inicialización, prueba e incremento. Una instrucción `for` consiste en la palabra reservada `for` seguida de un par de paréntesis. Dentro de los paréntesis se encuentran tres instrucciones separadas con punto y coma.

La primera instrucción es la inicialización. Cualquier instrucción válida de C++ se puede colocar aquí, pero generalmente esto se utiliza para crear e inicializar una variable de conteo. La instrucción 2 es la condición, y cualquier expresión válida de C++ se puede utilizar aquí. Esto juega el mismo papel que la condición en el ciclo `while`. La instrucción 3 es la acción. Por lo general se incrementa o decrementa un valor, aunque se puede colocar aquí cualquier instrucción válida de C++. Observe que las instrucciones 1 y 3 pueden ser cualquier instrucción válida de C++, pero la instrucción 2 debe ser una expresión (una instrucción de C++ que regrese un valor). El listado 7.9 muestra el uso de un ciclo `for`.

**ENTRADA** **LISTADO 7.9** Muestra del ciclo `for`

```

1:      // Listado 7.9
2:      // Uso de ciclos con for
3:
4:      #include <iostream.h>
5:
6:      int main()
7:      {
8:          int contador;
9:
10:         for (contador = 0; contador < 5; contador++)
11:             cout << "¡Haciendo un ciclo! ";
12:             cout << "\nContador: " << contador << ".\n";
13:         return 0;
14:     }

```

**SALIDA**

```

¡Haciendo un ciclo! ¡Haciendo un ciclo! ¡Haciendo un ciclo!
¡Haciendo un ciclo! ¡Haciendo un ciclo!
Contador: 5.

```

**ANÁLISIS** La instrucción `for` de la línea 10 combina en una sola línea la inicialización de `contador`, la prueba para ver si es menor que 5, y su incremento. El cuerpo de la instrucción `for` se encuentra en la línea 11. Claro que aquí también se podría utilizar un bloque.
**La instrucción `for`**

La sintaxis para la instrucción `for` es la siguiente:

```

for (inicialización; condición; acción )
    instrucción;

```

La instrucción `inicialización` se utiliza para inicializar el estado de un contador, o para prepararse para el ciclo. `condición` es cualquier expresión válida de C++ y se evalúa cada

vez que se repite el ciclo. Si la prueba de la condición es verdadera, se ejecuta el cuerpo del ciclo `for` y luego se ejecuta la instrucción acción (por lo general se incrementa el contador).

#### Ejemplo 1

```
// imprimir Hola diez veces
for (int i = 0; i<10; i++)
    cout << "¡Hola!"
```

#### Ejemplo 2

```
for (int i = 0; i < 10; i++)
{
    cout << "¡Hola!" << endl;
    cout << "el valor de i es: " << i << endl;
}
```

## Ciclos for avanzados

Las instrucciones `for` son poderosas y flexibles. Las tres instrucciones independientes (inicialización, condición y acción) se prestan a un número increíble de variaciones.

Un ciclo `for` funciona en la siguiente secuencia:

1. Realiza las operaciones de la inicialización
2. Evalúa la condición
3. Si la condición es verdadera, ejecuta el cuerpo del ciclo `for`; de no ser así, el ciclo termina
4. Ejecuta la instrucción de acción

Después de pasar cada vez por el cuerpo del ciclo, ocurre el paso 4 y el ciclo repite los pasos 2 y 3.

## Inicialización e incrementos múltiples

Es común inicializar más de una variable para probar una expresión lógica compuesta y para ejecutar más de una instrucción. La inicialización y la acción se pueden reemplazar por varias instrucciones de C++, cada una separada por una coma. El listado 7.10 muestra la inicialización y el incremento de dos variables.

### ENTRADA

### LISTADO 7.10 Muestra de varias instrucciones en el ciclo for

```
1: //listado 7.10
2: // muestra de varias instrucciones en
3: // ciclos for
```

```

4:
5: #include <iostream.h>
6:
7: int main()
8: {
9:     for (int i=0, j=0; i<3; i++, j++)
10:        cout << "i: " << i << " j: " << j << endl;
11:    return 0;
12: }

```

**SALIDA**

```

i: 0  j: 0
i: 1  j: 1
i: 2  j: 2

```

**ANÁLISIS** En la línea 9 se inicializan dos variables, *i* y *j*, cada una con el valor 0. Se evalúa la condición (*i*<3), y como es verdadera, se ejecuta el cuerpo de la instrucción **for** y se imprimen los valores. Finalmente, se ejecuta la tercera cláusula de la instrucción **for** y se incrementan *i* y *j*.

Después de que termina la línea 10, se evalúa otra vez la condición, y si aún es verdadera, se repiten las acciones (se incrementan de nuevo *i* y *j*) y se ejecuta otra vez el cuerpo del ciclo. Esto continúa hasta que falle la prueba, en cuyo caso no se ejecuta la instrucción acción (el cuerpo del ciclo, en este caso **cout**), y el control sale del ciclo.

### Instrucciones nulas en ciclos for

Cualquiera o todas las instrucciones del encabezado de un ciclo **for** pueden ser nulas. Para lograr esto, utilice el punto y coma (:) para indicar el lugar en el que debería estar la instrucción. Para crear un ciclo **for** que actúe exactamente como un ciclo **while**, omita la primera y tercera instrucciones. El listado 7.11 ilustra esta idea.

**ENTRADA** **LISTADO 7.11** Instrucciones nulas en ciclos for

```

1: // Listado 7.11
2: // Ciclos for con instrucciones nulas
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:     int contador = 0;
9:
10:    for( ; contador < 5; )
11:    {
12:        contador++;

```

**LISTADO 7.11** CONTINUACIÓN

```

13:           cout << "¡Haciendo un ciclo! ";
14:       }
15:       cout << "\nContador: " << contador << ".\n";
16:   return 0;
17: }
```

**SALIDA**

¡Haciendo un ciclo! ¡Haciendo un ciclo! ¡Haciendo un ciclo!  
 ¡Haciendo un ciclo! ¡Haciendo un ciclo!  
 Contador: 5.

**ANÁLISIS**

Observe que este código es idéntico al ciclo `while` que se muestra en el listado 7.8. En la línea 8 se inicializa la variable `contador`. La instrucción `for` de la línea 10 no inicializa ningún valor, pero incluye la condición `contador < 5`. No hay instrucción de incremento, por lo que este ciclo se comporta exactamente como si se hubiera escrito de la siguiente manera:

```
while (contador < 5)
```

Una vez más, C++ le proporciona varias maneras de lograr lo mismo. Ningún programador de C++ experimentado utilizaría un ciclo `for` de esta manera, pero esto muestra la flexibilidad de la instrucción `for`. De hecho, es posible, mediante el uso de `break` y `continue`, crear un ciclo `for` que no tenga ninguna de las tres instrucciones. El listado 7.12 muestra cómo.

**ENTRADA****LISTADO 7.12** Ejemplo de una instrucción de ciclo `for` con encabezado vacío

```

1: //Listado 7.12 muestra de una
2: //instrucción de ciclo for con encabezado vacío
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:     int contador=0;          // inicialización
9:     int max;
10:
11:    cout << "¿Cuántos holas?: ";
12:    cin >> max;
13:    for (;;)                 // un ciclo for que no termina
14:    {
15:        if (contador < max)      // prueba
16:        {
17:            cout << "¡Hola!\n";
18:            contador++;          // incremento
19:        }
20:        else
21:            break;
```

---

```

22:         }
23:     return 0;
24: }
```

---

**SALIDA**

¿Cuántos holas?: 3  
¡Hola!  
¡Hola!  
¡Hola!

**ANÁLISIS**

Ahora, el ciclo `for` ha sido llevado a su límite absoluto. **inicialización**, **condición** y **acción** se han omitido en la instrucción `for`. La inicialización se lleva a cabo en la línea 8, antes de que inicie el ciclo `for`. En la línea 15 se realiza la prueba en una instrucción `if` separada, y si la prueba tiene éxito, la acción, un incremento a contador, se realiza en la línea 18. Si la prueba falla, la salida del ciclo ocurre en la línea 21.

Aunque este programa en especial es un poco absurdo, algunas veces un ciclo `for(;;)` o un ciclo `while(true)` es justo lo que queremos. Un ejemplo de un uso más razonable de tales ciclos se presentará más adelante en este día, cuando hablaremos sobre las instrucciones `switch`.

## Ciclos for vacíos

Debido a que se pueden hacer muchas cosas en el encabezado de una instrucción `for`, algunas veces no será necesario que el cuerpo haga algo. En este caso, asegúrese de colocar una instrucción nula (`:`) como cuerpo del ciclo. El punto y coma puede estar en la misma línea que el encabezado, pero esto se puede confundir y se puede pasar por alto. El listado 7.13 muestra cómo utilizar un cuerpo nulo en un ciclo `for`.

**ENTRADA****LISTADO 7.13** Muestra de la instrucción nula en un ciclo for

---

```

1: //Listado 7.13
2: //Muestra de la instrucción nula
3: // como cuerpo de un ciclo for
4:
5: #include <iostream.h>
6: int main()
7: {
8:     for (int i = 0; i<5; cout << "i: " << i++ << endl)
9:     ;
10:    return 0;
11: }
```

---

**SALIDA**

i: 0  
i: 1  
i: 2  
i: 3  
i: 4

**ANÁLISIS** El ciclo **for** de la línea 8 incluye tres instrucciones: la instrucción **inicialización** establece el contador **i** y lo inicializa en **0**. La instrucción **condición** prueba si **i<5**, y la instrucción **acción** imprime el valor de **i** y lo incrementa.

No queda nada por hacer en el cuerpo del ciclo **for**, por lo que se utiliza la instrucción nula (**:**). Observe que éste no es un ciclo **for** bien diseñado: la instrucción **acción** está haciendo demasiado. Esto quedaría mejor si se escribiera de la siguiente manera:

```
8:     for (int i = 0; i<5; i++)
9:         cout << "i: " << i << endl;
```

Aunque ambos hacen lo mismo, este ejemplo es más fácil de comprender.

## Ciclos anidados

Los ciclos pueden estar anidados, un ciclo dentro del cuerpo de otro. El ciclo interno se ejecutará por completo en cada ejecución del ciclo externo. Con base en la experiencia, parece que el ciclo **for** es el tipo de ciclo que se anida con más frecuencia. Cualquiera de los tres ciclos se puede anidar, pero el ciclo **while** y el ciclo **do...while** son simplemente menos comunes. El listado 7.14 muestra la escritura de marcas en una matriz usando ciclos **for** anidados.

---

**ENTRADA** **LISTADO 7.14** Muestra de ciclos **for** anidados

---

```
1: //Listado 7.14
2: //Muestra de ciclos for anidados
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:     int filas, columnas;
9:     char elCaracter;
10:
11:    cout << "¿Cuántas filas? ";
12:    cin >> filas;
13:    cout << "¿Cuántas columnas? ";
14:    cin >> columnas;
15:    cout << "¿Cuál carácter? ";
16:    cin >> elCaracter;
17:    for (int i = 0; i<filas; i++)
18:    {
19:        for (int j = 0; j<columnas; j++)
20:            cout << elCaracter;
21:            cout << "\n";
22:    }
23:    return 0;
24: }
```

---

**SALIDA**

```
¿Cuántas filas? 4
¿Cuántas columnas? 12
¿Cuál carácter? x
xxxxxxxxxxxx
xxxxxxxxxxxx
xxxxxxxxxxxx
xxxxxxxxxxxx
```

**ANÁLISIS**

Se pide al usuario el número de filas y columnas, así como un carácter a imprimir. El primer ciclo **for**, que se encuentra en la línea 17, inicializa un contador (*i*) en 0, se comprueba la condición y luego se ejecuta el cuerpo del ciclo **for** externo.

En la línea 19 se establece la primera línea del cuerpo del ciclo **for** externo. Se inicializa también un segundo contador (*j*) en 0, se comprueba la condición, y se ejecuta el cuerpo del ciclo **for** interno. En la línea 20 se imprime el carácter elegido, y el control regresa al encabezado del ciclo **for** interno. Observe que el ciclo **for** interno consta de una sola instrucción (la impresión del carácter). La variable *j* se incrementa y se prueba la condición (*j < columnas*), si es verdadera (*true*), se imprime el siguiente carácter. Esto continúa hasta que *j* sea igual al número de columnas.

Al fallar la prueba del ciclo **for** interno, en este caso después de imprimir 12 letras X, la ejecución pasa a la línea 21 y se imprime una nueva línea. Ahora el ciclo **for** externo regresa a su encabezado, en donde *i* se incrementa y se prueba su condición (*i < filas*). Si es verdadera, se ejecuta el cuerpo del ciclo.

En la segunda iteración del ciclo **for** externo, el ciclo **for** interno empieza de nuevo. Por lo tanto, *j* se vuelve a inicializar en 0, se prueba la condición y se ejecuta otra vez el ciclo interno completo.

Lo importante aquí es que al utilizar un ciclo anidado, el ciclo interno se ejecuta por completo en cada iteración del ciclo externo. Por lo tanto, el carácter se imprime un número de veces igual al valor de columnas para cada fila.

**Nota**

Muchos programadores de C++ utilizan las letras *i* y *j* como contadores. Esta tradición tiene sus raíces en la notación matemática para establecer índices. Los científicos llevaron esa notación al lenguaje FORTRAN, en el que las letras *i*, *j*, *k*, *l*, *m* y *n* eran automáticamente enteros (incluso cualquier variable que empezara con las letras *i* hasta la *n*) y los enteros eran los únicos contadores válidos.

Otros programadores prefieren utilizar nombres de contadores más descriptivos, como *Cont1* y *Cont2*. Sin embargo, el uso de *i* y *j* en encabezados de ciclos **for** no debe ocasionar mucha confusión.

## Los ciclos for y su alcance

Anteriormente, las variables declaradas en el ciclo `for` tenían alcance en el bloque externo. El nuevo estándar ANSI cambia el alcance de estas variables a sólo el bloque del mismo ciclo `for`; sin embargo, no todos los compiladores soportan este cambio. Si usted utiliza un compilador que no sea de GNU, puede probarlo con el siguiente código:

```
#include <iostream.h>
int main()
{
    // ¿i tiene alcance sólo en el ciclo for?
    for (int i = 0; i<5; i++)
    {
        cout << "i: " << i << endl;
    }
    i = 7; // i no debería estar dentro de su alcance!
    return 0;
}
```

El compilador GNU emite una advertencia con este código; también puede obligarlo a ser estricto y que muestre un error. La advertencia se ve así:

```
example.cxx: In function 'int main()':
example.cxx:10: warning: name lookup of 'i' changed for new ANSI 'for' scoping
example.cxx:5: warning:   using obsolete binding at 'i'
```

Si esto compila sin problemas, su compilador (que no sea de GNU) no soporta aún este aspecto del estándar ANSI.

Si su compilador reclama que `i` aún no se define (en la línea `i=7`), entonces sí soporta el nuevo estándar. Puede escribir código que compile en cualquier compilador si cambia esto por lo siguiente:

```
#include <iostream.h>
int main()
{
    int i; //declarar fuera del ciclo for

    for (i = 0; i<5; i++)
    {
        cout << "i: " << i << endl;
    }

    i = 7; // ahora esto está dentro del alcance para todos los compiladores
    return 0;
}
```

## Resumen de los ciclos

En el día 5 aprendió cómo resolver el problema de la serie de Fibonacci por medio de la recursión. Para repasar esto brevemente, la serie de Fibonacci empieza con 1, 1, 2, 3, y todos los números subsecuentes son la suma de los dos anteriores:

1,1,2,3,5,8,13,21,34...

El enésimo número de Fibonacci es la suma de los números  $n-1$  y  $n-2$ . El problema resuelto en el día 5 fue encontrar el valor del enésimo número de Fibonacci. Esto se hizo mediante la recursión. El listado 7.15 ofrece una solución por medio de la iteración.

### ENTRADA LISTADO 7.15 Solución del enésimo número de Fibonacci por medio de la iteración

```
1: // Listado 7.15
2: // Muestra de la solución del enésimo
3: // número de Fibonacci por medio de la iteración
4:
5: #include <iostream.h>
6:
7: int fib(int posicion);
8:
9: int main()
10: {
11:     int respuesta, posicion;
12:
13:     cout << "¿Cuál posición?: ";
14:     cin >> posicion;
15:     cout << "\n";
16:     respuesta = fib(posicion);
17:     cout << respuesta << " es el número ";
18:     cout << posicion << " de la serie de Fibonacci.\n";
19:     return 0;
20: }
21:
22: int fib(int n)
23: {
24:     int menosDos=1, menosUno=1, respuesta=2;
25:
26:     if (n < 3)
27:         return 1;
28:     for (n -= 3; n; n--)
29:     {
30:         menosDos = menosUno;
```

**LISTADO 7.15** CONTINUACIÓN

```

31:         menosUno = respuesta;
32:         respuesta = menosUno + menosDos;
33:     }
34:     return respuesta;
35: }
```

**SALIDA**    ¿Cuál posición? 4  
              3 es el número 4 en la serie de Fibonacci.

Si se ejecuta el programa por segunda vez:

¿Cuál posición? 5  
5 es el número 5 de la serie de Fibonacci.

Si se ejecuta el programa otra vez:

¿Cuál posición? 20  
6765 es el número 20 de la serie de Fibonacci.

Si se vuelve a ejecutar el programa:

¿Cuál posición? 70  
885444751 es el número 70 de la serie de Fibonacci.

**ANÁLISIS**    El listado 7.15 soluciona la serie de Fibonacci por medio de la iteración en lugar de la recursión. Este método es más rápido y utiliza menos memoria que la solución por medio de la recursión.

En la línea 13 se pide al usuario la posición a verificar. Se hace una llamada a la función `fib()`, la cual evalúa la posición. Si la posición es menor que 3, la función regresa el valor 1. Empezando en la posición 3, la función itera usando el siguiente algoritmo:

1. Establecer la posición inicial: llenar variable `respuesta` con 2, `menosDos` con 1 y `menosUno` con 1. Decrementar la posición en 3, ya que los dos primeros números se manejan por la posición de inicio.
2. Para cada número, contar en orden ascendente en la serie de Fibonacci. Esto se hace así:
  - a. Colocar en `menosDos` el valor actual de `menosUno`.
  - b. Colocar en `menosUno` el valor actual de `respuesta`.
  - c. Sumar `menosUno` y `menosDos` y colocar la suma en `respuesta`.
  - d. Decrementar `n`.
3. Cuando `n` llegue a 0, salir del ciclo y regresar el valor de `respuesta`.

Ésta es exactamente la manera en que usted resolvería el problema con lápiz y papel. Si se le pidiera el quinto número de Fibonacci, tendría que escribir lo siguiente:

1, 1, 2,

y pensar: "Faltan dos". Luego tendría que sumar  $2 + 1$  y escribir 3, y pensar: "Falta encontrar uno". Por último, tendría que escribir  $3 + 2$  y la respuesta sería 5. En efecto, desvía su atención un número hacia la derecha cada vez que pasa por el ciclo, y decremente el número que queda por encontrar.

Observe la condición que se prueba en la línea 28 (`n`). Éste es un modismo de C++, y equivale a `n != 0`. Este ciclo `for` se basa en el hecho de que cuando `n` llegue a 0 será `false`, debido a que 0 tiene el valor `false` en C++. El encabezado del ciclo `for` se hubiera podido escribir de la siguiente manera:

```
for (n-=3; n!=0; n--)
```

lo que hubiera sido un poco más claro. Sin embargo, este modismo es tan común en C++ que no tiene mucho sentido evitarlo.

Compile, enlace y ejecute este programa, junto con la solución por medio de la recursión que se ofrece en el día 5. Trate de encontrar la posición 25 y compare el tiempo que tarda cada programa. La recursión es elegante, pero cada llamada a la función incrementa la pila, y como se llama muchas veces, su rendimiento es notablemente menor que el de la iteración y puede provocar una sobrecarga. Las microcomputadoras tienden a estar optimizadas para las operaciones aritméticas, por lo que la solución por medio de iteraciones debería ser mucho más rápida.

Tenga cuidado con el tamaño del número que escriba. `fib` crece rápidamente, y los enteros largos se desbordarán después de ciertos valores.

## Instrucciones switch

En el día 4 vio cómo escribir instrucciones `if` e `if/else`. Estas instrucciones se pueden volver algo confusas cuando se generan instrucciones `if` complejas, por lo que C++ ofrece una alternativa. A diferencia de la instrucción `if`, que evalúa un solo valor, las instrucciones `switch` le permiten ramificar la ejecución del programa con base en cualquiera de varios valores. La forma general de la instrucción `switch` es

```
switch (expresión)
{
    case valorUno: instrucción;
                    break;
    case valorDos: instrucción;
                    otra_instrucción;
                    break;
    ....
    case valorN:   instrucción;
                    break;
    default:       instrucción;
}
```

`expresión` es cualquier expresión válida de C++, y las instrucciones son cualquier instrucción o bloque de instrucciones válidas de C++ que se evalúe como (o que se

pueda convertir sin ambigüedad en) un valor entero. Sin embargo, observe que la evaluación es sólo para igualdad; tal vez no se puedan utilizar aquí los operadores relacionales, ni las operaciones booleanas.

Si uno de los valores de la cláusula case concuerda con la expresión, la ejecución salta hacia esas instrucciones y continúa hasta el fin del bloque switch, a menos que se encuentre una instrucción break. Si nada concuerda, la ejecución se ramifica hacia la cláusula predeterminada (default) opcional. Si no existe un valor predeterminado ni uno que concuerde, la ejecución sale de la instrucción switch y ésta termina.

### Nota

Casi siempre es una buena idea tener un caso predeterminado en las instrucciones switch. Si no tiene necesidad de utilizar el caso predeterminado, úselo para probar el caso supuestamente imposible, e imprima un mensaje de error; esto puede ser de gran ayuda en la depuración.

Es importante observar que si no hay una instrucción break al final de una cláusula case, la ejecución pasará a la siguiente cláusula case. Esto es necesario algunas veces, pero por lo general es un error. Si decide dejar que la ejecución pase por la siguiente cláusula case, asegúrese de poner un comentario que indique que no olvidó colocar la instrucción break.

El listado 7.16 muestra el uso de la instrucción switch.

#### ENTRADA

#### LISTADO 7.16 Muestra de la instrucción switch

```
1: //Listado 7.16
2: // Muestra de la instrucción switch
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:     unsigned short int numero;
9:
10:    cout << "Escriba un número entre 1 y 5: ";
11:    cin >> numero;
12:    switch (numero)
13:    {
14:        case 0: cout << "Demasiado pequeño, ¡lo siento!";
15:                  break;
16:        case 5: cout << "¡Buen trabajo!\n"; // la ejecución pasa a la
17:                  // siguiente cláusula case
18:        case 4: cout << "¡Buena elección!\n"; // la ejecución pasa a la
19:                  // siguiente cláusula case
20:        case 3: cout << "¡Excelente!\n"; // la ejecución pasa a la
21:                  // siguiente cláusula case
22:        case 2: cout << "¡Magistral!\n"; // la ejecución pasa a la
23:                  // siguiente cláusula case
```

```

20:     case 1: cout << "¡Increíble!\n";
21:         break;
22:     default: cout << "¡Demasiado grande!\n";
23:         break;
24:     }
25:     cout << "\n\n";
26:     return 0;
27: }
```

**SALIDA**

Escriba un número entre 1 y 5: 3  
 ¡Excelente!  
 ¡Magistral!  
 ¡Increíble!

Si se ejecuta el programa por segunda vez:

Escriba un número entre 1 y 5: 8  
 ¡Demasiado grande!

**ANÁLISIS**

Se pide un número al usuario. Ese número se le proporciona a la instrucción `switch`. Si el número es 0, la cláusula `case` de la línea 14 concuerda y se imprime el mensaje `Demasiado pequeño, ¡lo siento!`, y la instrucción `break` hace que termine la instrucción `switch`. Si el valor es 5, la ejecución pasa a la línea 16, en la que se imprime un mensaje, y luego pasa a la línea 17, en la que se imprime otro mensaje, y así sucesivamente, hasta llegar a la instrucción `break` de la línea 21.

El efecto neto de estas instrucciones es que para un número entre 1 y 5, se imprime esa cantidad de mensajes. Si el valor del número no está entre 0 y 5, se da por hecho que es demasiado grande, y se invoca a la cláusula predeterminada en la línea 22.

**La instrucción switch**

La sintaxis para la instrucción `switch` es la siguiente:

```

switch (expresión)
{
    case valorUno: instrucción;
    case valorDos: instrucción;
    ....
    case valorN: instrucción;
    default: instrucción;
}
```

La instrucción `switch` permite la ramificación con base en múltiples valores de expresión. Se evalúa la expresión, y si concuerda con alguno de los valores de las cláusulas `case`, la ejecución salta hasta esa línea. La ejecución continúa hasta el final de la instrucción `switch`, o hasta que encuentra una instrucción `break`.

Si la expresión no concuerda con ninguna de las cláusulas `case`, y si existe una cláusula predeterminada (`default`), la ejecución se dirige hacia ella; de no ser así, la instrucción `switch` termina.

**Ejemplo 1**

```
switch (opcion)
{
case 0:
    cout << "iCero!" << endl;
    break;
case 1:
    cout << "iUno!" << endl;
    break;
case 2:
    cout << "iDos!" << endl;
default:
    cout << "iPredeterminado!" << endl;
}
```

**Ejemplo 2**

```
switch (opcion)
{
case 0:
case 1:
case 2:
    cout << "iMenor que 3!";
    break;
case 3:
    cout << "iIgual a 3!";
    break;
default:
    cout << "iMayor que 3!";
}
```

## Uso de una instrucción switch con un menú

El listado 7.17 regresa al ciclo `for(;;)` que se describió anteriormente. Estos ciclos también se llaman ciclos eternos (`forever`), ya que se ejecutarán eternamente si no se encuentra una instrucción `break`. El ciclo eterno se utiliza para desplegar un menú, solicitar una opción del usuario, actuar sobre esa opción y luego regresar al menú. Esto continúa hasta que el usuario elige salir.

**Nota**

A algunos programadores les gusta escribir

```
#define SIEMPRE ;;
for (SIEMPRE)
{
    // instrucciones...
}
```

Un ciclo eterno es un ciclo que no tiene una condición de salida. Para poder salir del ciclo, se debe utilizar una instrucción break.

**ENTRADA LISTADO 7.17** Muestra de un ciclo eterno

```
1: //Listado 7.17
2: //Uso de un ciclo eterno para manejar
3: //la interacción con el usuario
4: #include <iostream.h>
5:
6: // prototipos
7: int menu();
8: void HacerTareaUno();
9: void HacerTareaMuchos(int);
10:
11: int main()
12: {
13:
14:     bool salir = false;
15:     for (;;)
16:     {
17:         int opcion = menu();
18:         switch(opcion)
19:         {
20:             case (1):
21:                 HacerTareaUno();
22:                 break;
23:             case (2):
24:                 HacerTareaMuchos(2);
25:                 break;
26:             case (3):
27:                 HacerTareaMuchos(3);
28:                 break;
29:             case (4):
30:                 continue; // iredundante!
31:                 break;
32:             case (5):
33:                 salir=true;
34:                 break;
35:             default:
36:                 cout << "¡Seleccione otra vez!\n";
37:                 break;
38:         }           // fin de switch
39:
40:         if (salir)
41:             break;
42:     }           // fin de ciclo eterno
43:     return 0;
```

**LISTADO 7.17** CONTINUACIÓN

```
44:     }                                // fin de main()
45:
46:     int menu()
47:     {
48:         int opcion;
49:
50:         cout << " **** Menú ****\n\n";
51:         cout << "(1) Opción uno.\n";
52:         cout << "(2) Opción dos.\n";
53:         cout << "(3) Opción tres.\n";
54:         cout << "(4) Volver a desplegar menú.\n";
55:         cout << "(5) Salir.\n\n";
56:         cout << ": ";
57:         cin >> opcion;
58:         return opcion;
59:     }
60:
61:     void HacerTareaUno()
62:     {
63:         cout << "iTarea Uno!\n";
64:     }
65:
66:     void HacerTareaMuchos(int cual)
67:     {
68:         if (cual == 2)
69:             cout << "iTarea Dos!\n";
70:         else
71:             cout << "iTarea Tres!\n";
72:     }
```

**SALIDA**

```
**** Menú ****
(1) Opción uno.
(2) Opción dos.
(3) Opción tres.
(4) Volver a desplegar menú.
(5) Salir.

: 1
iTarea Uno!
**** Menú ****
(1) Opción uno.
(2) Opción dos.
(3) Opción tres.
(4) Volver a desplegar menú.
(5) Salir.

: 3
```

```

iTarea Tres!
**** Menú ****
(1) Opción uno.
(2) Opción dos.
(3) Opción tres.
(4) Volver a desplegar menú.
(5) Salir.

: 5

```

**ANÁLISIS**

Este programa reúne varios conceptos vistos en este día y en días anteriores.

También muestra un uso común de la instrucción `switch`.

El ciclo eterno empieza en la línea 15. Se hace una llamada a la función `menu()`, la cual imprime el menú en la pantalla y regresa la selección del usuario. La instrucción `switch`, que empieza en la línea 18 y termina en la línea 38, actúa en base a la opción que el usuario elija.

Si el usuario escribe 1, la ejecución salta a la cláusula `case (1)`: de la línea 20. La línea 21 cambia la ejecución a la función `HacerTareaUno`, la cual imprime un mensaje y regresa. A su regreso, la ejecución continúa en la línea 22, en donde la instrucción `break` termina la instrucción `switch`, y la ejecución se va hasta la línea 39. En la línea 40 se evalúa la variable `salir`. Si es verdadera (`true`), se ejecutará la instrucción de la línea 41 y terminará el ciclo `for(;;)`; pero si es falsa (`false`), la ejecución continúa en la línea 15, en el principio del ciclo.

Observe que la instrucción `continue` de la línea 30 es redundante. Si se omitiera esta instrucción y se llegara a la instrucción `break` en la ejecución del programa, la instrucción `switch` terminaría, `salir` sería falsa, el ciclo volvería a iterar y se volvería a imprimir el menú. Sin embargo, la instrucción `continue` pasa por alto la prueba de `salir`.

| <b>DEBE</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | <b>No DEBE</b>                                                                                                                                                                                                |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>DEBE</b> utilizar instrucciones <code>switch</code> para evitar las instrucciones <code>if</code> complejas.</p> <p><b>DEBE</b> documentar cuidadosamente todos los casos intencionales en los que se pase de una cláusula <code>case</code> a otra hasta llegar al fin de la instrucción <code>switch</code> o encontrar una instrucción <code>break</code>.</p> <p><b>DEBE</b> colocar una cláusula <code>case</code> predeterminada en las instrucciones <code>switch</code>, aunque sea sólo para detectar situaciones que parezcan imposibles.</p> | <p><b>NO DEBE</b> olvidar poner una instrucción <code>break</code> al final de cada instrucción <code>case</code>, a menos que quiera que la ejecución pase a la siguiente instrucción <code>case</code>.</p> |

## Resumen

Existen formas diferentes para hacer que un programa de C++ realice un ciclo. El ciclo `while` comprueba una condición, y si es verdadera, ejecuta las instrucciones que se encuentran en el cuerpo del ciclo. Los ciclos `do...while` ejecutan el cuerpo del ciclo y luego prueban la condición. Los ciclos `for` inicializan un valor y luego prueban una condición. Si la condición es verdadera, se ejecuta el cuerpo del ciclo. Cada vez que se pase por el ciclo, se ejecuta la instrucción final del encabezado del ciclo `for`, se evalúa otra vez la expresión y se repite todo el proceso.

Por lo general, el uso de la instrucción `goto` se evita, debido a que produce un salto incondicional a una ubicación aparentemente arbitraria en el código, lo cual provoca que el código sea difícil de entender y de mantener. La instrucción `continue` ocasiona que los ciclos `while`, `do...while`, y `for` empiecen de nuevo, y la instrucción `break` ocasiona que las instrucciones `while`, `do...while`, `for`, y `switch` terminen.

## Preguntas y respuestas

- P ¿Cómo puedo elegir entre `if/else` y `switch`?**
- R** Si se utilizan dos o más cláusulas `else`, y todas evalúan el mismo valor, debe considerar el uso de una instrucción `switch`.
- P ¿Cómo puedo elegir entre `while` y `do...while`?**
- R** Si el cuerpo del ciclo se debe ejecutar por lo menos una vez, utilice un ciclo `do...while`; en caso contrario, trate de usar el ciclo `while`.
- P ¿Cómo puedo elegir entre `while` y `for`?**
- R** Si está inicializando una variable de conteo, y la evalúa y la incrementa cada vez que pasa por el ciclo, considere el uso del ciclo `for`. Si su variable ya está inicializada y no se incrementa en cada ciclo, tal vez el ciclo `while` sea la mejor opción.
- P ¿Cómo puedo elegir entre recursión e iteración?**
- R** Algunos problemas piden a gritos la recursión, pero la mayoría de los problemas también se puede resolver mediante la iteración. Mantenga la recursión bajo su manga, tal vez le sea útil algún día..
- P ¿Qué es mejor, usar `while (true)` o `for (;;)`?**
- R** No existe ninguna diferencia considerable.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate

de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, "Respuestas a los cuestionarios y ejercicios", y asegúrese de comprender las respuestas antes de pasar al siguiente día.

## Cuestionario

1. ¿Cómo puede inicializar más de una variable en un ciclo `for`?
2. ¿Por qué se evita el uso de la instrucción `goto`?
3. ¿Es posible escribir un ciclo `for` que tenga un cuerpo que nunca se ejecute?
4. ¿Es posible anidar ciclos `while` dentro de ciclos `for`?
5. ¿Es posible crear un ciclo que nunca termine? Dé un ejemplo.
6. ¿Qué pasa si crea un ciclo que nunca termine?

## Ejercicios

1. ¿Cuál es el valor de `x` cuando el siguiente ciclo `for` finaliza su ejecución?

```
for (int x = 0; x < 100; x++)
```

2. Escriba un ciclo `for` anidado que imprima ceros en un patrón de 10 x 10.

3. Escriba una instrucción `for` que cuente del 100 al 200 de dos en dos.

4. Escriba un ciclo `while` que cuente del 100 al 200 de dos en dos.

5. Escriba un ciclo `do...while` que cuente del 100 al 200 de dos en dos.

6. **CAZA ERRORES:** ¿Qué está mal en el siguiente código?

```
int contador = 0;
while (contador < 10)
{
    cout << "contador: " << contador;
}
```

7. **CAZA ERRORES:** ¿Qué está mal en el siguiente código?

```
for (int contador = 0; contador < 10; contador++);
    cout << contador << " ";
```

8. **CAZA ERRORES:** ¿Qué está mal en el siguiente código?

```
int contador = 100;
while (contador < 10)
{
    cout << "contador ahora: " << contador;
    contador--;
}
```

9. **CAZA ERRORES:** ¿Qué está mal en el siguiente código?

```
cout << "Escriba un número entre 0 y 5: ";
cin >> elNúmero;
```

```
switch (elNúmero)
{
    case 0:
        hacerCero();
    case 1:           // pasar a la siguiente cláusula case
    case 2:           // pasar a la siguiente cláusula case
    case 3:           // pasar a la siguiente cláusula case
    case 4:           // pasar a la siguiente cláusula case
    case 5:
        hacerUnoHastaCinco();
        break;
    default:
        hacerPredeterminado();
        break;
}
```

1

2

3

4

5

6

7

# SEMANA 1

## Repasso

Acaba de terminar la primera de tres semanas de aprendizaje de C++ para Linux. Debe estar orgulloso. Pero antes de que se relaje demasiado, hay más por hacer.

Dé un vistazo al listado R1.1 y piense en todo lo que ha aprendido durante esta primera semana. El código de C++ de este listado utiliza la mayoría de las técnicas que se cubrieron en esta semana.

**ENTRADA**
**LISTADO R1.1** Listado de repaso de la semana 1

```

1: #include <iostream.h>
2: // Archivo del listado lstr1-01.cxx
3: enum OPCION { DibujaRect = 1, ObtenArea,
4:     ObtenPerim, CambiaDimensiones, Salir};
5: // Declaración de clase Rectangulo
6: class Rectangulo
7: {
8:     public:
9:         // constructores
10:        Rectangulo(int anchura, int altura);
11:        ~Rectangulo();
12:
13:        // funciones de acceso
14:        int ObtenAltura() const { return suAltura; }
15:        int ObtenAnchura() const { return suAnchura; }
16:        int ObtenArea() const { return suAltura *
17:            suAnchura; }
18:        int ObtenPerim() const { return 2*suAltura +
19:            2*suAnchura; }
20:        void AsignaTamanio(int nuevaAnchura, int
21:            nuevaAltura);
22:
23:    private:
24:        int suAnchura;
25:        int suAltura;
26: };

```

*continúa*

**LISTADO R1.1** CONTINUACIÓN

```
27:  
28: // Implementaciones de los métodos de la clase  
29: void Rectangulo::AsignaTamanio(int nuevaAnchura, int nuevaAltura)  
30: {  
31:     suAnchura = nuevaAnchura;  
32:     suAltura = nuevaAltura;  
33: }  
34:  
35:  
36: Rectangulo::Rectangulo(int anchura, int altura)  
37: {  
38:     suAnchura = anchura;  
39:     suAltura = altura;  
40: }  
41:  
42: Rectangulo::~Rectangulo() {}  
43:  
44: int HacerMenu();  
45: void HacerDibujaRect(Rectangulo);  
46: void HacerDibujaArea(Rectangulo);  
47: void HacerDibujaPerim(Rectangulo);  
48:  
49: int main ()  
50: {  
51:     // inicializar un rectángulo con 30,5  
52:     Rectangulo elRect(30,5);  
53:  
54:     int opcion = DibujaRect;  
55:     int fSalir = false;  
56:  
57:     while (!fSalir)  
58:     {  
59:         opcion = HacerMenu();  
60:         if (opcion < DibujaRect || opcion > Salir)  
61:         {  
62:             cout << "\nOpción inválida, por favor intente de nuevo.\n\n";  
63:             continue;  
64:         }  
65:         switch (opcion)  
66:         {  
67:             case DibujaRect:  
68:                 HacerDibujaRect(elRect);  
69:                 break;  
70:             case ObtenArea:  
71:                 HacerDibujaArea(elRect);  
72:                 break;  
73:             case ObtenPerim:  
74:                 HacerDibujaPerim(elRect);
```

```
75:         break;
76:     case CambiaDimensiones:
77:         int nuevaLongitud, nuevaAnchura;
78:         cout << "\nNueva anchura: ";
79:         cin >> nuevaAnchura;
80:         cout << "Nueva altura: ";
81:         cin >> nuevaLongitud;
82:         elRect.AsignaTamanio(nuevaAnchura, nuevaLongitud);
83:         HacerDibujaRect(elRect);
84:         break;
85:     case Salir:
86:         fSalir = true;
87:         cout << "\nSaliendo...\n\n";
88:         break;
89:     default:
90:         cout << "¡Error en opción!\n";
91:         fSalir = true;
92:         break;
93:     } // fin de switch
94: } // fin de while
95: return 0;
96: } // fin de main
97:
98: int HacerMenu()
99: {
100:     int opcion;
101:     cout << "\n\n    *** Menú *** \n";
102:     cout << "(1) Dibujar rectángulo\n";
103:     cout << "(2) Área\n";
104:     cout << "(3) Perímetro\n";
105:     cout << "(4) Cambiar tamaño\n";
106:     cout << "(5) Salir\n";
107:
108:     cin >> opcion;
109:     return opcion;
110: }
111:
112: void HacerDibujaRect(Rectangulo elRect)
113: {
114:     int altura = elRect.ObtenAltura();
115:     int anchura = elRect.ObtenAnchura();
116:
117:     for (int i = 0; i<altura; i++)
118:     {
119:         for (int j = 0; j< anchura; j++)
120:             cout << "*";
121:         cout << "\n";
122:     }
123: }
124:
```

**LISTADO R1.1** CONTINUACIÓN

---

```
125:  
126: void HacerDibujaArea(Rectangulo elRect)  
127: {  
128:     cout << "Área: " << elRect.ObtenArea() << endl;  
129: }  
130:  
131: void HacerDibujaPerim(Rectangulo elRect)  
132: {  
133:     cout << "Perímetro: " << elRect.ObtenPerim() << endl;  
134: }
```

---

**SALIDA**

```
*** Menú ***  
(1) Dibujar rectángulo  
(2) Área  
(3) Perímetro  
(4) Cambiar tamaño  
(5) Salir  
1  
*****  
*****  
*****  
*****  
*****
```

```
*** Menú ***  
(1) Dibujar rectángulo  
(2) Área  
(3) Perímetro  
(4) Cambiar tamaño  
(5) Salir  
2  
Área: 150
```

```
*** Menú ***  
(1) Dibujar rectángulo  
(2) Área  
(3) Perímetro  
(4) Cambiar tamaño  
(5) Salir  
3  
Perímetro: 70
```

```
*** Menú ***  
(1) Dibujar rectángulo  
(2) Área  
(3) Perímetro  
(4) Cambiar tamaño  
(5) Salir
```

4

Nueva anchura: 10

Nueva altura: 8

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

\*\*\* Menú \*\*\*

- (1) Dibujar rectángulo
- (2) Área
- (3) Perímetro
- (4) Cambiar tamaño
- (5) Salir

2

Área: 80

\*\*\* Menú \*\*\*

- (1) Dibujar rectángulo
- (2) Área
- (3) Perímetro
- (4) Cambiar tamaño
- (5) Salir

3

Perímetro: 36

\*\*\* Menú \*\*\*

- (1) Dibujar rectángulo
- (2) Área
- (3) Perímetro
- (4) Cambiar tamaño
- (5) Salir

5

Saliendo...

**ANÁLISIS**

El programa mostrado en el listado R1.1 utiliza la mayoría de las habilidades que aprendió en esta semana. No sólo debe ser capaz de escribir, compilar, enlazar y ejecutar este programa, sino también de entender lo que hace y cómo funciona, con base en el trabajo que realizó esta semana.

Las primeras seis líneas establecen los nuevos tipos y las definiciones que serán utilizadas en todo el programa.

Las líneas 6 a 25 declaran la clase `Rectangulo`. Hay métodos públicos de acceso para obtener y asignar la anchura y la altura del rectángulo, así como para calcular el área y el perímetro. Las líneas 29 a 40 contienen las definiciones de las funciones de la clase que no fueron declaradas en línea.

Los prototipos de las funciones para las funciones que no son miembros de la clase se encuentran en las líneas 44 a 47, y el programa empieza en la línea 49. La esencia de este programa es generar un rectángulo y luego imprimir un menú en el que se ofrecen cinco opciones: dibujar el rectángulo, determinar su área, determinar su perímetro, cambiar de tamaño el rectángulo o salir.

Se establece un indicador en la línea 55, y cuando ese indicador tiene asignado el valor `false`, el ciclo del menú continúa. El indicador sólo tendrá el valor `true` si el usuario elige la opción Salir del menú.

Cada una de las otras opciones, con la excepción de `CambiaDimensiones`, llama a una función. Esto hace que la instrucción `switch` sea más limpia. `CambiaDimensiones` no puede llamar a una función porque debe cambiar las dimensiones del rectángulo. Si se pasara el rectángulo (por valor) a una función tal como `HacerCambiarDimensiones()`, las dimensiones se cambiarían en la copia local del rectángulo en `HacerCambiarDimensiones()`, y no en el rectángulo en `main()`. En el día 8, “Apuntadores,” y en el día 10, “Funciones avanzadas,” veremos cómo vencer esta restricción, pero por ahora el cambio se hace en la función `main()`.

Observe cómo el uso de una enumeración hace que el enunciado `switch` sea más limpio y fácil de comprender. Si la instrucción `switch` dependiera de las opciones numéricas (1–5) del usuario, tendríamos que referirnos constantemente a la descripción del menú para ver cuál opción era cuál.

En la línea 60 se comprueba la elección del usuario para asegurar que esté dentro del rango. Si no es así, se imprime un mensaje de error y se vuelve a imprimir el menú. Observe que la instrucción `switch` incluye una condición predeterminada “imposible”. Esto ayuda en la depuración. Si el programa está funcionando, esta instrucción nunca podrá ejecutarse.

## Repaso de la semana

¡Felicidades! ¡Acaba de completar la primera semana! Ahora puede crear y comprender programas sofisticados en C++. Desde luego que hay mucho más por hacer, y la siguiente semana inicia con uno de los conceptos más difíciles en C++: los apuntadores. No se dé por vencido ahora—está a punto de profundizar en el significado y el uso de la programación orientada a objetos, las funciones virtuales y muchas de las características avanzadas de este poderoso lenguaje.

Incluso aprendió cómo utilizar el compilador. ¡Ésta es una habilidad que utilizará en cada programa! Y después de que aprenda a utilizar un compilador, no tendrá dificultades con los demás (debido a que es el mismo proceso).

Tome un descanso, disfrute la gloria de su logro, y luego dé vuelta a la página para empezar con la semana 2.



# SEMANA 2

## De un vistazo

Ha terminado su primera semana en el proceso de aprendizaje de la programación en C++. Para estos momentos debe estar familiarizado con la escritura de programas, con el uso de su compilador y con los conceptos relacionados con los objetos, las clases y el flujo de programa.

## Objetivos

La semana 2 empieza con los apuntadores. Éste es un tema tradicionalmente difícil para los nuevos programadores de C++, pero su explicación será completa y clara, por lo que no debe considerarse como un obstáculo invencible. El día 8, “Apuntadores”, habla sobre los apuntadores, y el día 9, “Referencias”, enseña las referencias, que son un parente cercano de los apuntadores. En el día 10, “Funciones avanzadas”, veremos cómo sobrecargar funciones, y en el día 11, “Herencia”, hablaremos sobre la herencia, un concepto fundamental en la programación orientada a objetos. En el día 12, “Arreglos, cadenas tipo C y listas enlazadas”, aprenderá a trabajar con los arreglos, las cadenas y las colecciones. El día 13, “Polimorfismo”, extiende las lecciones del día 11 para hablar sobre el polimorfismo, y el día 14, “Clases y funciones especiales”, finaliza la semana con una explicación sobre las funciones estáticas y amigas.

8

9

10

11

12

13

14

# SEMANAS

## De tu amigo

En el mundo de las personas que tienen la responsabilidad de desarrollar y administrar sistemas de información, es importante recordar que el desarrollo de software es un proceso continuo y constante. Los sistemas se actualizan y evolucionan constantemente para adaptarse a los cambios en las necesidades de los usuarios. Es por eso que es fundamental tener una visión integral del desarrollo de software, que incluya tanto la parte técnica como la parte humana.

## Opciones

En este número de Semanas de tu amigo, queremos presentarte algunas de las principales opciones disponibles para el desarrollo de software. Una de las más populares es el desarrollo de software libre, que ofrece una alternativa gratuita y abierta a la innovación y la colaboración. Otra opción es el desarrollo de software empresarial, que se enfoca en la optimización de procesos y la mejora de la eficiencia.

Además, también existen opciones para el desarrollo de software móvil, que permite crear aplicaciones para dispositivos móviles como teléfonos celulares y tabletas. Estas aplicaciones pueden ser desarrolladas para diferentes plataformas, como iOS, Android y Windows Phone. Otra opción es el desarrollo de software web, que permite crear aplicaciones que se ejecutan en un navegador web y se acceden a través de Internet. Estas aplicaciones pueden ser desarrolladas para diferentes plataformas, como HTML5, CSS3 y JavaScript. Finalmente, también existen opciones para el desarrollo de software para la nube, que permite almacenar datos en la nube y acceder a ellos desde cualquier dispositivo con conexión a Internet.

# SEMANA 2

DÍA 8

## Apuntadores

Una de las herramientas más poderosas disponibles para un programador de C++ es la capacidad de manipular directamente la memoria mediante el uso de apuntadores. Hoy aprenderá lo siguiente:

- Qué son los apuntadores
- Cómo declarar y utilizar apuntadores
- Qué es el heap y cómo manipular memoria

Los apuntadores le plantean dos retos especiales si está aprendiendo a programar en C++. Pueden ser algo confusos, y no se aprecia de inmediato por qué se necesitan. Esta lección explica paso a paso cómo funcionan los apuntadores. Usted comprenderá completamente por qué se necesitan los apuntadores, a medida que lea el resto del libro.

### ¿Qué es un apuntador?

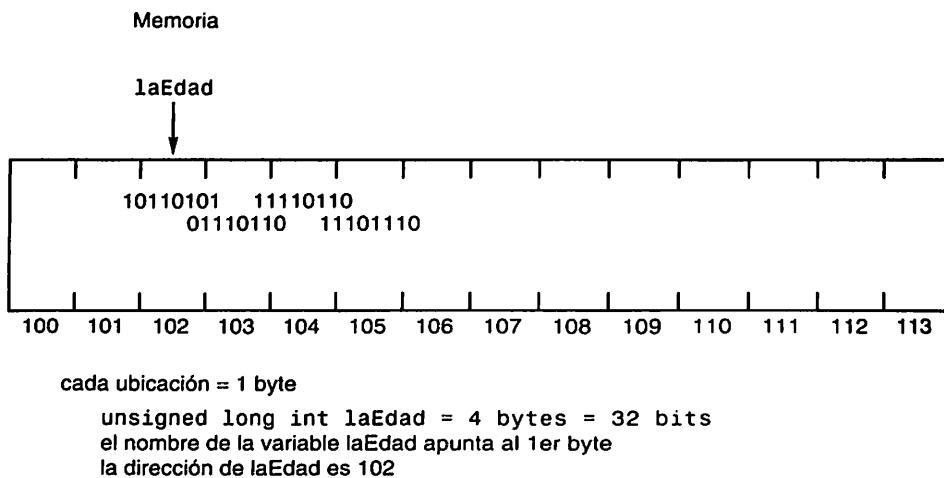
Un *apuntador* es una variable que guarda una dirección de memoria.

Para comprender lo que son los apuntadores, debe saber un poco acerca de la memoria de la computadora. La memoria se divide en ubicaciones de memoria numeradas en forma secuencial. Cada variable se encuentra en una ubicación

única en memoria, conocida como su dirección. La figura 8.1 muestra una representación esquemática del almacenamiento de una variable de tipo entero largo sin signo (`unsigned long int`) llamada `laEdad`.

**FIGURA 8.1**

*Una representación esquemática de laEdad.*



La numeración de la memoria varía entre computadoras, usando distintos esquemas complejos. Por lo general, los programadores no necesitan conocer la dirección específica de alguna variable dada, ya que el compilador se encarga de los detalles. No obstante, si usted quiere esta información, puede utilizar el operador de dirección (&), el cual se muestra en el listado 8.1.

### ENTRADA

### **LISTADO 8.1** Muestra del operador de dirección

```

1: // Listado 8.1 Muestra del operador de dirección,
2: // y de direcciones de variables locales
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:     unsigned short shortVar = 5;
9:     unsigned long longVar = 65535;
10:    long sVar = -65535;
11:
12:    cout << "Variable de tipo short sin signo:\t";
13:    cout << shortVar << "\n";
14:    cout << "Dirección de variable de tipo short:\t";
15:    cout << &shortVar << "\n";
16:
17:    cout << "Variable de tipo long sin signo:\t";
18:    cout << longVar << "\n";
19:    cout << "Dirección de variable de tipo long:\t" ;
20:    cout << &longVar << "\n";
21:
22:    cout << "Variable de tipo long con signo:\t";
23:    cout << sVar << "\n";

```

```

24:     cout << "Dirección de variable de tipo long con signo:\t" ;
25:     cout << &sVar << "\n";
26:
27:     return 0;
28: }
```

**SALIDA**

```

Variable de tipo short sin signo:      5
Dirección de variable de tipo short: 0xbfffffa16
Variable de tipo long sin signo:       65535
Dirección de variable de tipo long:   0xbfffffa10
Variable de tipo long con signo:      - 65535
Dirección de variable de tipo long con signo: 0xbfffffa0c
```

(La salida que usted obtenga puede ser distinta, dependiendo de la configuración de su sistema; algunos compiladores GNU mostrarán todas las direcciones como el valor 1 (uno). Tal vez necesite utilizar la función `printf()` con el especificador de formato %p en caso de que esto le ocurra.)

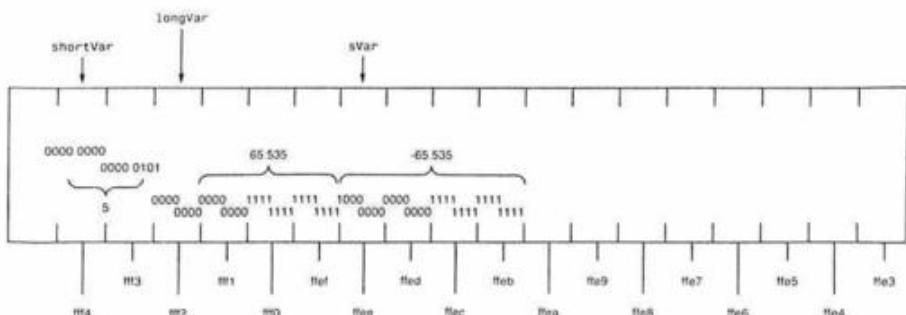
**ANÁLISIS**

Se declaran y se inicializan tres variables: una de tipo `unsigned short` en la línea 8, una de tipo `unsigned long` en la línea 9 y otra de tipo `long` en la línea 10. Sus valores y direcciones se imprimen en las líneas 12 a 25 mediante el uso del operador de dirección (&).

El valor de `shortVar` es 5, como se esperaba, y su dirección es 0xbfffffa16. Esta compleja dirección es específica de la computadora, del compilador y del sistema operativo, y puede cambiar ligeramente cada vez que el programa se ejecute. Los resultados que usted obtenga serán diferentes. Sin embargo, lo que no cambia es que la diferencia en las dos primeras direcciones es de 2 bytes, si su computadora utiliza enteros cortos de 2 bytes. La diferencia entre la segunda y la tercera es de 4 bytes, si su computadora utiliza enteros largos de 4 bytes. La figura 8.2 muestra cómo se guardarían en memoria las variables de este programa.

**FIGURA 8.2**

Ejemplo del almacenamiento de variables.



Usted no necesita conocer el valor numérico real de la dirección de cada variable. Lo que debe importarle es que cada una tiene una dirección y que se reserva la cantidad correcta de memoria. Al declarar el tipo de la variable, usted le indica al compilador cuánta memoria debe asignar para sus variables; el compilador asigna automáticamente una dirección para la variable. Por ejemplo, un entero largo por lo general tiene un tamaño de 4 bytes, lo que significa que la variable tiene la dirección del primer byte de los 4 bytes reservados en memoria.

No hay relación entre el tamaño de un tipo de datos y el tamaño del apuntador utilizado para apuntar a ese tipo de datos. Que un entero largo y un apuntador a ese entero largo sean de 4 bytes, es sólo una coincidencia. Usted no debe hacer suposiciones en relación con el tamaño de un apuntador, y no debería tratar de guardar una dirección en algo que no sea un apuntador (no trate de guardar una dirección en una variable de tipo `unsigned long int`).

## Cómo guardar la dirección en un apuntador

Cada variable tiene una dirección. Aunque no conozca la dirección específica de una variable dada, puede guardar esa dirección en un apuntador.

Por ejemplo, suponga que `queTanViejo` es una variable de tipo entero. Para declarar un apuntador llamado `apEdad` para que guarde la dirección de esta variable, tendría que escribir lo siguiente:

```
int * apEdad = NULL;
```

Esto declara `apEdad` como un apuntador a `int`. Es decir, `apEdad` se declara para guardar la dirección de un `int`.

Observe que `apEdad` es una variable como cualquier otra. Cuando se declara una variable de tipo entero, se prepara para que guarde un entero. Cuando se declara una variable de apuntador como `apEdad`, está preparada para guardar una dirección. `apEdad` es sólo un tipo distinto de variable.

En este ejemplo, `apEdad` se inicializa con la dirección constante `NULL`. Un apuntador cuyo valor sea `NULL` se conoce como *apuntador nulo*. Todos los apuntadores, al ser creados, deben ser inicializados con algo. Si usted no sabe qué quiere asignar al apuntador, asígnele `NULL`. Un apuntador que no está inicializado se conoce como *apuntador perdido*. Este tipo de apuntadores es muy peligroso.

### Nota

Practique la programación segura: ¡Inicialice sus apuntadores! A algunos programadores les gusta utilizar el valor 0 (cero) para inicializar sus apuntadores. Lo mejor es utilizar la constante `NULL`.

Si inicializa el apuntador con `NULL`, debe asignar específicamente la dirección de `queTanViejo` a `apEdad`. El siguiente ejemplo muestra cómo hacer esto:

```
unsigned short int queTanViejo = 50; // crear una variable
unsigned short int * apEdad = NULL; // crear un apuntador
apEdad = &queTanViejo; // colocar la dirección de queTanViejo
                      // en apEdad
```

La primera línea crea una variable (`queTanViejo`, del tipo `unsigned short int`) y la inicializa con el valor 50. La segunda línea declara `apEdad` como un apuntador al tipo `unsigned short int` y lo inicializa con `NULL`. Usted sabe que `apEdad` es un apuntador por el asterisco (\*) que va después del tipo de variable y antes del nombre de la variable.

La tercera y última línea asigna la dirección de queTanViejo al apuntador apEdad. Puede ver que se está asignando la dirección de queTanViejo gracias al operador de dirección (&). Si este operador no se hubiera utilizado, se habría asignado el valor de queTanViejo en lugar de su dirección. Ésta podría ser o no una dirección válida.

En este punto, apEdad tiene como valor la dirección de queTanViejo. queTanViejo, a su vez, tiene el valor 50. Usted hubiera podido lograr esto con un solo paso, como se muestra a continuación:

```
unsigned short int queTanViejo = 50;           // crear una variable  
unsigned short int * apEdad = &queTanViejo; // crear un apuntador a  
queTanViejo
```

apEdad es un apuntador que ahora contiene la dirección de la variable queTanViejo. Por medio de apEdad puede determinar el valor de queTanViejo, que en este caso es 50. Acceder a queTanViejo mediante el uso del apuntador apEdad se conoce como indirección, ya que usted está accediendo indirectamente a queTanViejo por medio de apEdad. Más adelante en este día verá cómo utilizar la indirección para tener acceso al valor de una variable.

*Indirección* significa acceder al valor de una variable cuya dirección está guardada en un apuntador. El apuntador proporciona una manera indirecta de obtener el valor que se guarda en esa dirección.

## Elección de nombres de apuntadores

Los apuntadores pueden tener cualquier nombre que sea válido para otras variables. Muchos programadores siguen la convención de nombrar a todos los apuntadores con una p o con ap al principio (ap es una contracción de “apuntador”, mientras que p proviene de “pointer”, que en inglés significa apuntador), como en pEdad, pNúmero, apEdad o apNúmero. En este libro utilizaremos la forma apNombreVariable para designar a los apuntadores.

## Uso del operador de indirección

El operador de indirección (\*) también se conoce como *operador de desreferencia*. Cuando un apuntador es desreferenciado, se recupera el valor que se encuentra en la dirección guardada por el apuntador.

Las variables normales proporcionan un acceso directo a sus propios valores. Si usted crea una nueva variable de tipo unsigned short int llamada suEdad y quiere asignarle el valor de queTanViejo, debe escribir lo siguiente:

```
unsigned short int suEdad;  
suEdad = queTanViejo;
```

Un apuntador proporciona un acceso indirecto al valor de la variable cuya dirección se encuentra almacenada en otra variable. Para asignar el valor contenido en queTanViejo a la nueva variable suEdad por medio del apuntador apEdad, debe escribir lo siguiente:

```
unsigned short int suEdad;  
suEdad = *apEdad;
```

El operador de indirección (\*) que está antes de la variable `apEdad` significa "el valor guardado en". Esta asignación dice: "tomar el valor guardado en la dirección de `apEdad` y asignarlo a `suEdad`".

### Nota

El operador de indirección (\*) se utiliza de dos maneras distintas con los apuntadores: declaración y desreferencia. Cuando se declara un apuntador, el asterisco indica que es un apuntador, no una variable normal. Por ejemplo,

```
unsigned short * apEdad = NULL; // crear un apuntador a un entero corto sin signo
```

Cuando el apuntador es dereferenciado, el operador de indirección indica que se debe acceder al valor que se encuentra en la dirección de memoria guardada en el apuntador, y no a la dirección en sí.

```
*apEdad = 5; // asignar 5 al valor que se encuentra en la dirección a la que apunta apEdad
```

Observe también que este mismo carácter (\*) se utiliza como operador de multiplicación. El compilador sabe a cuál operador llamar, basándose en el contexto (se dice que estos lenguajes tienen una gramática sensible al contexto).

## Apuntadores, direcciones y variables

Es importante distinguir entre un apuntador, la dirección que guarda el apuntador y el valor que se encuentra en la dirección guardada por el apuntador. Esto es lo que provoca la mayor parte de la confusión acerca de los apuntadores.

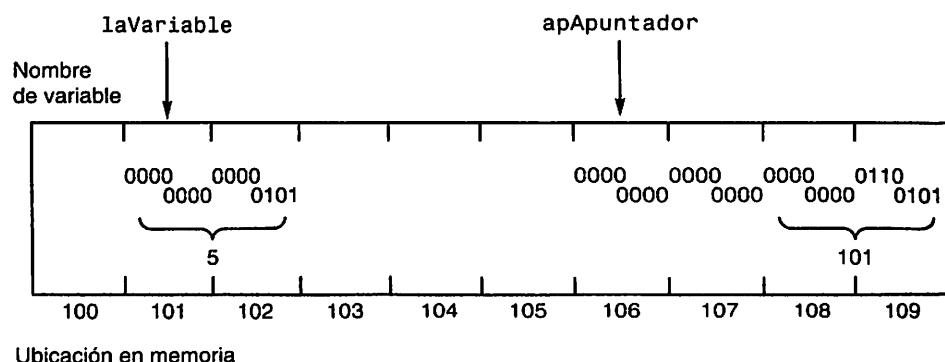
Considere el siguiente fragmento de código:

```
int laVariable = 5;
int * apApuntador = &laVariable ;
```

`laVariable` se declara como variable de tipo entero inicializada con el valor 5. `apApuntador` se declara como apuntador a un entero, y se inicializa con la dirección de `laVariable`. `apApuntador` es el apuntador. La dirección que guarda `apApuntador` es la dirección de `laVariable`. El valor que se encuentra en la dirección guardada por `apApuntador` es 5. La figura 8.3 muestra una representación esquemática de `laVariable` y `apApuntador`.

**FIGURA 8.3**

Una representación esquemática de la memoria.



## Manipulación de datos mediante el uso de apuntadores

Después de asignar la dirección de una variable a un apuntador, puede utilizar ese apuntador para tener acceso a los datos guardados en esa variable. El listado 8.2 muestra cómo se asigna la dirección de una variable local a un apuntador, y cómo manipula el apuntador los valores que se guardan en esa variable.

### ENTRADA LISTADO 8.2 Manipulación de datos mediante el uso de apuntadores

```
1: // Listado 8.2 Uso de apuntadores
2:
3: #include <iostream.h>
4:
5: typedef unsigned short int USHORT;
6: int main()
7: {
8:     USHORT miEdad; // una variable
9:     USHORT * apEdad = NULL; // un apuntador
10:
11:    miEdad = 5;
12:    cout << "miEdad: " << miEdad << "\n";
13:    // asignar dirección de miEdad a apEdad
14:    apEdad = &miEdad;
15:    cout << "*apEdad: " << *apEdad << "\n\n";
16:    cout << "*apEdad = 7\n";
17:    // asigna el valor 7 a miEdad
18:    *apEdad = 7;
19:    cout << "*apEdad: " << *apEdad << "\n";
20:    cout << "miEdad: " << miEdad << "\n\n";
21:    cout << "miEdad = 9\n";
22:    miEdad = 9;
23:    cout << "miEdad: " << miEdad << "\n";
24:    cout << "*apEdad: " << *apEdad << "\n";
25:
26:    return 0;
27: }
```

### SALIDA

```
miEdad: 5
*apEdad: 5
```

```
*apEdad = 7
*apEdad: 7
miEdad: 7

miEdad = 9
miEdad: 9
*apEdad: 9
```

### ANÁLISIS

Este programa declara dos variables: una de tipo `unsigned short`, `miEdad`, y un apuntador a un `unsigned short`, `apEdad`. En la línea 11 se asigna el valor 5 a `miEdad`; esto se verifica por medio de la impresión en la línea 12.

En la línea 14 se asigna la dirección de `miEdad` a `apEdad`. En la línea 15, `apEdad` es desreferenciado e impreso, lo que muestra que el valor que está en la dirección guardada por `apEdad` es el 5 guardado en `miEdad`. En la línea 18 se asigna el valor 7 a la variable que se encuentra en la dirección guardada por `apEdad`. Esto asigna un 7 a `miEdad`, lo que queda confirmado por las impresiones de las líneas 19 y 20.

En la línea 22 se asigna el valor 9 a la variable `miEdad`. Este valor se obtiene directamente en la línea 23 y se obtiene indirectamente (al desreferenciar a `apEdad`) en la línea 24.

## Cómo examinar una dirección

Los apuntadores le permiten manipular direcciones sin que necesite saber su valor real. Después de hoy, puede confiar ciegamente en que, cuando asigne la dirección de una variable a un apuntador, éste realmente tendrá como valor la dirección de esa variable. Pero sólo por esta vez, ¿por qué no comprobar para estar seguros? El listado 8.3 muestra esto.

**ENTRADA****LISTADO 8.3 Cómo averiguar lo que está guardado en un apuntador**

```
1: // Listado 8.3 Qué se guarda en un apuntador.
2:
3: #include <iostream.h>
4:
5: int main()
6: {
7:     unsigned short int miEdad = 5, suEdad = 10;
8:     unsigned short int * apEdad = &miEdad; // un apuntador
9:
10:    cout << "miEdad:\t" << miEdad
11:        << "\tsuEdad:\t" << suEdad << "\n";
12:    cout << "&miEdad:\t" << &miEdad
13:        << "\t&suEdad:\t" << &suEdad << "\n";
14:    cout << "apEdad:\t" << apEdad << "\n";
15:    cout << "*apEdad:\t" << *apEdad << "\n";
16:    // reasignar el apuntador
17:    apEdad = &suEdad;
18:    cout << "miEdad:\t" << miEdad
19:        << "\tsuEdad:\t" << suEdad << "\n";
20:    cout << "&miEdad:\t" << &miEdad
21:        << "\t&suEdad:\t" << &suEdad << "\n";
22:    cout << "apEdad:\t" << apEdad << "\n";
23:    cout << "*apEdad:\t" << *apEdad << "\n";
24:    cout << "&apEdad:\t" << &apEdad << "\n";
25:    return 0;
26: }
```

**SALIDA**

```

miEdad:      5          suEdad:    10
&miEdad:    0xbffffa16  &suEdad:   0xbffffa14
apEdad:     0xbffffa16
*apEdad:    5
miEdad:      5          suEdad:    10
&miEdad:    0xbffffa16  &suEdad:   0xbffffa14
apEdad:     0xbffffa14
*apEdad:    10
&apEdad:   0xbffffa10

```

(Tal vez la salida que usted obtenga sea distinta.)

**ANÁLISIS** En la línea 7, `miEdad` y `suEdad` se declaran como variables de tipo entero corto sin signo (`unsigned short int`). En la línea 8, `apEdad` se declara como apuntador a un entero corto sin signo, y se inicializa con la dirección de la variable `miEdad`.

Las líneas 10 a 13 imprimen los valores y las direcciones de `miEdad` y `suEdad`. La línea 14 imprime el contenido de `apEdad`, que es la dirección de `miEdad`. La línea 14 imprime el resultado de desreferenciar a `apEdad`, lo que imprime el valor guardado en la dirección a la que apunta `apEdad` (el valor guardado en `miEdad`, es decir, 5).

Ésta es la esencia de los apuntadores. La línea 14 muestra que `apEdad` guarda la dirección de `miEdad`, y la línea 15 muestra cómo obtener el valor guardado en `miEdad` desreferenciando al apuntador `apEdad`. Asegúrese de entender esto completamente antes de seguir adelante. Estudie el código y examine la salida.

En la línea 17, `apEdad` se reasigna para apuntar a la dirección de `suEdad`. Los valores y las direcciones se imprimen de nuevo. La salida muestra que ahora `apEdad` tiene la dirección de la variable `suEdad` y que con esa desreferencia se obtiene el valor guardado en `suEdad`.

La línea 24 imprime la dirección del mismo apuntador `apEdad`. Como cualquier otra variable, tiene una dirección, y esa dirección se puede guardar en un apuntador. (En breve discutiremos la asignación de la dirección de un apuntador a otro apuntador.)

| DEBE                                                                                                                                 | NO DEBE |
|--------------------------------------------------------------------------------------------------------------------------------------|---------|
| <b>DEBE</b> utilizar el operador de indirección (*) para tener acceso a los datos guardados en la dirección que guarda un apuntador. |         |
| <b>DEBE</b> inicializar todos los apuntadores ya sea con una dirección válida o con NULL.                                            |         |
| <b>DEBE</b> recordar la diferencia entre la dirección que guarda un apuntador y el valor que se guarda en esa dirección.             |         |

### Uso de apuntadores

Para declarar un apuntador, escriba el tipo de la variable u objeto cuya dirección se va a guardar en el apuntador, seguido del operador de indirección (\*) y del nombre del apuntador. Por ejemplo,

```
unsigned short int * apApuntador = NULL;
```

Para inicializar o asignar un valor a un apuntador, coloque el operador de dirección (&) antes del nombre de la variable cuya dirección se va a asignar. Por ejemplo,

```
unsigned short int laVariable = 5;  
unsigned short int * apApuntador = & laVariable;
```

Para desreferenciar un apuntador, coloque el operador de indirección (\*) antes del nombre del apuntador. Por ejemplo,

```
unsigned short int elValor = *apApuntador
```

## ¿Por qué utilizar apuntadores?

Hasta ahora ha visto paso a paso los detalles de la asignación de la dirección de una variable a un apuntador. No obstante, en la práctica nunca haría esto. Después de todo, ¿para qué batallar con un apuntador cuando ya tiene una variable con acceso a ese valor? La única razón para utilizar este tipo de manipulación por medio de apuntadores de una variable automática es para mostrar la forma en que trabajan los apuntadores.

Ahora que está familiarizado con la sintaxis de los apuntadores, puede empezar a utilizarlos. Por lo general, los apuntadores se utilizan para tres cosas:

- Manejar datos en el heap
- Tener acceso a los datos miembro y a las funciones de las clases
- Pasar variables por referencia a las funciones

El resto de esta lección se enfoca en el manejo de datos en el heap y en el acceso de los datos miembro y las funciones de las clases. Mañana aprenderá cómo pasar variables por referencia.

## La pila y el heap

En el día 5, “Funciones”, en la sección “Cómo trabajan las funciones: un vistazo a su interior”, se mencionan cinco áreas de memoria:

- Espacio de nombres global
- Heap

- Registros
- Espacio de código
- Pila

Las variables locales se encuentran en la pila, junto con los parámetros de funciones. El código se encuentra, desde luego, en el espacio de código, y las variables globales se encuentran en el espacio de nombres global. Los registros se utilizan para el mantenimiento interno de las funciones, como llevar el registro de la parte superior de la pila y del apuntador de instrucciones. Casi toda la memoria restante se pasa al heap.

El problema con las variables locales es que no duran; cuando alguna función se ejecuta, los datos y las variables se colocan en la pila, y cuando la función termina, las variables locales desaparecen. Las variables globales solucionan ese problema, pero la desventaja es que ofrecen un acceso sin restricciones en todo el programa, lo que conduce a la creación de código difícil de entender y de mantener. Al colocar los datos en el heap se solucionan ambos problemas.

Imagine que el heap es una sección masiva de memoria en la que miles de casillas numeradas en forma secuencial permanecen en espera de sus datos. Lo malo es que no puede etiquetar estas casillas, como se puede hacer con la pila. Tiene que pedir la dirección de la casilla que va a reservar y luego guardar esa dirección en un apuntador.

Una forma de visualizar esto es con una analogía: Un amigo le proporciona el número 800 de la compañía Acme de pedidos por correspondencia. Usted va a su casa y programa su teléfono con ese número, luego tira el pedazo de papel que tiene anotado el número telefónico. Si oprime el botón, un teléfono timbra en alguna parte, y contesta el servicio de pedidos por correspondencia de la compañía Acme. Este servicio representa los datos en el heap. Usted no sabe donde está, pero sabe cómo llegar a él. Accede a él usando su dirección (en este caso, el número telefónico). No tiene que conocer ese número, sólo tiene que colocarlo en un apuntador (el botón). El apuntador le da el acceso a sus datos sin molestarlo con los detalles.

La pila se limpia automáticamente cuando una función termina. Todas las variables locales quedan fuera de alcance, y se eliminan de la pila. El heap se limpia hasta que termina el programa, y es responsabilidad de usted liberar cualquier memoria que haya reservado, cuando ya no la utilice. Como se habrá dado cuenta, tiene que reservar la memoria antes de poder utilizar variables en el heap (apuntadores). Aunque muchos programas y compiladores le permitirán utilizar código que no reserve espacio explícitamente, dicho código puede generar errores muy difíciles de depurar.

La ventaja de usar el heap es que la memoria que usted reserve estará disponible hasta que la libere explícitamente. Si reserva memoria en el heap mientras se encuentra en una función, la memoria todavía estará disponible cuando la función termine.

La ventaja de acceder a la memoria de esta manera, en lugar de utilizar variables globales, es que sólo las funciones que tengan acceso al apuntador tendrán acceso a los datos. Esto proporciona una interfaz estrechamente controlada para esos datos, y elimina el problema de que una función cambie esos datos en forma inesperada.

Para que esto funcione, usted debe ser capaz de crear un apuntador hacia un área en el heap, y debe pasar ese apuntador entre las distintas funciones. Las siguientes secciones describen cómo hacer esto.

### **new**

En C++ se utiliza la palabra reservada `new` para asignar memoria en el heap. Después de `new` debe ir el tipo del objeto que quiere asignar, para que el compilador sepa cuánta memoria se requiere. Por lo tanto, `new unsigned short int` asigna 2 bytes en el heap, y `new long` asigna 4.

El valor de retorno de `new` es una dirección de memoria, la cual se debe asignar a un apuntador. Para crear un tipo `unsigned short` en el heap, podría escribir lo siguiente:

```
unsigned short int * apApuntador;  
apApuntador = new unsigned short int;
```

Puede, desde luego, al crear el apuntador, inicializarlo con lo siguiente:

```
unsigned short int * apApuntador = new unsigned short int;
```

En cualquier caso, ahora `apApuntador` apunta a un tipo `unsigned short int` en el heap. Puede utilizar este apuntador como cualquier otro que apunte a una variable, y puede asignar un valor en esa área de memoria si escribe lo siguiente:

```
*apApuntador = 72;
```

Esto significa: “Colocar 72 en el valor de `apApuntador`” o “Asignar el valor 72 al área del heap a la que apunta `apApuntador`”.

Si `new` no puede crear memoria en el heap (la memoria es, después de todo, un recurso limitado), se producirá una excepción (vea el día 20, “Excepciones y manejo de errores”).

### **delete**

Al terminar de utilizar su área de memoria, debe llamar a `delete` para que actúe sobre el apuntador. `delete` libera la memoria reservada. Recuerde que el apuntador mismo (a diferencia de la memoria a la que apunta) es una variable local. Cuando la función en la que está declarado termina, ese apuntador queda fuera de alcance y se pierde. La memoria asignada con `new` no se libera automáticamente. Esa memoria ya no está disponible porque no hay manera de referenciarla (no tenemos idea de dónde está); esta situación se conoce como *fuga de memoria*. Se llama así porque esa memoria sólo se puede recuperar hasta que termina el programa. Es como si la memoria se hubiera fugado de su computadora.

Para regresar la memoria al heap, se utiliza la palabra reservada `delete`. Por ejemplo,

```
delete apApuntador;
```

Al eliminar el apuntador, lo que realmente está haciendo es liberar la memoria cuya dirección está guardada en el apuntador. Está diciendo: "Regresar al heap la memoria a la que apunta este apuntador". El apuntador todavía es un apuntador, y se puede asignar a otra dirección. El listado 8.4 muestra la asignación de una variable al heap, el uso de esa variable y su eliminación.

8

### Precaución

Al utilizar `delete` en un apuntador, se libera la memoria a la que éste apunta. ¡Si vuelve a utilizar `delete` en ese apuntador, el programa dejará de funcionar! Cuando elimine un apuntador, asignele el valor `NULL`. Utilizar `delete` en un apuntador nulo siempre es seguro. Por ejemplo,

```
Animal *apPerro = new Animal;  
delete apPerro; //libera la memoria  
apPerro = NULL; //asigna NULL al apuntador  
//...  
delete apPerro; //inofensivo
```

### ENTRADA

### LISTADO 8.4 Asignación, uso y eliminación de apuntadores

```
1: // Listado 8.4  
2: // Asignación y eliminación de un apuntador  
3:  
4: #include <iostream.h>  
5: int main()  
6: {  
7:     int variableLocal = 5;  
8:     int * apLocal= &variableLocal;  
9:     int * apHeap = new int;  
10:  
11:    *apHeap = 7;  
12:    cout << "variableLocal: " << variableLocal << "\n";  
13:    cout << "*apLocal: " << *apLocal << "\n";  
14:    cout << "*apHeap: " << *apHeap << "\n";  
15:    delete apHeap;  
16:    apHeap = new int;  
17:    *apHeap = 9;  
18:    cout << "*apHeap: " << *apHeap << "\n";  
19:    delete apHeap;  
20:    return 0;  
21: }
```

**SALIDA**

```
variableLocal: 5
*apLocal: 5
*apHeap: 7
*apHeap: 9
```

**ANÁLISIS**

La línea 7 declara e inicializa una variable local. La línea 8 declara e inicializa un apuntador con la dirección de la variable local. La línea 9 declara otro apuntador pero lo inicializa con el resultado obtenido al solicitar un nuevo int. Esto asigna espacio en heap para un int. La línea 11 verifica que se haya asignado la memoria y que el apuntador sea válido (no nulo) al utilizarlo. Si no se puede asignar memoria, el apuntador es nulo y se imprime un mensaje de error.

Para mantener las cosas simples, la verificación de errores real no se incluye en muchos de los programas de muestra que vienen en este libro, pero usted debe incluir algún tipo de verificación de errores en sus propios programas. Una mejor forma de comprobar errores sería probar si un apuntador es nulo antes de inicializarlo con el valor. Podría reemplazar la línea 11 del listado 8.4 con lo siguiente:

```
if (apHeap == NULL)
{
    // encargarse aquí del error:
    // reportar el problema. En la mayoría de los casos los siguientes pasos
    // son realizar la limpieza (cerrar archivos, establecer estados, etc.)
    // y salir del programa.
}
*apHeap = 7;
```

La línea 11 asigna el valor 7 a la memoria recién asignada. La línea 12 imprime el valor de la variable local, y la línea 13 imprime el valor al que apunta apLocal. Como es de esperarse, son iguales. La línea 14 imprime el valor al que apunta apHeap. Esto muestra que el valor que se asigna en la línea 11 sí es accesible.

En la línea 15, la memoria que se asigna en la línea 9 se libera mediante una llamada a delete. Esto libera la memoria y el apuntador queda desasociado de esa memoria. Ahora apHeap está libre para apuntar a otra dirección de memoria. Se le vuelve a asignar una dirección en las líneas 16 y 17, y la línea 18 imprime el resultado. La línea 19 libera esa memoria.

Aunque la línea 19 es redundante (al terminar el programa se habría liberado esa memoria), es una buena idea liberar esta memoria en forma explícita. Si el programa cambia o se extiende, haberse hecho cargo de este paso será benéfico.

## Fugas de memoria

8

Otra forma de crear inadvertidamente una fuga de memoria es reasignar su apuntador antes de eliminar la memoria a la cual está apuntando. Considere este fragmento de código:

```
1: unsigned short int * apApuntador = new unsigned short int;
2: *apApuntador = 72;
3: apApuntador = new unsigned short int;
4: *apApuntador = 84;
```

La línea 1 crea a `apApuntador` y le asigna la dirección de un área del heap. La línea 2 guarda el valor 72 en esa área de memoria. La línea 3 vuelve a asignar otra área de memoria a `apApuntador`. La línea 4 coloca el valor 84 en esa área. El área original (en la que se guarda el valor 72) no está disponible, ya que el apuntador a esa área de memoria ha sido reasignado. No existe forma de tener acceso a esa área original de memoria, ni de liberarla antes de que el programa termine.

El código se debió escribir de la siguiente manera:

```
1: unsigned short int * apApuntador = new unsigned short int;
2: *apApuntador = 72;
3: delete apApuntador;
4: apApuntador = new unsigned short int;
5: *apApuntador = 84;
```

Ahora la memoria a la que apuntaba originalmente `apApuntador` se ha eliminado, es decir, liberado, en la línea 3.

### Nota

Por cada vez que utilice `new` en su programa, debe haber un `delete` correspondiente. Es importante llevar un registro de cuál área de memoria pertenece a cuál apuntador, y de asegurarse de que esa memoria se libere al dejar de utilizarla.

## Objetos en el heap

Aún hay muchas cosas que aprender acerca del uso y la creación de objetos. En las siguientes secciones verá estos temas:

- Creación de objetos en el heap
- Eliminación de objetos
- Acceso a los datos miembro
- Datos miembro en el heap

## Creación de objetos en el heap

Así como puede crear un apuntador a un entero, también puede crear un apuntador a cualquier objeto. Si crea un objeto de la clase `Gato`, puede declarar un apuntador a ese objeto `Gato` en el heap, de igual forma que como crea uno en la pila. La sintaxis es la misma que la de los enteros:

```
Gato *apGato = new Gato;
```

Esto llama al constructor predeterminado (el constructor que no lleva parámetros). El constructor se llama siempre que se crea un objeto (en la pila o en el heap).

## Eliminación de objetos

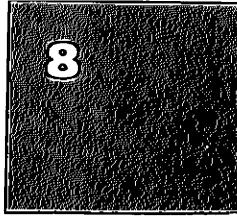
Cuando llama a `delete` para que actúe sobre un apuntador a un objeto en heap, el destructor de ese objeto se llama antes de que se libere la memoria. Esto le da a su clase una oportunidad para limpiarse, igual que como se hace para los objetos que se destruyen en la pila. El listado 8.5 muestra la creación y la eliminación de objetos en el heap.

**ENTRADA****LISTADO 8.5 Creación y eliminación de objetos en el heap**

---

```
1: // Listado 8.5
2: // Creación de objetos en el heap
3:
4: #include <iostream.h>
5:
6: class GatoSimple
7: {
8: public:
9:     GatoSimple();
10:    ~GatoSimple();
11: private:
12:     int suEdad;
13: };
14:
15: GatoSimple::GatoSimple()
16: {
17:     cout << "Se llamó al constructor.\n";
18:     suEdad = 1;
19: }
20:
21: GatoSimple::~GatoSimple()
22: {
23:     cout << "Se llamó al destructor.\n";
24: }
25:
26: int main()
27: {
```

```
28:             cout << "GatoSimple Pelusa...\n";
29:             GatoSimple Pelusa;
30:             cout << "GatoSimple *apFelix = new GatoSimple...\n";
31:             GatoSimple * apFelix = new GatoSimple;
32:             cout << "delete apFelix...\n";
33:             delete apFelix;
34:             cout << "saliendo, observe cómo se va Pelusa...\n";
35:             return 0;
36: }
```

8**SALIDA**

```
GatoSimple Pelusa...
Se llamó al constructor.
GatoSimple *apFelix = new GatoSimple...
Se llamó al constructor.
delete apFelix...
Se llamó al destructor.
saliendo, observe cómo se va Pelusa...
Se llamó al destructor.
```

**ANÁLISIS**

Las líneas 6 a 13 declaran la clase simplificada `GatoSimple`. La línea 9 declara el constructor de `GatoSimple`, y las líneas 15 a 19 contienen su definición. La línea 10 declara el destructor de `GatoSimple`, y las líneas 21 a 24 contienen su definición.

En la línea 29 se crea en la pila el objeto `Pelusa`, lo que ocasiona que se llame al constructor. En la línea 31 se crea en el heap el `GatoSimple` al que apunta `apFelix`; se llama otra vez al constructor. En la línea 33 se llama a `delete` para que actúe sobre `apFelix`, y se llama al destructor. Al terminar la función, `Pelusa` queda fuera de alcance, y se llama al destructor.

## Acceso a los datos miembro

Usted tiene acceso a los datos miembro y a las funciones mediante el operador de punto (`.`) para los objetos `Gato` creados en forma local. Para tener acceso al objeto `gato` en el heap, debe desreferenciar el apuntador y llamar al operador de punto en el objeto al que apunta el apuntador. Por lo tanto, para tener acceso a la función miembro `ObtenerEdad`, debe escribir lo siguiente:

```
(*apFelix).ObtenerEdad();
```

Los paréntesis se utilizan para asegurar que `apFelix` sea desreferenciado antes de tener acceso a `ObtenerEdad()`.

Como esto es un poco extraño, C++ proporciona un operador de método abreviado para el acceso indirecto: el operador de flecha (`->`), que se crea escribiendo un guión corto (`-`) seguido del signo de mayor que (`>`). C++ trata esto como un solo símbolo. El listado 8.6 muestra el acceso a las variables y funciones miembro de objetos creados en el heap.

**ENTRADA****LISTADO 8.6** Acceso a los datos miembro de los objetos que se encuentran en el heap

```

1:      // Listado 8.6 Acceso a los datos miembro
2:      // de objetos que se encuentran en el heap
3:
4:      #include <iostream.h>
5:
6:      class GatoSimple
7:      {
8:      public:
9:          GatoSimple()
10:         { suEdad = 2; }
11:         ~GatoSimple() {}
12:         int ObtenerEdad() const
13:             { return suEdad; }
14:         void AsignarEdad(int edad)
15:             { suEdad = edad; }
16:     private:
17:         int suEdad;
18:     };
19:
20:    int main()
21:    {
22:        GatoSimple * Pelusa = new GatoSimple;
23:        cout << "Pelusa tiene " << Pelusa->ObtenerEdad();
24:        cout << "años de edad\n";
25:        Pelusa->AsignarEdad(5);
26:        cout << "Pelusa tiene " << Pelusa->ObtenerEdad();
27:        cout << " años de edad\n";
28:        delete Pelusa;
29:        return 0;
30:    }

```

**SALIDA**

Pelusa tiene 2 años de edad  
 Pelusa tiene 5 años de edad

**ANÁLISIS**

En la línea 22 se crea una instancia de un objeto `GatoSimple` en el heap. El constructor predeterminado establece su edad en 2, y en la línea 23 se llama al método `ObtenerEdad()`. Como éste es un apuntador, se utiliza el operador de flecha (`->`) para tener acceso a los datos miembro y a las funciones. En la línea 25 se hace una llamada al método `AsignarEdad()`, y se vuelve a acceder a `ObtenerEdad()` en la línea 26.

## Datos miembro en el heap

Uno o más de los datos miembro de una clase pueden ser apuntadores a un objeto que se encuentre en el heap. La memoria se puede asignar en el constructor de la clase o en uno de sus métodos, y se puede eliminar en su destructor, como se muestra en el listado 8.7.

**ENTRADA****LISTADO 8.7** Apuntadores como datos miembro

```
1: // Listado 8.7
2: // Apuntadores como datos miembro
3:
4: #include <iostream.h>
5:
6: class GatoSimple
7: {
8: public:
9:     GatoSimple();
10:    ~GatoSimple();
11:    int ObtenerEdad() const
12:    { return *suEdad; }
13:    void AsignarEdad(int edad)
14:    { *suEdad = edad; }
15:    int ObtenerPeso() const
16:    { return *suPeso; }
17:    void AsignarPeso (int peso)
18:    { *suPeso = peso; }
19: private:
20:     int * suEdad;
21:     int * suPeso;
22: };
23:
24: GatoSimple::GatoSimple()
25: {
26:     suEdad = new int(2);
27:     suPeso = new int(5);
28: }
29:
30: GatoSimple::~GatoSimple()
31: {
32:     delete suEdad;
33:     delete suPeso;
34: }
35:
36: int main()
37: {
38:     GatoSimple * Pelusa = new GatoSimple;
39:     cout << "Pelusa tiene " << Pelusa->ObtenerEdad();
40:     cout << " años de edad\n";
41:     Pelusa->AsignarEdad(5);
42:     cout << "Pelusa tiene " << Pelusa->ObtenerEdad();
43:     cout << " años de edad\n";
44:     delete Pelusa;
45:     return 0;
46: }
```

**SALIDA**

```
Pelusa tiene 2 años de edad
Pelusa tiene 5 años de edad
```

**ANÁLISIS**

En las líneas 20 y 21 se declara la clase `GatoSimple` con dos variables miembro (ambas son apuntadores a enteros). El constructor (líneas 24 a 28) inicializa, con los valores predeterminados, los apuntadores a la memoria del heap.

El destructor (líneas 30 a 34) libera la memoria asignada. Como éste es el destructor, no tiene caso asignarles `NULL` a estos apuntadores, pues ya no serán accesibles. Éste es uno de los lugares seguros en los que se puede romper la regla que establece que a los apuntadores eliminados se les debe asignar `NULL`, aunque seguir la regla no hace daño.

La función que hace la llamada (en este caso, `main()`) no sabe que `suEdad` y `suPeso` son apuntadores a la memoria del heap. `main()` continúa llamando a `ObtenerEdad()` y a `AsignarEdad()`, y los detalles del manejo de memoria se ocultan en la implementación de la clase (como debe ser).

Cuando se elimina a `Pelusa` en la línea 44, se llama a su destructor. El destructor elimina cada uno de sus apuntadores miembro. Si éstos, a su vez, apuntan a objetos de otras clases definidas por el usuario, también se llama a sus destructores.

**Preguntas frecuentes**

**FAQ:** Si declaro un objeto en la pila que tiene variables miembro en el heap, ¿qué hay en la pila y qué hay en el heap? Por ejemplo:

```
#include <iostream.h>

class GatoSimple
{
public:
    GatoSimple();
    ~GatoSimple();
    int ObtenerEdad() const
        { return *suEdad; }
    // otros métodos

private:
    int * suEdad;
    int * suPeso;
};

GatoSimple::GatoSimple()
{
    suEdad = new int( 2 );
    suPeso = new int( 5 );
}
GatoSimple::~GatoSimple()
{
    delete suEdad;
    delete suPeso;
```

```
}

int main()
{
    GatoSimple Pelusa;
    cout << "Pelusa tiene " <<
        Pelusa.ObtenerEdad() << " años de edad\n";
    Pelusa.AsignarEdad(5);
    cout << "Pelusa tiene " <<
        Pelusa.ObtenerEdad() << " años de edad\n";
    return 0;
}
```

8

**Respuesta:** Lo que hay en la pila es la variable local Pelusa. Esa variable tiene dos apuntadores, cada uno de los cuales ocupa algo de espacio en la pila y guarda la dirección de un entero asignado en el heap. Por lo tanto, en el ejemplo hay 8 bytes en la pila (se da por hecho que son apuntadores de 4 bytes) y 8 bytes en el heap.

Ahora esto sería bastante raro en un programa real, a menos que existiera un buen motivo para que el objeto Gato guardara sus miembros por referencia. En este caso no hay un buen motivo, pero en otros casos esto tendría mucho sentido.

Esto hace que surja la siguiente pregunta: ¿Qué está tratando de lograr? Debe entender también que debe empezar con el diseño. Si lo que diseñó es un objeto que se refiere a otro objeto, pero el segundo objeto tal vez empieza a existir antes que el primero, y continúe después de que el primero haya desaparecido, entonces el primer objeto debe contener al segundo por referencia.

Por ejemplo, el primer objeto podría ser una ventana y el segundo podría ser un documento. La ventana necesita acceso al documento, pero no controla el tiempo de vida de éste. Por lo tanto, la ventana necesita guardar el documento por referencia.

En C++, esto se implementa mediante el uso de apuntadores o referencias. Las referencias se tratan en el día 9, “Referencias”.

## Apunadores especiales y otras cuestiones

Aún quedan muchas cosas por aprender acerca del uso de apuntadores. Las siguientes secciones tratarán sobre:

- El apuntador `this`
- Apuntadores perdidos, descontrolados o ambulantes
- Apuntadores `const`

## El apuntador **this**

Toda función miembro de una clase tiene un parámetro oculto: el apuntador **this**. **this** apunta al propio objeto. Por lo tanto, en cada llamada a `ObtenerEdad()` o a `AsignarEdad()`, el apuntador **this** para el objeto se incluye como un parámetro oculto.

Es posible usar el apuntador **this** en forma explícita, como lo muestra el listado 8.8.

**ENTRADA****LISTADO 8.8 Uso del apuntador **this****

```
1:      // Listado 8.8
2:      // Uso del apuntador this
3:
4:      #include <iostream.h>
5:
6:      class Rectangulo
7:      {
8:      public:
9:      Rectangulo();
10:     ~Rectangulo();
11:     void AsignarLongitud(int longitud)
12:     { this->suLongitud = longitud; }
13:     int ObtenerLongitud() const
14:     { return this->suLongitud; }
15:     void AsignarAncho(int ancho)
16:     { suAncho = ancho; }
17:     int ObtenerAncho() const
18:     { return suAncho; }
19:     private:
20:     int suLongitud;
21:     int suAncho;
22:   };
23:
24:   Rectangulo::Rectangulo()
25:   {
26:     suAncho = 5;
27:     suLongitud = 10;
28:   }
29:
30:   Rectangulo::~Rectangulo()
31:   {}
32:
33:   int main()
34:   {
35:     Rectangulo elRect;
36:
37:     cout << "elRect tiene " << elRect.ObtenerLongitud();
38:     cout << " pies de largo.\n";
39:     cout << "elRect tiene " << elRect.ObtenerAncho();
40:     cout << " pies de ancho.\n";
41:     elRect.AsignarLongitud(20);
42:     elRect.AsignarAncho(10);
43:     cout << "elRect tiene " << elRect.ObtenerLongitud();
44:     cout << " pies de largo.\n";
```

```
45:         cout << "elRect tiene " << elRect.ObtenerAncho();
46:         cout << " pies de ancho.\n";
47:         return 0;
48:     }
```

8

**SALIDA**

```
elRect tiene 10 pies de largo.
elRect tiene 5 pies de ancho.
elRect tiene 20 pies de largo.
elRect tiene 10 pies de ancho.
```

**ANÁLISIS**

Las funciones de acceso `AsignarLongitud()` y `ObtenerLongitud()` utilizan explícitamente el apuntador `this` para tener acceso a las variables miembro del objeto `Rectangulo`. Las funciones de acceso `AsignarAncho` y `ObtenerAncho` no lo hacen. No existe ninguna diferencia en su comportamiento, aunque la sintaxis es más fácil de entender.

Si eso fuera todo lo que hubiera en relación con el apuntador `this`, no tendría mucho caso que usted se tomara la molestia de utilizarlo. Sin embargo, hay que recordar que es un apuntador; guarda la dirección de memoria de un objeto, y como tal, puede ser una herramienta poderosa.

En el día 10, “Funciones avanzadas”, verá un uso práctico del apuntador `this`, cuando hablemos de la sobrecarga de operadores. Por ahora, su meta es conocer este apuntador y entender lo que es: un apuntador al objeto en sí.

No tiene que preocuparse por crear o eliminar el apuntador `this`. El compilador se encarga de eso.

## Apuntadores perdidos, descontrolados o ambulantes

Los apuntadores perdidos producen errores desagradables y difíciles de encontrar. Un *apuntador perdido* (también conocido como apuntador descontrolado o ambulante) se crea si usted llama a `delete` para que actúe sobre un apuntador (lo que en consecuencia libera la memoria a la que éste apunta), y no le asigna `NULL` a ese apuntador. Si trata entonces de utilizar ese apuntador sin reasignarlo, el resultado será impredecible y con suerte el programa sólo dejará de funcionar.

Es como si la compañía Acme de pedidos por correspondencia se mudara, y usted aún oprimiera el botón programado en su teléfono. Es posible que no ocurra nada terrible (es un teléfono que suena en un almacén desierto). Tal vez el número haya sido reasignado a una fábrica de municiones, su llamada detone un explosivo ¡y toda la ciudad vuele en pedazos!

Para resumir, tenga cuidado de no utilizar un apuntador después de haber llamado a `delete` para que actúe sobre él. El apuntador aún apunta al área de memoria, pero el compilador tiene la libertad de colocar otros datos ahí; si usa el apuntador, su programa puede dejar de funcionar. O peor aún, su programa podría proceder como si nada y luego dejar de funcionar varios minutos después. Esto se conoce como bomba de tiempo, y no es divertido. Para estar seguro, después de usar `delete` sobre un apuntador, asígnelo a `NULL`. Con esto, el apuntador queda desarmado.

 **Nota**

Los apuntadores perdidos se conocen comúnmente como apuntadores descontrolados o ambulantes. Esto es sólo una diferencia en la terminología.

El listado 8.9 muestra la creación de un apuntador perdido.

 **Precaución**

Este programa crea un apuntador perdido. NO ejecute este programa; con suerte, sólo dejará de funcionar.

**ENTRADA****LISTADO 8.9** Creación de un apuntador perdido

```
1: // Listado 8.9
2: // Muestra de un apuntador perdido
3:
4: #include <iostream.h>
5:
6: typedef unsigned short int USHORT;
7:
8: int main()
9: {
10:     USHORT * apInt = new USHORT;
11:     *apInt = 10;
12:     cout << "*apInt: " << *apInt << endl;
13:     delete apInt;
14:
15:     long * apLong = new long;
16:     *apLong = 90000;
17:     cout << "*apLong: " << *apLong << endl;
18:
19:     *apInt = 20; // icaramba, éste fue eliminado!
20:
21:     cout << "*apInt: " << *apInt << endl;
22:     cout << "*apLong: " << *apLong << endl;
23:     delete apLong;
24:     return 0;
25: }
```

**SALIDA**

```
*apInt: 10
*apLong: 90000
*apInt: 20
*apLong: 65556
```

(La salida que usted obtenga puede ser distinta.)

**ANÁLISIS**

La línea 10 declara a apInt como apuntador a USHORT, y el mismo apInt apunta a la memoria recién asignada. La línea 11 coloca el valor 10 en esa memoria, y la línea 12 imprime su valor. Después de imprimir el valor, se utiliza delete en el apuntador. Ahora apInt es un apuntador perdido, o descontrolado.

La línea 15 declara un nuevo apuntador, apLong, el cual apunta a la memoria asignada por new. La línea 16 asigna el valor de 90000 a apLong, y la línea 17 imprime su valor.

La línea 19 asigna el valor 20 a la memoria a la que apunta apInt, pero apInt ya no apunta a ningún lugar válido. La memoria a la que apunta apInt fue liberada por la llamada a delete, por lo que asignar un valor a esa memoria es un desastre evidente.

La línea 21 imprime el valor de apInt. Como era de esperarse, es 20. La línea 22 imprime el valor de apLong; este valor ha cambiado repentinamente a 65556. Surgen dos preguntas:

1. ¿Cómo pudo cambiar el valor de apLong, si ni siquiera se tocó a apLong?
2. ¿En dónde se almacenó el valor 20 cuando se utilizó apInt en la línea 19?

Como podría suponer, éstas son preguntas relacionadas. Cuando se colocó un valor en apInt en la línea 19, el compilador colocó felizmente el valor de 20 en la ubicación de memoria a la que apInt apuntaba previamente. Sin embargo, como esa memoria fue liberada en la línea 13, el compilador tenía la libertad de reasignarla. Cuando se creó apLong en la línea 15, se le asignó la antigua ubicación de memoria de apInt. (En algunas computadoras tal vez no pase esto, dependiendo del lugar de la memoria en donde se guarden estas variables.) Cuando se asignó el valor 20 a la ubicación a la que apuntaba apInt previamente, éste sobreescribió el valor al que apuntaba apLong. Esto se conoce como “pisotear un apuntador”. Por lo general, éste es el desafortunado producto obtenido al usar un apuntador perdido.

Éste es un error especialmente desagradable, ya que el valor que cambió no estaba asociado con el apuntador perdido. El cambio al valor en apLong fue un efecto secundario del mal uso de apInt. En un programa grande, esto sería muy difícil de rastrear.

Sólo por diversión, he aquí los detalles de cómo llegó el valor 65556 a esa dirección de memoria:

1. apInt apuntaba a una ubicación específica de memoria, y se asignó el valor 10.
2. Se llamó a delete para que actuara sobre apInt, lo cual le indicó al compilador que podía colocar cualquier otra cosa en esa ubicación. Luego se asignó a apLong la misma ubicación de memoria.
3. Se asignó el valor de 90000 a \*apLong. La computadora utilizada en este ejemplo guardó el valor de 4 bytes de 90.000 (00 01 5F 90) con los bytes intercambiados. Por lo tanto, dicho valor se guardó como 5F 90 00 01.
4. Se asignó a apInt el valor de 20 (o 00 14 en notación hexadecimal). Como apInt aún apuntaba a la misma dirección, se sobreescribieron los dos primeros bytes de apLong, con lo cual quedó como 00 14 00 01.

5. Se imprimió el valor en apLong, con lo cual los bytes regresaron a su orden correcto de 00 01 00 14, lo que se tradujo al valor 65556.

| DEBE                                                                                                                                                                                                                             | No DEBE                                                                                                                                                                                                                          |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>DEBE</b> utilizar new para crear objetos en el heap.</p> <p><b>DEBE</b> utilizar delete para destruir objetos en heap y para liberar la memoria que ocupaban.</p> <p><b>DEBE</b> verificar el valor regresado por new.</p> | <p><b>NO DEBE</b> olvidarse de balancear todas las instrucciones new con su instrucción delete correspondiente.</p> <p><b>NO DEBE</b> olvidarse de asignar NULL a todos los apuntadores sobre los que haya utilizado delete.</p> |

#### Preguntas frecuentes

**FAQ:** ¿Cuál es la diferencia entre un apuntador nulo y un apuntador perdido?

**Respuesta:** Cuando elimina un apuntador, le dice al compilador que libere la memoria, pero el apuntador en sí no deja de existir. Ahora es un apuntador perdido.

Si después de esto escribe miApuntador = NULL;, lo transforma de apuntador perdido en apuntador nulo.

Por lo general, si utiliza delete sobre un apuntador y luego vuelve a utilizar delete, su programa queda indefinido. Es decir, podría pasar cualquier cosa (con suerte, el programa sólo dejará de funcionar). Si elimina un apuntador nulo, no pasa nada; esto es seguro.

El uso de un apuntador perdido o de uno nulo (por ejemplo, escribir miApuntador = 5;) es ilegal, y el programa podría dejar de funcionar. Si el apuntador es nulo, dejará de funcionar, un beneficio más del apuntador nulo sobre el apuntador perdido. Los autores prefieren que el programa deje de funcionar en forma predecible, ya que esto es más fácil de depurar.

## Apuntadores const

En los apuntadores, puede utilizar la palabra reservada const antes del tipo, después del tipo, o en ambos lugares. Por ejemplo, las siguientes declaraciones son válidas:

```
const int * apUno;
int * const apDos;
const int * const apTres;
```

apUno es un apuntador a un valor entero constante. El valor al que apunta no se puede cambiar.

apDos es un apuntador constante a un entero. Se puede cambiar el entero, pero apDos no puede apuntar a ninguna otra cosa.

apTres es un apuntador constante a un entero constante. El valor al que apunta no se puede cambiar, y apTres no puede apuntar a ninguna otra cosa.

El truco para usar esto sin problemas es mirar a la derecha de la palabra reservada `const` para saber qué se está declarando como constante. Si el tipo se encuentra a la derecha de la palabra reservada, es el valor el que se declara como constante. Si es la variable la que se encuentra a la derecha, entonces es la propia variable de apuntador la que se declara como constante.

```
const int * ap1; // el valor int al que se apunta es constante
int * const ap2; // ap2 es constante, no puede apuntar a ninguna otra cosa
```

### Apuntadores `const` y funciones miembro `const`

En el día 6, "Clases base", aprendió que puede aplicar la palabra reservada `const` a una función miembro o método. Cuando una función se declara como `const`, el compilador marca como error cualquier intento por cambiar los datos del objeto desde el interior de esa función.

Si declara un apuntador a un objeto `const`, los únicos métodos que puede llamar con ese apuntador son métodos `const`. El listado 8.10 muestra esto.

**ENTRADA****LISTADO 8.10** Uso de apuntadores a objetos `const`

```
1:  // Listado 8.10
2:  // Uso de apuntadores con métodos const
3:
4:  #include <iostream.h>
5:
6:  class Rectangulo
7:  {
8:  public:
9:      Rectangulo();
10:     ~Rectangulo();
11:     void AsignarLongitud(int longitud)
12:     { suLongitud = longitud; }
13:     int ObtenerLongitud() const
14:     { return suLongitud; }
15:     void AsignarAncho(int ancho)
16:     { suAncho = ancho; }
17:     int ObtenerAncho() const
18:     { return suAncho; }
19: private:
20:     int suLongitud;
21:     int suAncho;
22: };
23:
24: Rectangulo::Rectangulo()
25: {
26:     suAncho = 5;
27:     suLongitud = 10;
28: }
29:
30: Rectangulo::~Rectangulo()
31: {}
32:
```

**LISTADO 8.10** continuación

```

33:     int main()
34:     {
35:         Rectangulo * apRect = new Rectangulo;
36:         const Rectangulo * apConstRect = new Rectangulo;
37:         Rectangulo * const apConstApunt = new Rectangulo;
38:
39:         cout << "Ancho de apRect: ";
40:         cout << apRect->ObtenerAncho() << " pies\n";
41:         cout << "Ancho de apConstRect: ";
42:         cout << apConstRect->ObtenerAncho() << " pies\n";
43:         cout << "Ancho de apConstApunt: ";
44:         cout << apConstApunt->ObtenerAncho() << " pies\n";
45:
46:         apRect->AsignarAncho(10);
47:         // apConstRect->AsignarAncho(10);
48:         apConstApunt->AsignarAncho(10);
49:
50:         cout << "Ancho de apRect: ";
51:         cout << apRect->ObtenerAncho() << " pies\n";
52:         cout << "Ancho de apConstRect: ";
53:         cout << apConstRect->ObtenerAncho() << " pies\n";
54:         cout << "Ancho de apConstApunt: ";
55:         cout << apConstApunt->ObtenerAncho() << " pies\n";
56:     return 0;
57: }
```

**SALIDA**

Ancho de apRect: 5 pies  
 Ancho de apConstRect: 5 pies  
 Ancho de apConstApunt: 5 pies  
 Ancho de apRect: 10 pies  
 Ancho de apConstRect: 5 pies  
 Ancho de apConstApunt: 10 pies

**ANÁLISIS**

Las líneas 6 a 22 declaran la clase Rectangulo. La línea 17 declara el método const llamado ObtenerAncho(). La línea 35 declara un apuntador a Rectangulo.

La línea 36 declara a apConstRect, que es un apuntador a un Rectangulo constante. La línea 37 declara a apConstApunt, que es un apuntador constante a Rectangulo.

Las líneas 39 a 44 imprimen sus valores.

En la línea 46 se utiliza apRect para establecer en 10 el ancho del rectángulo. En la línea 47 se utilizaría apConstRect, pero se declaró como apuntador a un Rectangulo constante. Por lo tanto, no puede llamar legalmente a una función miembro que no sea constante; se convierte en comentario. En la línea 48, apConstApunt llama a AsignarAncho(). apConstApunt se declara como apuntador constante a un rectángulo. En otras palabras, el apuntador es constante y no puede apuntar a ninguna otra cosa, pero el rectángulo no es constante. Si elimina las barras diagonales de comentario en la línea 41, se ejecutará una instrucción ilegal. En algunos compiladores GNU sólo recibirá una advertencia que le indicará que puede estar modificando una variable o una función declarada como const. El mensaje del compilador se verá de la siguiente manera:

```
lst08-10.cxx: in function int main():
lst08-10.cxx:41: warning: passing 'const Rectangulo' as 'this' argument of 'void
Rectangulo::AsignarAncho(int)' discards const
```

Esto le indica que se descartará este método como `const` y se podrá utilizar como cualquier otro método. En este caso, la salida del programa será la siguiente:

```
Ancho de apRect: 5 pies
Ancho de apConstRect: 5 pies
Ancho de apConstApunt: 5 pies
Ancho de apRect: 10 pies
Ancho de apConstRect: 10 pies
Ancho de apConstApunt: 10 pies
```

Otros compiladores sólo detendrán la compilación del código. Verifique la acción que realiza su compilador en estos casos.

### Apuntadores `this const`

Cuando declara un objeto para que sea `const`, está en efecto declarando que el apuntador `this` es un apuntador a un objeto `const`. Un apuntador `this` que sea `const` se puede utilizar solamente con funciones miembro `const`.

Los objetos constantes y los apuntadores constantes se describirán otra vez mañana, cuando hablemos sobre las referencias a objetos constantes.

| DEBE                                                                                                                                                                                                                                                                                        | No DEBE |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|
| <p><b>DEBE</b> proteger los objetos pasados por referencia con <code>const</code> si no se deben modificar.</p> <p><b>DEBE</b> pasar por referencia los objetos que se puedan modificar.</p> <p><b>DEBE</b> realizar el paso por valor cuando los objetos pequeños no se deban cambiar.</p> |         |

## Aritmética de apuntadores

Los apuntadores se pueden restar unos de otros. Una técnica poderosa es hacer que dos apuntadores apunten a diferentes elementos de un arreglo, y tomar su diferencia para ver cuántos elementos separan a los dos miembros. Esto puede ser útil al analizar sintácticamente arreglos de caracteres, como se muestra en el listado 8.11.

### ENTRADA

### LISTADO 8.11 Ejemplo de cómo analizar sintácticamente palabras de una cadena de caracteres

```
1: // Listado 8.11 Muestra el análisis sintáctico de
2: // palabras en una cadena de caracteres
```

*continúa*

**LISTADO 8.11** CONTINUACIÓN

```
3:      #include <iostream.h>
4:      #include <ctype.h>
5:      #include <string.h>
6:
7:
8:      bool ObtenerPalabra(char* cadena,
9:                            char* palabra,
10:                           int & desplazamientoDePalabra);
11:
12:     // programa controlador
13:     int main()
14:     {
15:         const int tamBufer = 255;
16:         char buffer[ tamBufer + 1 ];      // guardar toda la cadena
17:         char palabra[ tamBufer + 1 ];    // guardar una palabra
18:         int desplazamientoDePalabra = 0; // empezar al principio
19:
20:         cout << "Escriba una cadena: ";
21:         cin.getline(buffer, tamBufer);
22:         while (ObtenerPalabra(buffer,
23:                               palabra,
24:                               desplazamientoDePalabra))
25:         {
26:             cout << "Obtuve esta palabra: " << palabra << endl;
27:         }
28:         return 0;
29:     }
30:
31:     // función para analizar sintácticamente
32:     // palabras de una cadena.
33:     bool ObtenerPalabra(char* cadena,
34:                          char* palabra,
35:                          int & desplazamientoDePalabra)
36:     {
37:         // ¿es fin de cadena?
38:         if (!cadena[ desplazamientoDePalabra ])
39:             return false;
40:         char * ap1, * ap2;
41:
42:         // apuntar a la siguiente palabra
43:         ap1 = ap2 = cadena + desplazamientoDePalabra;
44:
45:         // saltarse los primeros espacios en blanco
46:         for (int i = 0;
47:              i < (int)strlen(ap1) && !isalnum(ap1[ 0 ]);
48:              i++)
49:             ap1++;
50:
51:         // ver si se tiene una palabra
52:         if (!isalnum(ap1[ 0 ]))
53:             return false;
54:
55:         // ap1 ahora apunta al inicio de la siguiente palabra
56:         // hacer que ap2 apunte ahí también
```

```
57:         ap2 = ap1;
58:
59:         // hacer que ap2 apunte al final de la palabra
60:         while (isalnum(ap2[ 0 ]))
61:             ap2++;
62:
63:         // ap2 ahora se encuentra al final de la palabra
64:         // ap1 está al principio de la palabra
65:         // la longitud de la palabra es la diferencia
66:         int len = int (ap2 - ap1);
67:
68:         // copiar la palabra en el búfer
69:         strncpy (palabra, ap1, len);
70:
71:         // hacer que termine con el carácter nulo
72:         palabra[ len ]='\'0';
73:
74:         // ahora encontrar el principio de la siguiente palabra
75:         for (int i = int(ap2 - cadena);
76:              i < (int)strlen(cadena) && !isalnum(ap2[ 0 ]);
77:              i++)
78:             ap2++;
79:         desplazamientoDePalabra = int(ap2 - cadena);
80:         return true;
81:     }
```

**SALIDA**

Escriba una cadena: Este listado se obtuvo de la revista C++ Report  
Obtuve esta palabra: Este  
Obtuve esta palabra: listado  
Obtuve esta palabra: se  
Obtuve esta palabra: obtuve  
Obtuve esta palabra: de  
Obtuve esta palabra: la  
Obtuve esta palabra: revista  
Obtuve esta palabra: C  
Obtuve esta palabra: Report

**ANÁLISIS**

En la línea 20 se pide al usuario que escriba una cadena. Ésta se proporciona a ObtenerPalabra() en la línea 22, junto con un búfer en el que se va a guardar la primera palabra y una variable entera llamada desplazamientoDePalabra, la cual se inicializa con cero en la línea 18. A medida que ObtenerPalabra() regresa palabras, éstas se imprimen hasta que ObtenerPalabra() regrese el valor false.

Cada llamada a ObtenerPalabra() provoca un salto a la línea 33. En la línea 38 verificamos si el valor de cadena[desplazamientoDePalabra] es igual a cero. Esto será verdadero (true) si el código llega al final de la cadena, y en este momento ObtenerPalabra() regresará el valor false.

En la línea 40 se declaran dos apuntadores a carácter, ap1 y ap2, y en la línea 43 se hace que apunten al desplazamiento de la cadena por medio de desplazamientoDePalabra. Al principio, desplazamientoDePalabra es cero, por lo que dichos apuntadores apuntan al principio de la cadena.

Las líneas 46 a 49 avanzan por la cadena, llevando a ap1 hasta el primer carácter alfanumérico. Las líneas 52 y 53 aseguran que se encuentre un carácter alfanumérico; en caso contrario, se regresa el valor `false`.

Ahora `ap1` apunta al principio de la siguiente palabra, y la línea 57 hace que `ap2` apunte a la misma posición.

Entonces, las líneas 60 y 61 ocasionan que `ap2` avance por la palabra, deteniéndose en el primer carácter que no sea alfanumérico (si existen caracteres con acentos, diéresis o eñes se tomarán como fin de palabra). Ahora `ap2` apunta al final de la palabra, y `ap1` apunta al principio de la misma palabra. Al restar `ap1` de `ap2` en línea 66 y al convertir el resultado en un valor entero, el código puede establecer la longitud de la palabra. Luego copia esa palabra en el búfer llamado `palabra`, pasando a `ap1` como el punto de inicio, y pasando como longitud la diferencia que se acaba de establecer.

En la línea 72 el código agrega un carácter nulo para marcar el final de la palabra. Luego se incrementa `ap2` para que apunte al principio de la siguiente palabra, y se coloca el desplazamiento de esa palabra en la referencia a la variable entera llamada `desplazamiento-DePalabra`. Finalmente, se regresa el valor `true` para indicar que se ha encontrado una palabra.

Éste es un ejemplo clásico de código que se entiende mejor si se coloca en un depurador y se analiza su ejecución paso a paso.

## Resumen

Los apuntadores proporcionan un medio poderoso para tener acceso a los datos mediante la dirección. Cada variable tiene una dirección, misma que se puede obtener mediante el uso del operador de dirección (`&`). La dirección se puede guardar en un apuntador.

Los apuntadores se declaran escribiendo el tipo de objeto al que van a apuntar, seguido del operador de dirección (`*`) y del nombre del apuntador. Los apuntadores se deben inicializar para que apunten a un objeto o a `NULL`.

Puede tener acceso al valor que se encuentra en la dirección guardada en un apuntador mediante el uso del operador de dirección (`*`). Puede declarar apuntadores `const`, los cuales no se pueden reasignar para que apunten a otros objetos, y apuntadores a objetos `const`, los cuales no se pueden utilizar para cambiar los objetos a los que apuntan.

Para crear nuevos objetos en el heap, se utiliza la palabra reservada `new` y la dirección que regresa se asigna a un apuntador. Esta memoria se libera llamando a la palabra reservada `delete` para que actúe sobre el apuntador. `delete` libera la memoria, pero no destruye el apuntador. Por lo tanto, usted debe asignarle `NULL` al apuntador después de que se haya liberado su memoria.

## Preguntas y respuestas

**P ¿Por qué son tan importantes los apuntadores?**

**R** Hoy vio cómo se utilizan los apuntadores para guardar la dirección de los objetos que se encuentran en el heap, y cómo se utilizan para pasar argumentos por referencia. Además, en el día 13, "Polimorfismo", verá cómo se utilizan los apuntadores en el poliformismo de clases.

**P ¿Por qué debo preocuparme por declarar algo en el heap?**

**R** Los objetos que se encuentran en el heap persisten después de que la función termina. Además, la capacidad de guardar objetos en el heap le permite decidir en tiempo de ejecución cuántos objetos necesita, en lugar de tener que declarar esto por adelantado. Esto se verá con más detalle mañana.

**P ¿Por qué debo declarar un objeto como const si limita lo que puedo hacer con él?**

**R** Como programador, querrá que el compilador le ayude a encontrar errores. Un grave error difícil de encontrar es una función que cambia un objeto en formas que no son obvias para la función que hace la llamada. Al declarar un objeto como `const` se previenen tales cambios.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, "Respuestas a los cuestionarios y ejercicios", y asegúrese de comprender las respuestas antes pasar al siguiente día.

### Cuestionario

1. ¿Qué operador se utiliza para determinar la dirección de una variable?
2. ¿Qué operador se utiliza para encontrar el valor guardado en una dirección que se guarda en un apuntador?
3. ¿Qué es un apuntador?
4. ¿Cuál es la diferencia entre la dirección que se guarda en un apuntador y el valor que se encuentra en esa dirección?
5. ¿Cuál es la diferencia entre el operador de indirección y el operador de dirección?
6. ¿Cuál es la diferencia entre `const int * apuntUno` e `int * const apuntDos`?

## Ejercicios

1. ¿Qué hacen estas declaraciones?
  - a. int \* apUno;
  - b. int varDos;
  - c. int \* apTres = &varDos;
2. Si tiene una variable de tipo entero corto sin signo llamada suEdad, ¿cómo declararía un apuntador para que manipule a la variable suEdad?
3. Asigne el valor 50 a la variable suEdad usando el apuntador que declaró en el ejercicio 2.
4. Escriba un pequeño programa que declare un entero y un apuntador a ese entero. Asígnale al apuntador la dirección del entero. Utilice el apuntador para asignarle un valor a la variable de tipo entero.
5. **CAZA ERRORES:** ¿Qué está mal en este código?

```
#include <iostream.h>
int main()
{
    int * apInt;
    *apInt = 9;
    cout << "El valor en apInt: " << *apInt;
    return 0;
}
```

6. **CAZA ERRORES:** ¿Qué está mal en este código?

```
int main()
{
    int UnaVariable = 5;
    cout << "UnaVariable: " << UnaVariable << "\n";
    int * apVar = &UnaVariable;
    apVar = 9;
    cout << "UnaVariable: " << *apVar << "\n";
    return 0;
}
```

# SEMANA 2

## Día 9

### Referencias

Ayer aprendió cómo utilizar apuntadores para manipular objetos en el heap y cómo hacer referencia a esos objetos en forma indirecta. Las referencias, el tema de la lección de hoy, le proporcionan casi todo el poder de los apuntadores, pero con una sintaxis mucho más sencilla. Hoy aprenderá lo siguiente:

- Qué son las referencias
- Cuál es la diferencia entre referencias y apuntadores
- Cómo crear referencias y utilizarlas
- Cuáles son las limitaciones de las referencias
- Cómo pasar valores y objetos por referencia hacia y desde las funciones

#### ¿Qué es una referencia?

Una *referencia* es un alias o sinónimo; cuando crea una referencia, la inicializa con el nombre de otro objeto, que viene siendo el destino. A partir de ese momento, la referencia actúa como un nombre alternativo para el destino, y cualquier cosa que le haga a la referencia, en realidad se la hace al destino.

Puede crear una referencia escribiendo el tipo del objeto de destino, seguido del operador de referencia (&) y del nombre de la referencia. Las referencias pueden utilizar cualquier nombre válido para una variable, pero muchos programadores prefieren colocar una "r" antes del nombre de la referencia. Por lo tanto, si tiene una variable de tipo entero llamada `unEntero`, puede crear una referencia a esa variable escribiendo lo siguiente:

```
int & rUnaRef = unEntero;
```

Esto se lee como `rUnaRef` y es una referencia a un entero que se inicializa como referencia a la variable `unEntero`. El listado 9.1 muestra cómo se crean y se utilizan las referencias.



### Nota

Observe que el operador de referencia (&) es el mismo símbolo que se utiliza para el operador de dirección. Sin embargo, son distintos operadores, aunque están relacionados.

---

#### ENTRADA LISTADO 9.1 Creación y uso de referencias

---

```
1:  //Listado 9.1
2:  // Muestra del uso de referencias
3:
4:  #include <iostream.h>
5:
6:  int main()
7:  {
8:      int intUno;
9:      int & rUnaRef = intUno;
10:
11:     intUno = 5;
12:     cout << "intUno: " << intUno << endl;
13:     cout << "rUnaRef: " << rUnaRef << endl;
14:
15:     rUnaRef = 7;
16:     cout << "intUno: " << intUno << endl;
17:     cout << "rUnaRef: " << rUnaRef << endl;
18:     return 0;
19: }
```

---

#### SALIDA

```
intUno: 5
rUnaRef: 5
intUno: 7
rUnaRef: 7
```

**ANÁLISIS**

En la línea 8 se declara una variable local de tipo `int` llamada `intUno`. En la línea 9 se declara una referencia a un entero llamada `rUnaRef`, y se inicializa para hacer referencia a `intUno`. Si declara una referencia pero no la inicializa, obtendrá un error en tiempo de compilación. Las referencias se deben inicializar.

En la línea 11 se asigna el valor 5 a `intUno`. En las líneas 12 y 13 se imprimen los valores de `intUno` y `rUnaRef`, y son, desde luego, el mismo valor.

En la línea 15 se asigna el valor 7 a `rUnaRef`. Como ésta es una referencia, es un alias para `intUno`, y por lo tanto valor 7 se asigna realmente a `intUno`, como se muestra en las impresiones de las líneas 16 y 17.

9

## Uso del operador de dirección (&) en referencias

Si le pide su dirección a una referencia, ésta regresa la dirección de su destino. Ésa es la naturaleza de las referencias: Son un alias para el destino. El listado 9.2 muestra esto.

**ENTRADA****LISTADO 9.2** Cómo tomar la dirección de una referencia

```
1:  //Listado 9.2
2:  // Muestra del uso de referencias
3:
4:  #include <iostream.h>
5:
6:  int main()
7:  {
8:      int intUno;
9:      int & rUnaRef = intUno;
10:
11:     intUno = 5;
12:     cout << "intUno: " << intUno << endl;
13:     cout << "rUnaRef: " << rUnaRef << endl;
14:
15:     cout << "&intUno: " << &intUno << endl;
16:     cout << "&rUnaRef: " << &rUnaRef << endl;
17:
18:     return 0;
19: }
```

**SALIDA**

```
intUno: 5
rUnaRef: 5
&intUno: 0xbfffffa14
&rUnaRef: 0xbfffffa14
```

**Nota**

La salida que usted obtenga puede ser distinta en las dos últimas líneas.

**ANÁLISIS**

Una vez más, `rUnaRef` se inicializa como referencia a `intUno`. Esta vez se imprimen las direcciones de las dos variables, y son idénticas. C++ no le proporciona la manera de tener acceso a la dirección de la referencia en sí, pues no es significativa, como lo sería si estuviera utilizando un apuntador u otra variable. Las referencias se inicializan al crearse, y siempre actúan como sinónimo para su destino, incluso al aplicar el operador de dirección.

Por ejemplo, si tiene una clase llamada `Presidente`, podría declarar una instancia de esa clase, como se muestra a continuación:

```
Presidente Vicente_Fox_Quezada;
```

Entonces podría declarar una referencia a `Presidente` e inicializarla con este objeto:

```
Presidente &Vicente_Fox = Vicente_Fox_Quezada;
```

Sólo existe un `Presidente`; ambos identificadores hacen referencia al mismo objeto de la misma clase. Cualquier acción realizada sobre `Vicente_Fox` afectará también a `Vicente_Fox_Quezada`.

Tenga cuidado de diferenciar entre el símbolo `&` de la línea 9 del listado 9.2, el cual declara una referencia a un entero llamado `rUnaRef`, y los símbolos `&` de las líneas 15 y 16, los cuales regresan las direcciones de la variable de tipo entero llamada `intUno` y de la referencia `rUnaRef`.

Por lo general, al utilizar una referencia, no se utiliza el operador de dirección. Simplemente se utiliza la referencia de la misma forma en que se utilizaría la variable destino. Esto se muestra en la línea 13.

## Las referencias no se pueden reasignar

Aun los programadores de C++ experimentados que conocen la regla (que establece que las referencias no se pueden reasignar y son siempre un alias para su destino) se pueden confundir por lo que pasa si se trata de reasignar una referencia. Lo que parece ser una reasignación resulta ser la asignación de un nuevo valor al destino. El listado 9.3 muestra esto.

**ENTRADA** LISTADO 9.3 Cómo asignar una referencia

```
1: //Listado 9.3
2: //Reasignación de una referencia
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:     int intUno;
9:     int & rUnaRef = intUno;
10:
11:    intUno = 5;
12:    cout << "intUno:\t" << intUno << endl;
13:    cout << "rUnaRef:\t" << rUnaRef << endl;
14:    cout << "&intUno:\t" << &intUno << endl;
15:    cout << "&rUnaRef:\t" << &rUnaRef << endl;
16:
17:    int intDos = 8;
18:    rUnaRef = intDos; // ino es lo que usted piensa!
19:    cout << "\nintUno:\t" << intUno << endl;
20:    cout << "intDos:\t" << intDos << endl;
21:    cout << "rUnaRef:\t" << rUnaRef << endl;
22:    cout << "&intUno:\t" << &intUno << endl;
23:    cout << "&intDos:\t" << &intDos << endl;
24:    cout << "&rUnaRef:\t" << &rUnaRef << endl;
25:
26: }
```

**SALIDA**

```
intUno:      5
rUnaRef:    5
&intUno: 0xbfffffa14
&rUnaRef: 0xbfffffa14

intUno:      8
intDos:      8
rUnaRef:    8
&intUno: 0xbfffffa14
&intDos: 0xbfffffa0c
&rUnaRef: 0xbfffffa14
```

**ANÁLISIS**

Una vez más, en las líneas 8 y 9 se declaran una variable de tipo entero y una referencia a un entero. En la línea 11 se asigna el valor 5 a la variable, y los valores y sus direcciones se imprimen en las líneas 12 a 15.

En la línea 17 se crea una nueva variable llamada `intDos`, y se inicializa con el valor 8. En la línea 18, el programador trata de reasignar a `rUnaRef` a fin de que sea un alias para la

variable `intDos`, pero esto no es lo que ocurre. Lo que ocurre realmente es que `rUnaRef` sigue actuando como alias para `intUno`, por lo que esta asignación es equivalente a:

```
intUno = intDos;
```

Evidentemente, cuando se imprimen los valores de `intUno` y `rUnaRef`, son iguales que el de `intDos` (líneas 19 a 21). De hecho, cuando se imprimen las direcciones en las líneas 22 a 24, puede ver que `rUnaRef` sigue haciendo referencia a `intUno` y no a `intDos`.

| DEBE                                                                 | No DEBE                                                                          |
|----------------------------------------------------------------------|----------------------------------------------------------------------------------|
| <b>DEBE</b> utilizar referencias para crear un alias para un objeto. | <b>NO DEBE</b> tratar de reasignar una referencia.                               |
| <b>DEBE</b> inicializar todas las referencias.                       | <b>NO DEBE</b> confundir el operador de dirección con el operador de referencia. |

## ¿Qué se puede referenciar?

Cualquier objeto se puede referenciar, incluyendo los objetos definidos por el usuario. Observe que crea una referencia a un objeto, pero no a una clase o a un tipo. No debe escribir lo siguiente:

```
int & rIntRef = int; // incorrecto
```

Debe inicializar a `rIntRef` como referencia a un entero específico, como se muestra a continuación:

```
int queTanGrande = 200;
int & rIntRef = queTanGrande;
```

De la misma manera, no debe inicializar una referencia a una clase GATO:

```
GATO & rGatoRef = GATO; // incorrecto
```

Debe inicializar `rGatoRef` como referencia a un objeto específico de la clase GATO:

```
GATO pelusa;
GATO & rGatoRef = pelusa;
```

Las referencias a objetos se utilizan igual que el objeto mismo. Los datos miembro y los métodos se acceden mediante el operador común de acceso a los miembros de una clase (`.`), y al igual que los tipos integrados, la referencia actúa como alias para el objeto. El listado 9.4 muestra esto.

### ENTRADA

### LISTADO 9.4 Referencias a objetos

```
1: // Listado 9.4
2: // Referencias a objetos de una clase
3:
4: #include <iostream.h>
```

```
5: class GatoSimple
6: {
7: public:
8:     GatoSimple (int edad, int peso);
9:     -GatoSimple() {}
10:    int ObtenerEdad()
11:        { return suEdad; }
12:    int ObtenerPeso()
13:        { return suPeso; }
14: private:
15:     int suEdad;
16:     int suPeso;
17: };
18:
19:
20: GatoSimple::GatoSimple(int edad, int peso)
21: {
22:     suEdad = edad;
23:     suPeso = peso;
24: }
25:
26: int main()
27: {
28:     GatoSimple Pelusa(5, 8);
29:     GatoSimple & rGato = Pelusa;
30:
31:     cout << "Pelusa tiene: ";
32:     cout << Pelusa.ObtenerEdad() << " años de edad. \n";
33:     cout << "Y Pelusa pesa: ";
34:     cout << rGato.ObtenerPeso() << " libras. \n";
35:
36: }
```



**SALIDA** Pelusa tiene: 5 años de edad.  
Y Pelusa pesa: 8 libras.

**ANÁLISIS** En la línea 28, Pelusa se declara como un objeto de la clase GatoSimple. En la línea 29 se declara una referencia a GatoSimple llamada rGato, y se inicializa como referencia a Pelusa. En las líneas 32 y 34 se realiza una llamada a los métodos de acceso de GatoSimple utilizando primero el objeto Pelusa y luego la referencia a Pelusa. Observe que el acceso es idéntico. De nuevo, la referencia es un alias para el objeto actual.

### Referencias

Una referencia se declara escribiendo el tipo, seguido del operador de referencia (&) y del nombre de la referencia. Las referencias se deben inicializar al momento de su creación.

**Ejemplo 1:**

```
int suEdad;  
int & rEdad = suEdad;
```

**Ejemplo 2:**

```
GATO silvestre;  
GATO & rGatoRef = silvestre;
```

## Uso de apuntadores nulos y referencias nulas

Cuando no se inicializan los apuntadores, o cuando se eliminan, se les debe asignar NULL. Esto no se aplica a las referencias. De hecho, una referencia no puede ser nula, y un programa que tenga una referencia a un objeto nulo se considera no válido. Cuando un programa no es válido, puede ocurrir cualquier cosa. Puede parecer que funciona, o puede escribir datos raros (e incorrectos) en los archivos de su disco. Ambas son posibles consecuencias de un programa no válido.

La mayoría de los compiladores soporta un objeto nulo sin muchos problemas, y el programa deja de funcionar sólo si se trata de utilizar el objeto. Sin embargo, aprovecharse de esto no es una buena idea. Si transporta su programa a otro equipo o compilador, se podrían desarrollar errores misteriosos si tiene objetos nulos.

## Paso de argumentos de funciones por referencia

En el día 5, "Funciones", aprendió que las funciones tienen dos limitaciones: los argumentos se pasan por valor, y la instrucción `return` sólo puede regresar un valor.

Si se pasan valores a una función por referencia se pueden resolver ambas limitaciones. En C++, pasar por referencia se logra de dos formas: usando apuntadores y usando referencias. Observe la diferencia: Se pasa *por* referencia usando un apuntador, o se pasa *por* referencia usando una referencia.

La sintaxis para usar un apuntador es distinta de la requerida para usar una referencia, pero el efecto neto es el mismo. En lugar de que se cree una copia dentro del alcance de la función, en realidad se pasa el objeto original a la función (en realidad se pasa la dirección del objeto, es decir, su referencia).

En el día 5 aprendió que los parámetros de las funciones se pasan en la pila. Cuando a una función se le pasa un valor por referencia (usando ya sea apuntadores o referencias), se coloca en la pila la dirección del objeto, no todo el objeto.

De hecho, en algunas computadoras la dirección se guarda en un registro y no se coloca nada en la pila. En cualquier caso, ahora el compilador sabe cómo tener acceso al objeto original, y hace los cambios ahí, no en una copia, como sucede cuando se pasan los parámetros por valor.

Si se pasa un objeto por referencia, se permite que la función cambie el objeto al que hace referencia.

Recuerde que el listado 5.5 del día 5 demostró que una llamada a la función `intercambiar()` no afectaba los valores de la función que hizo la llamada. Para su comodidad, el listado 5.5 se reproduce aquí, en el listado 9.5.

9

**ENTRADA LISTADO 9.5 Una muestra de parámetros pasados por valor**

```
1: // Listado 5.5 - muestra de parámetros pasados por valor
2:
3: #include <iostream.h>
4:
5: void intercambiar(int x, int y);
6:
7: int main()
8: {
9:     int x = 5, y = 10;
10:
11:    cout << "main(). Antes del intercambio, x: " << x << " y:
   << y << "\n";
12:    intercambiar(x,y);
13:    cout << "main(). Después del intercambio, x: " << x << " y:
   << y << "\n";
14:    return 0;
15: }
16:
17: void intercambiar (int x, int y)
18: {
19:     int temp;
20:
21:     cout << "Intercambiar(). Antes del intercambio, x: " << x << " y:
   << y << "\n";
22:
23:     temp = x;
24:     x = y;
25:     y = temp;
26:
27:     cout << "Intercambiar(). Después del intercambio, x: " << x << " y:
   << y << "\n";
28:
29: }
```

**SALIDA**

```
main(). Antes del intercambio, x: 5 y: 10
Intercambiar(). Antes del intercambio, x: 5 y: 10
Intercambiar(). Después del intercambio, x: 10 y: 5
main(). Después del intercambio: x: 5 y: 10
```

**ANÁLISIS**

Este programa inicializa dos variables en `main()` y luego las pasa a la función `intercambiar()`, la cual aparentemente las intercambia. Sin embargo, al examinarlas otra vez en `main()`, ¡permanecen sin cambio!

El problema aquí es que `x` y `y` se están pasando a `intercambiar()` por valor. Es decir, se hicieron copias locales en la función. Lo que necesita es pasar `x` y `y` por referencia.

En C++, este problema se puede resolver de dos maneras: puede hacer que los parámetros de `intercambiar()` sean apuntadores a los valores originales, o puede pasar referencias a los valores originales.

## Cómo hacer que `intercambiar()` funcione con apuntadores

Al pasar un apuntador se pasa la dirección del objeto, y por consecuencia la función puede manipular el valor que se encuentra en esa dirección. Para hacer que `intercambiar()` cambie el valor actual por medio de apuntadores, se debe declarar la función `intercambiar()` de manera que acepte dos apuntadores de tipo entero. Entonces, al desreferenciar los apuntadores, se cambiarán realmente los valores de `x` y `y`. El listado 9.6 muestra esto.

**ENTRADA****LISTADO 9.6** Cómo simular el paso por referencia usando apuntadores

```
1:  // Listado 9.6 Muestra el paso por
2:  // referencia simulado usando apuntadores
3:
4:  #include <iostream.h>
5:
6:  void intercambiar(int * x, int * y);
7:
8:  int main()
9:  {
10:    int x = 5, y = 10;
11:
12:    cout << "Main. Antes del intercambio, ";
13:    cout << "x: " << x << " y: " << y << "\n";
14:    intercambiar(&x, &y);
15:    cout << "Main. Después del intercambio, ";
16:    cout << "x: " << x << " y: " << y << "\n";
17:    return 0;
18:  }
19:
20:  void intercambiar (int * apx, int * apy)
21:  {
22:    int temp;
```

```

24:     cout << "Intercambiar. Antes del intercambio, ";
25:     cout << "*apx: " << *apx << " *apy: " << *apy << "\n";
26:
27:     temp = *apx;
28:     *apx = *apy;
29:     *apy = temp;
30:
31:     cout << "Intercambiar. Después del intercambio, ";
32:     cout << "*apx: " << *apx << " *apy: " << *apy << "\n";
33: }

```

9

**SALIDA**

```

Main. Antes del intercambio, x: 5 y: 10
Intercambiar. Antes del intercambio, *apx: 5 *apy: 10
Intercambiar. Después del intercambio, *apx: 10 *apy: 5
Main. Después del intercambio, x: 10 y: 5

```

**ANÁLISIS**

¡Se logró!

En la línea 6, el prototipo de `intercambiar()` se cambia para indicar que sus dos parámetros serán apuntadores a un tipo `int` en lugar de variables. Cuando se llama a `intercambiar()` en la línea 14, se pasan las direcciones de `x` y de `y` como argumentos.

En la línea 22, en la función `intercambiar()`, se declara una variable local llamada `temp`. Esta variable no necesita ser apuntador; sólo guardará el valor de `*apx` (es decir, el valor de `x` en la función que hace la llamada) durante la vida de la función. Al terminar la función, `temp` ya no se necesitará.

En la línea 27 se asigna a `temp` el valor contenido en `apx`. En la línea 28, ese valor se asigna al valor contenido en `apy`. En la línea 29, la variable guardada en `temp` (es decir, el valor original contenido en `apx`) se coloca en `apy`.

El efecto ocasionado con esto es que los valores de la función que hizo la llamada, cuya dirección se pasó a `intercambiar()`, se intercambiaron realmente.

## Cómo implementar a `intercambiar()` con referencias

El programa anterior funciona, pero la sintaxis de la función `intercambiar()` es incómoda por dos motivos. En primer lugar, la necesidad repetida de desreferenciar los apuntadores dentro de la función `intercambiar()` hace que ésta sea propensa a errores y difícil de leer. En segundo lugar, la necesidad de pasar la dirección de las variables de la función que hace la llamada hace que el funcionamiento interno de `intercambiar()` sea demasiado transparente para sus usuarios.

Uno de los objetivos de C++ es prevenir que el usuario de una función se preocupe por la forma en que ésta funciona. Al pasar los parámetros mediante apuntadores se coloca la carga en la función que hace la llamada, en lugar de colocarla donde pertenece (en la función que es llamada). El listado 9.7 vuelve a escribir la función `intercambiar()`, pero esta vez usando referencias.

**ENTRADA****LISTADO 9.7** intercambiar() reescrita usando referencias

```

1:  //Listado 9.7 Muestra del paso de parámetros por referencia
2:  // iuso de referencias!
3:
4:  #include <iostream.h>
5:
6:  void intercambiar(int & x, int & y);
7:
8:  int main()
9:  {
10:     int x = 5, y = 10;
11:
12:     cout << "Main. Antes del intercambio, ";
13:     cout << "x: " << x << " y: " << y << "\n";
14:     intercambiar(x, y);
15:     cout << "Main. Después del intercambio, ";
16:     cout << "x: " << x << " y: " << y << "\n";
17:     return 0;
18: }
19:
20: void intercambiar (int & rx, int & ry)
21: {
22:     int temp;
23:
24:     cout << "Intercambiar. Antes del intercambio, ";
25:     cout << "rx: " << rx << " ry: " << ry << "\n";
26:
27:     temp = rx;
28:     rx = ry;
29:     ry = temp;
30:
31:     cout << "Intercambiar. Después del intercambio, ";
32:     cout << "rx: " << rx << " ry: " << ry << "\n";
33: }
```

**SALIDA**

```

Main. Antes del intercambio, x: 5 y: 10
Intercambiar. Antes del intercambio, rx: 5 ry: 10
Intercambiar. Después del intercambio, rx: 10 ry: 5
Main. Después del intercambio, x: 10 y: 5
```

**ANÁLISIS**

Igual que en el ejemplo con apuntadores, en la línea 10 se declaran dos variables, y sus valores se imprimen en la línea 13. En la línea 14 se hace una llamada a la función `intercambiar()`, pero observe que se pasan `x` y `y`, no sus direcciones. La función que hace la llamada simplemente pasa las variables.

Al llamar a `intercambiar()`, la ejecución del programa salta a la línea 20, en donde las variables se identifican como referencias. Sus valores se imprimen en la línea 25, pero observe que no se requieren operadores especiales. Éstos son alias para los valores originales y se pueden utilizar como tales.

En las líneas 27 a 29 se intercambian los valores, y luego se imprimen en la línea 32. La ejecución del programa salta de regreso a la función que hizo la llamada, y en la línea 16, dentro de `main()`, se imprimen los valores. Debido a que los parámetros para `intercambiar()` se declaran como referencias, los valores de `main()` se pasan por referencia, y por consecuencia se cambian también en `main()`. Otro método exitoso.

Las referencias proporcionan la conveniencia y facilidad del uso de las variables normales, ¡junto con el poder y la capacidad del paso por referencia que ofrecen los apuntadores!

9

## Comprensión de los encabezados y prototipos de funciones

El listado 9.6 muestra a la función `intercambiar()` utilizando apuntadores, y el listado 9.7 la muestra utilizando referencias. Usar la función que toma referencias es más sencillo, y el código es más fácil de leer, pero ¿cómo sabe la función que hace la llamada si los valores se pasan por referencia o por valor? Como cliente (o usuario) de `intercambiar()`, el programador debe asegurarse de que `intercambiar()` cambie los parámetros.

Éste es otro uso del prototipo de la función. Al examinar los parámetros declarados en el prototipo, el cual se encuentra por lo general en un archivo de encabezado junto con los demás prototipos, el programador sabe que los valores que se pasan a `intercambiar()` se pasan por referencia, y por lo tanto se intercambiarán apropiadamente.

Si `intercambiar()` hubiera sido una función miembro de una clase, la declaración de la clase, que también se encuentra disponible en un archivo de encabezado, hubiera proporcionado esta información.

En C++, los clientes de las clases y funciones dependen del archivo de encabezado para que les indique todo lo necesario; este archivo actúa como la interfaz para la clase o función. La implementación real está oculta para el cliente. Esto permite que el programador se enfoque en el problema en cuestión y utilice la clase o función sin preocuparse por la forma en que funciona.

Cuando el coronel John Roebling diseñó el puente de Brooklyn, se preocupó mucho por la forma en que se vertió el cemento y por la forma en que se fabricó el cable para el puente. Estaba íntimamente involucrado en los procesos mecánicos y químicos requeridos para crear sus materiales. No obstante, en la actualidad los ingenieros aprovechan su tiempo en forma más eficiente al utilizar materiales bien conocidos, sin importar la forma en que sus fabricantes los hayan producido.

El objetivo de C++ es permitir que los programadores dependan de clases y funciones bien conocidas sin importar su funcionamiento interno. Estos "componentes" se pueden ensamblar para producir un programa, lo que es muy parecido a la forma en que se ensamblan cables, tuberías, abrazaderas y otros componentes para construir edificios y puentes.

Así como un ingeniero examina la hoja de especificaciones de una tubería para determinar su capacidad de soporte de flujo, el tamaño de las uniones, etc., un programador de C++ lee la interfaz de la función o de la clase para determinar los servicios que ésta proporciona, los parámetros que necesita y los valores que regresa.

Esto también se conoce como *método de la caja negra*. Usted sabe cuáles son las entradas de la caja negra, tiene una idea general de lo que hace, y obtiene las salidas esperadas. Pero no tiene idea de cómo ocurrió esto porque no puede ver su funcionamiento interno (ni debe importarle, pues hay demasiadas cosas más por las que debe preocuparse).

Un ejemplo sencillo de esto es la función `sqrt()` que se encuentra en la biblioteca estándar. Usted sabe que toma un argumento de punto flotante y regresa la raíz cuadrada de ese argumento. Conoce la interfaz (entrada y salida), conoce los detalles generales (calcula la raíz cuadrada), pero no conoce los detalles internos de cómo se realiza ese cálculo.

## Regreso de varios valores por medio de apuntadores

Como se dijo anteriormente, las funciones sólo pueden regresar un valor. ¿Qué pasa si necesita obtener dos valores de una función? Una forma de solucionar este problema es pasar por referencia dos objetos a la función. Entonces la función puede llenar los objetos con los valores correctos. Como el paso por referencia permite que una función cambie los objetos originales, esto efectivamente permite que la función regrese dos piezas de información. Este método no utiliza el valor de retorno de la función, el cual se puede reservar entonces para reportar errores.

De nuevo, esto se puede hacer con referencias o con apuntadores. El listado 9.8 muestra una función que regresa tres valores: dos como parámetros apuntadores y uno como el valor de retorno de la función.

**ENTRADA****LISTADO 9.8** Regreso de valores con apuntadores

---

```
1:  //Listado 9.8 Regreso de varios valores de
2:  // una función por medio de apuntadores
3:
4:  #include <iostream.h>
5:
6:  short Factor(int n, int * apAlCuadrado, int * apAlCubo);
7:
8:  int main()
9:  {
10:    int numero, alcuadrado, alcubo;
11:    short error;
12:
13:    cout << "Escriba un número (0 - 20): ";
14:    cin >> numero;
```

```
15:         error = Factor(numero, &alcuadrado, &alcubo);
16:     if (!error)
17:     {
18:         cout << "número: " << numero << "\n";
19:         cout << "al cuadrado: " << alcuadrado << "\n";
20:         cout << "al cubo: " << alcubo << "\n";
21:     }
22:     else
23:         cout << "¡Se encontró un error!!\n";
24:     return 0;
25: }
26: }
27:
28: short Factor(int n, int * apAlCuadrado, int * apAlCubo)
29: {
30:     short Valor = 0;
31:
32:     if (n > 20)
33:         Valor = 1;
34:     else
35:     {
36:         *apAlCuadrado = n * n;
37:         *apAlCubo = n * n * n;
38:         Valor = 0;
39:     }
40:     return Valor;
41: }
```

**SALIDA**

Escriba un número (0 - 20): 3  
número: 3  
al cuadrado: 9  
al cubo: 27

**ANÁLISIS**

En la línea 10, `numero`, `alcuadrado` y `alcubo` se definen como valores de tipo `int`. A `numero` se le asigna un valor basado en la entrada que proporcione el usuario. Este valor y las direcciones de `alcuadrado` y `alcubo` se pasan a la función `Factor()`.

`Factor()` examina el primer parámetro, el cual se pasa por medio de `Valor`. Si es mayor que 20 (el valor máximo que esta función puede manejar), asigna a `Valor` un 1 como valor de error. Observe que el valor de retorno de `Factor()` se reserva ya sea para este valor de error o para el valor `0`, lo cual indica que todo salió bien, y observe que la función regresa este valor en la línea 40.

Los valores que realmente se necesitan, el cuadrado y el cubo de `numero`, no se regresan por medio del mecanismo de retorno, sino cambiando las variables originales (cuyas direcciones están contenidas en los apuntadores que se pasaron a la función).

En las líneas 36 y 37 se les asignan a las variables originales (referenciadas mediante apuntadores) sus valores de retorno. En la línea 38 se asigna un cero a `Valor` como valor de éxito. En la línea 40 se regresa ese `Valor`.

Una mejora a este programa podría ser la siguiente declaración:

```
enum CODIGO_ERR { EXITO, ERROR};
```

Así, en lugar de regresar 0 o 1, el programa podría regresar EXITO o ERROR.

## Regreso de valores por referencia

Aunque el listado 9.8 funciona, se puede facilitar su legibilidad y mantenimiento si se utilizan referencias en lugar de apuntadores. El listado 9.9 muestra el mismo programa reescrito para utilizar referencias e incorporar la enumeración llamada `CODIGO_ERR`.

**ENTRADA****LISTADO 9.9** Listado 9.8 modificado, esta vez para utilizar referencias

---

```
1: //Listado 9.9 Regreso de varios valores de
2: // una función por medio de referencias
3:
4: #include <iostream.h>
5:
6: typedef unsigned short USHORT;
7: enum CODIGO_ERR { EXITO, ERROR };
8:
9: CODIGO_ERR Factor(USHORT, USHORT &, USHORT &);
10:
11: int main()
12: {
13:     USHORT numero, alcuadrado, alcubo;
14:     CODIGO_ERR resultado;
15:
16:     cout << "Escriba un número (0 - 20): ";
17:     cin >> numero;
18:
19:     resultado = Factor(numero, alcuadrado, alcubo);
20:
21:     if (resultado == EXITO)
22:     {
23:         cout << "número: " << numero << "\n";
24:         cout << "al cuadrado: " << alcuadrado << "\n";
25:         cout << "al cubo: " << alcubo << "\n";
26:     }
27:     else
28:         cout << "¡Se encontró un error!!\n";
29:     return 0;
30: }
```

```
31:  
32:     CODIGO_ERR Factor(USHORT n, USHORT &rAlCuadrado, USHORT &rAlCubo)  
33:     {  
34:         if (n > 20)  
35:             return ERROR; // código simple de error  
36:         else  
37:             {  
38:                 rAlCuadrado = n * n;  
39:                 rAlCubo = n * n * n;  
40:                 return EXITO;  
41:             }  
42:     }
```

9

**SALIDA**

```
Escriba un número (0 - 20): 3  
número: 3  
al cuadrado: 9  
al cubo: 27
```

**ANÁLISIS**

El listado 9.9 es idéntico al listado 9.8, con dos excepciones. La enumeración `CODIGO_ERR` hace que el reporte de errores sea más explícito en las líneas 35 y 40, así como el manejo de errores en la línea 21.

Sin embargo, el cambio más importante es que ahora `Factor()` se declara para tomar referencias a `alcuadrado` y `alcubo` en vez de apunadores. Esto hace que la manipulación de estos parámetros sea más sencilla y fácil de entender.

## Cómo pasar parámetros por referencia para tener eficiencia

Cada vez que le pasa un objeto por valor a una función, se crea una copia del objeto. Cada vez que regresa un objeto por valor de una función, se crea otra copia.

En el día 5 aprendió que estos objetos se copian en la pila. Hacer esto toma tiempo y ocupa memoria. Para objetos pequeños, como los tipos enteros integrados, esto es un costo trivial.

Sin embargo, con objetos más grandes creados por el usuario, el costo es mayor. El tamaño de un objeto creado por el usuario en la pila es la suma de cada una de sus variables miembro. Éstas, a su vez, pueden ser objetos creados por el usuario, y pasar toda esa estructura masiva copiándola en la pila puede provocar una reducción bastante considerable en el rendimiento y un consumo excesivo de memoria.

También hay otro costo. Con las clases que se crean, cada una de estas copias temporales se crea cuando el compilador llama a un constructor especial: el constructor de copia. Mañana aprenderá cómo funcionan los constructores de copia y cómo hacer sus propios constructores, pero por ahora basta con saber que el constructor de copia se llama cada vez que se coloca en la pila una copia temporal del objeto.

Cuando se destruye el objeto temporal, lo que ocurre cuando una función termina, se llama al destructor del objeto. Si un objeto se regresa de la función por valor, se debe crear y destruir también una copia de ese objeto.

Con objetos grandes, estas llamadas al constructor y al destructor pueden disminuir la velocidad y aumentar el uso de la memoria. Para ilustrar esta idea, el listado 9.10 crea un objeto simplificado creado por el usuario: `GatoSimple`. Un objeto real podría ser más grande y costoso, pero esto es suficiente para mostrar con qué frecuencia se llama al constructor y al destructor de copia.

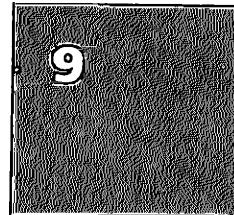
El listado 9.10 crea el objeto `GatoSimple` y luego llama dos funciones. La primera función recibe al objeto `GatoSimple` por valor y luego lo regresa por valor. La segunda recibe un apuntador al objeto, en lugar del objeto en sí, y regresa un apuntador al objeto.

**ENTRADA** **LISTADO 9.10** Paso de objetos por referencia mediante el uso de apuntadores

---

```
1: //Listado 9.10
2: // Paso de apuntadores a objetos
3:
4: #include <iostream.h>
5:
6: class GatoSimple
7: {
8: public:
9:     GatoSimple ();                                // constructor
10:    GatoSimple(GatoSimple &); // constructor de copia
11:    -GatoSimple();                                // destructor
12: };
13:
14: GatoSimple::GatoSimple()
15: {
16:     cout << "Constructor de GatoSimple...\n";
17: }
18:
19: GatoSimple::GatoSimple(GatoSimple &)
20: {
21:     cout << "Constructor de copia de GatoSimple...\n";
22: }
23:
24: GatoSimple::~GatoSimple()
25: {
26:     cout << "Destructor de GatoSimple...\n";
27: }
28:
29:     GatoSimple FuncionUno(GatoSimple elGato);
30:     GatoSimple * FuncionDos(GatoSimple * elGato);
31:
32: int main()
33: {
```

```
34:         cout << "Crear un gato...\n";
35:         GatoSimple Pelusa;
36:         cout << "Llamando a FuncionUno...\n";
37:         FuncionUno(Pelusa);
38:         cout << "Llamando a FuncionDos...\n";
39:         FuncionDos(&Pelusa);
40:     return 0;
41: }
42:
43: // FuncionUno, pasa por valor
44: GatoSimple FuncionUno(GatoSimple elGato)
45: {
46:         cout << "FuncionUno. Regresando...\n";
47:         return elGato;
48: }
49:
50: // FuncionDos, pasa por referencia
51: GatoSimple * FuncionDos(GatoSimple * elGato)
52: {
53:         cout << "FuncionDos. Regresando...\n";
54:         return elGato;
55: }
```

**SALIDA**

```
1: Crear un gato...
2: Constructor de GatoSimple...
3: Llamando a FuncionUno...
4: Constructor de copia de GatoSimple...
5: FuncionUno. Regresando...
6: Constructor de copia de GatoSimple...
7: Destructor de GatoSimple...
8: Destructor de GatoSimple...
9: Llamando a FuncionDos...
10: FuncionDos. Regresando...
11: Destructor de GatoSimple...
```

**Nota**

Los números de línea mostrados en la salida anterior no se imprimirán. Se agregaron como ayuda para el análisis.

**ANÁLISIS**

En las líneas 6 a 12 se declara una clase `GatoSimple` muy simplificada. El constructor, el constructor de copia y el destructor imprimen cada uno un mensaje informativo para que podamos saber cuándo son llamados.

En la línea 34, `main()` imprime un mensaje que se ve en la línea 1 de la salida. En la línea 35 se crea una instancia de un objeto de la clase `GatoSimple`. Esto ocasiona que se llame al constructor, y la salida producida por el constructor se ve en la línea 2 de la salida.

En la línea 36, `main()` informa que está llamando a `FuncionUno`, la cual crea la línea 3 de la salida. Debido a que al llamar a `FuncionUno()` se le pasa el objeto `GatoSimple` por valor, se crea una copia de este objeto en la pila como un objeto local para la función llamada. Esto ocasiona que se llame al constructor de copia, el cual crea la línea 4 de la salida.

La ejecución del programa salta hasta la línea 46 en la función llamada, la cual imprime un mensaje informativo, el de la línea 5 de la salida. La función regresa entonces el objeto `GatoSimple` por valor. Esto crea otra copia del objeto, se llama al constructor de copia y se produce la línea 6 de la salida.

El valor de retorno de `FuncionUno()` no se asigna a ningún objeto, por lo que se descarta el valor temporal creado para el valor de retorno, lo que ocasiona que se llame al destructor, el cual produce la línea 7 de la salida. Como la `FuncionUno()` ha terminado, su copia local queda fuera de alcance y se destruye, por lo que se llama al destructor y se produce la línea 8 de la salida.

La ejecución del programa regresa a `main()`, y se llama a `FuncionDos()`, pero se pasa el parámetro por referencia. No se produce copia, por lo que no hay salida. `FuncionDos()` imprime el mensaje que aparece en la línea 10 de la salida y luego regresa el objeto `GatoSimple`, de nuevo por referencia, por lo que no se produce ninguna llamada al constructor ni al destructor.

Finalmente, el programa termina y `Pelusa` queda fuera de alcance, lo que produce una última llamada al destructor y se imprime la línea 11 de la salida.

El efecto ocasionado con esto es que la llamada a `FuncionUno()`, debido a que pasó el objeto gato por valor, produjo dos llamadas al constructor de copia y dos al destructor, mientras que la llamada a `FuncionDos()` no produjo ninguna.

## Paso de un apuntador `const`

Aunque pasar un apuntador a `FuncionDos()` es más eficiente, es peligroso. No se debe permitir que `FuncionDos()` cambie el objeto `GatoSimple` que recibe, pero aún así se le proporciona la dirección de `GatoSimple`. Esto expone seriamente al objeto a ser cambiado y viola la protección que se ofrece al pasar parámetros por valor.

Pasar por valor es como dar a un museo una fotografía de su obra maestra en lugar de darle la verdadera. Si los vándalos la rayan, no se daña la original. Pasar por referencia es como enviar su dirección particular al museo y hacer que vengan invitados y vean la obra verdadera.

La solución es pasar un apuntador a un objeto `GatoSimple` constante. Hacer esto evita que se llame a cualquier método que no sea constante en `GatoSimple`, y por consecuencia evita que el objeto sea cambiado. El listado 9.11 muestra esta idea.

**ENTRADA LISTADO 9.11** Paso de apunadores const

```
1: //Listado 9.11
2:      // Paso de apunadores const a objetos
3:
4:      #include <iostream.h>
5:
6:      class GatoSimple
7:      {
8:      public:
9:          GatoSimple();
10:         GatoSimple(GatoSimple &);
11:         ~GatoSimple();
12:         int ObtenerEdad() const
13:             { return suEdad; }
14:         void AsignarEdad(int edad)
15:             { suEdad = edad; }
16:      private:
17:          int suEdad;
18:      };
19:
20:      GatoSimple::GatoSimple()
21:      {
22:          cout << "Constructor de GatoSimple...\n";
23:          suEdad = 1;
24:      }
25:
26:      GatoSimple::GatoSimple(GatoSimple &)
27:      {
28:          cout << "Constructor de copia de GatoSimple...\n";
29:      }
30:
31:      GatoSimple::~GatoSimple()
32:      {
33:          cout << "Destructor de GatoSimple...\n";
34:      }
35:
36: const GatoSimple * const FuncionDos(const GatoSimple * const elGato);
37:
38:     int main()
39:     {
40:         cout << "Crear un gato...\n";
41:         GatoSimple Pelusa;
42:         cout << "Pelusa tiene " ;
43:         cout << Pelusa.ObtenerEdad();
44:         cout << " años de edad\n";
45:         int edad = 5;
46:         Pelusa.AsignarEdad(edad);
47:         cout << "Pelusa tiene " ;
48:         cout << Pelusa.ObtenerEdad();
49:         cout << " años de edad\n";
```

**LISTADO 9.11** CONTINUACIÓN

```

50:           cout << "Llamando a FuncionDos...\n";
51:           FuncionDos(&Pelusa);
52:           cout << "Pelusa tiene " ;
53:           cout << Pelusa.ObtenerEdad();
54:           cout << " años de edad\n";
55:       return 0;
56:   }
57:
58: // FuncionDos, pasa un apuntador const
59: const GatoSimple * const FuncionDos(const GatoSimple * const elGato)
60: {
61:     cout << "FuncionDos. Regresando...\n";
62:     cout << "Ahora Pelusa tiene " << elGato->ObtenerEdad();
63:     cout << " años de edad \n";
64:     // elGato->AsignarEdad(8); iconst!
65:     return elGato;
66: }
```

**SALIDA**

```

Crear un gato...
Constructor de GatoSimple...
Pelusa tiene 1 años de edad
Pelusa tiene 5 años de edad
Llamando a FuncionDos...
FuncionDos. Regresando...
Ahora Pelusa tiene 5 años de edad
Pelusa tiene 5 años de edad
Destructor de GatoSimple...
```

**ANÁLISIS** GatoSimple ha agregado dos funciones de acceso, ObtenerEdad() en la línea 12, la cual es una función const, y AsignarEdad() en la línea 14, la cual no es una función const. También ha agregado la variable miembro suEdad en la línea 17.

El constructor, el constructor de copia y el destructor aún se definen para imprimir sus mensajes. Sin embargo, nunca se llama al constructor de copia, ya que el objeto se pasa por referencia y no se crea ninguna copia. En la línea 41 se crea un objeto y se imprime su edad predeterminada, empezando en la línea 42.

En la línea 46 se asigna un valor a Pelusa por medio de la función de acceso AsignarEdad, y el resultado se imprime en la línea 47. En este programa no se utiliza FuncionUno, pero sí se llama a FuncionDos(). Esta función ha cambiado ligeramente; el parámetro y el valor de retorno ahora se declaran, en la línea 36, para recibir un apuntador constante a un objeto constante y para regresar un apuntador constante a un objeto constante.

Como el parámetro y el valor de retorno se siguen pasando por referencia, no se crean copias y no se llama al constructor de copia. Sin embargo, ahora el apuntador de FuncionDos() es constante, y no puede llamar al método no constante llamado AsignarEdad(). Si la llamada a AsignarEdad() de la línea 64 no se hubiera convertido en comentario, el com-

pilador enviaría un mensaje de advertencia y eliminaría la calidad de constante del objeto GATO (con algunas versiones del compilador GNU, este programa no compilará).

Observe que el objeto creado en `main()` no es constante, y Pelusa puede llamar a `AsignarEdad()`. La dirección de este objeto no constante se pasa a `FuncionDos()`, pero como la declaración de `FuncionDos()` declara al apuntador como apuntador constante a un objeto constante, ¡el objeto se trata como si fuera constante!

9

## Referencias como alternativa para los apuntadores

El listado 9.11 soluciona el problema de crear copias adicionales, y por consecuencia evita las llamadas al constructor y al destructor de copia. Utiliza apuntadores constantes a objetos constantes, y por lo tanto resuelve el problema de que la función cambie el objeto. Sin embargo, todavía es algo incómodo debido a que los objetos que se pasan a la función son apuntadores.

Debido a que usted sabe que el objeto nunca será nulo, sería más sencillo trabajar con la función si se pasara una referencia, en lugar de un apuntador. El listado 9.12 muestra esto.

### ENTRADA LISTADO 9.12 Paso de referencias a objetos

```
1: //Listado 9.12
2: // Paso de referencias a objetos
3:
4: #include <iostream.h>
5:
6: class GatoSimple
7: {
8: public:
9:     GatoSimple();
10:    GatoSimple(GatoSimple &);
11:    ~GatoSimple();
12:    int ObtenerEdad() const
13:    { return suEdad; }
14:    void AsignarEdad(int edad)
15:    { suEdad = edad; }
16: private:
17:     int suEdad;
18: };
19:
20: GatoSimple::GatoSimple()
21: {
22:     cout << "Constructor de GatoSimple...\n";
23:     suEdad = 1;
24: }
```

continua

**LISTADO 9.12** CONTINUACIÓN

```
26:     GatoSimple::GatoSimple(GatoSimple &)
27:     {
28:         cout << "Constructor de copia de GatoSimple...\n";
29:     }
30:
31:     GatoSimple::~GatoSimple()
32:     {
33:         cout << "Destructor de GatoSimple...\n";
34:     }
35:
36:     const GatoSimple & FuncionDos(const GatoSimple & elGato);
37:
38:     int main()
39:     {
40:         cout << "Crear un gato...\n";
41:         GatoSimple Pelusa;
42:         cout << "Pelusa tiene " << Pelusa.ObtenerEdad();
43:         cout << " años de edad\n";
44:         int edad = 5;
45:         Pelusa.AsignarEdad(edad);
46:         cout << "Pelusa tiene " << Pelusa.ObtenerEdad();
47:         cout << " años de edad\n";
48:         cout << "Llamando a FuncionDos...\n";
49:         FuncionDos(Pelusa);
50:         cout << "Pelusa tiene " << Pelusa.ObtenerEdad();
51:         cout << " años de edad\n";
52:     return 0;
53:     }
54:
55: // FuncionDos, pasa una referencia a un objeto const
56: const GatoSimple & FuncionDos(const GatoSimple & elGato)
57: {
58:         cout << "FuncionDos. Regresando...\n";
59:         cout << "Ahora Pelusa tiene " << elGato.ObtenerEdad();
60:         cout << " años de edad \n";
61:         // elGato.AsignarEdad(8); iconst!
62:         return elGato;
63: }
```

**SALIDA**

Crear un gato...  
Constructor de GatoSimple...  
Pelusa tiene 1 años de edad  
Pelusa tiene 5 años de edad  
Llamando a FuncionDos...  
FuncionDos. Regresando...  
Ahora Pelusa tiene 5 años de edad  
Pelusa tiene 5 años de edad  
Destructor de GatoSimple...

**ANÁLISIS**

La salida es idéntica a la producida por el listado 9.11. El único cambio significativo es que ahora FuncionDos() toma y regresa una referencia a un objeto constante. De nuevo, trabajar con referencias es un poco más sencillo que trabajar con apuntadores, y se logran los mismos ahorros y eficiencia, además de la seguridad proporcionada al usar const.

9

**Referencias const**

Por lo general, los programadores de C++ no hacen diferencias entre "referencia constante a un objeto GatoSimple" y "referencia a un objeto GatoSimple constante". Las referencias no se pueden reasignar para referirse a otro objeto, por lo que siempre son constantes. Si se aplica la palabra reservada const a una referencia, es para hacer que el objeto al que se refiere sea constante.

## Cuándo utilizar referencias y cuándo utilizar apuntadores

Los programadores de C++ prefieren más las referencias que los apuntadores. Las referencias son más limpias y fáciles de utilizar, y realizan mejor el trabajo de ocultar la información, como vio en el ejemplo anterior.

Sin embargo, las referencias no se pueden reasignar. Si necesita apuntar primero a un objeto y luego a otro, debe utilizar un apuntador. Las referencias no pueden ser nulas, por lo que si existe alguna probabilidad de que el objeto en cuestión pueda ser nulo, no debe utilizar una referencia. Debe usar un apuntador.

Un ejemplo de lo anterior es el operador new. Si new no puede asignar memoria en el heap, regresa un apuntador nulo. Debido a que una referencia no puede ser nula, usted sólo puede inicializar una referencia a este segmento de memoria hasta que compruebe que no es nula. El siguiente ejemplo muestra cómo manejar esto:

```
int * apInt = new int;
if (apInt != NULL)
    int & rInt = *apInt;
```

En este ejemplo se declara un apuntador a un entero llamado apInt, y se inicializa con la memoria regresada por el operador new. Se prueba la dirección contenida en apInt, y si no es nula, apInt es desreferenciado. El resultado de desreferenciar una variable de tipo int es un objeto de tipo int, y rInt se inicializa como referencia a ese objeto. Por lo tanto, rInt se convierte en un alias para el entero regresado por el operador new.

| DEBE                                                                                                | NO DEBE                                                            |
|-----------------------------------------------------------------------------------------------------|--------------------------------------------------------------------|
| <b>DEBE</b> pasar parámetros por referencia siempre que sea posible.                                | <b>NO DEBE</b> utilizar apuntadores si puede utilizar referencias. |
| <b>DEBE</b> regresar parámetros por referencia siempre que sea posible.                             | <b>NO DEBE</b> regresar una referencia a un objeto local.          |
| <b>DEBE</b> utilizar const para proteger las referencias y los apuntadores siempre que sea posible. |                                                                    |

## Cómo mezclar referencias y apuntadores

Es perfectamente válido declarar tanto apuntadores como referencias en la misma lista de parámetros de la función, así como objetos pasados por valor. He aquí un ejemplo:

```
GATO * UnaFuncion(Persona & elPropietario, Casa * laCasa, int edad);
```

Esta declaración indica que UnaFuncion toma tres parámetros. El primero es una referencia a un objeto llamado Persona, el segundo es un apuntador a un objeto llamado Casa, y el tercero es un entero. La función regresa un apuntador a un objeto GATO.

La cuestión de dónde colocar el operador de referencia (&) o el de indirección (\*) al declarar las variables causa gran controversia. Puede escribir válidamente cualquiera de las siguientes instrucciones:

- 1: GATO& rPelusa;
- 2: GATO & rPelusa;
- 3: GATO &rPelusa;

### Nota

El espacio en blanco se ignora por completo, por lo que puede colocar tantos espacios, tabuladores y nuevas líneas como desee en cualquier lugar en el que vea un espacio.

Dejando a un lado la libertad de cuestiones relacionadas con la expresión, ¿cuál es mejor? He aquí argumentos para las tres:

El argumento para el caso 1 es que rPelusa es una variable cuyo nombre es rPelusa y cuyo tipo se puede considerar como una "referencia a un objeto de la clase GATO". Por lo tanto, como este argumento indica, el & debe ir con el tipo.

El argumento contrario es que la clase es GATO. El & es parte del "declarador", el cual incluye el nombre de la variable y el símbolo &. Lo que es más importante, tener el símbolo & cerca de GATO puede provocar el siguiente error:

```
GATO& rPelusa, rSlvestre;
```

Un análisis casual de esta línea le conduciría a pensar que tanto `rPelusa` como `rSilvestre` son referencias a objetos de clase `GATO`, pero estaría equivocado. Lo que esto realmente dice es que `rPelusa` es una referencia a `GATO`, y `rSilvestre` (a pesar de su nombre) no es una referencia, sino una simple variable de la clase `GATO`. Esto se debe reescribir de la siguiente manera:

```
GATO      &rPelusa, rSilvestre;
```

La respuesta a esta objeción es que las declaraciones de referencias y variables nunca se deben combinar así. La respuesta correcta es la siguiente:

```
GATO& rPelusa;  
GATO  Silvestre;
```

Finalmente, muchos programadores optan por evadir el argumento y utilizan la posición media, la de colocar el `&` en medio de los dos, como se muestra en el caso 2.

Desde luego que todo lo que se ha dicho acerca del operador de referencia (`&`) se aplica de igual forma al operador de indirección (`*`). Lo importante es reconocer que personas razonables difieren en sus percepciones acerca del método único y verdadero. Elija un estilo que funcione para usted, y sea consistente dentro de cualquier programa; la claridad es, y sigue siendo, el objetivo.

Muchos programadores prefieren las siguientes convenciones para declarar referencias y apuntadores:

1. Colocar el `&` y el `*` en medio, con un espacio en cada lado.
2. Nunca declarar referencias, apuntadores y variables juntos en la misma línea.

9

## ¡No regrese una referencia a un objeto que esté fuera de alcance!

Cuando los programadores de C++ aprenden a pasar parámetros por referencia, tienen una tendencia a utilizarlos mucho. Sin embargo, es posible que los utilicen más de la cuenta. Recuerde que una referencia siempre es un alias para algún otro objeto. Si pasa una referencia hacia o desde una función, pregúntese: “¿Cuál es el objeto que estoy pasando? y ¿Seguirá existiendo cada vez que lo utilice?”

El listado 9.13 muestra el peligro de regresar una referencia a un objeto que ya no exista.

### ENTRADA

### LISTADO 9.13 Regreso de una referencia a un objeto inexistente

```
1: // Listado 9.13 Regreso de una referencia  
2: // a un objeto que ya no existe
```

continua

**LISTADO 9.13** CONTINUACIÓN

```
3:  
4: #include <iostream.h>  
5:  
6: class GatoSimple  
7: {  
8: public:  
9:     GatoSimple(int edad, int peso);  
10:    ~GatoSimple() {}  
11:    int ObtenerEdad()  
12:        { return suEdad; }  
13:    int ObtenerPeso()  
14:        { return suPeso; }  
15: private:  
16:    int suEdad;  
17:    int suPeso;  
18: };  
19:  
20: GatoSimple::GatoSimple(int edad, int peso)  
21: {  
22:     suEdad = edad;  
23:     suPeso = peso;  
24: }  
25:  
26: GatoSimple & LaFuncion();  
27:  
28: int main()  
29: {  
30:     GatoSimple & rGato = LaFuncion();  
31:     int edad = rGato.ObtenerEdad();  
32:     cout << "irGato tiene " << edad << " años de edad!\n";  
33:     return 0;  
34: }  
35:  
36: GatoSimple & LaFuncion()  
37: {  
38:     GatoSimple Pelusa(5, 9);  
39:     return Pelusa;  
40: }
```

**SALIDA**

irGato tiene -1073743376 años de edad!

 **Precaución**

Este programa no se compilará en los compiladores GNU 2.7.2 y anteriores.  
En las versiones 2.91.66 y 2.96 recibirá el mensaje de error siguiente:

1st09-13.cxx: In function `GatoSimple &LaFuncion ()':

1st09-13.cxx:38: warning: reference to local variable `Pelusa' returned

Sin embargo, usted tendrá un programa ejecutable que no hará lo correcto.

Aún más, en la versión 2.91.66 la salida del programa será:

irGato tiene 5 años de edad!

A pesar de esto, este programa tiene fugas de memoria y tendrá un comportamiento impredecible. Tampoco se compilará en el compilador de Borland, pero si lo hará en los compiladores de Microsoft.

Regresar una referencia a una variable local es una mala práctica de codificación.

9

**ANÁLISIS** En las líneas 6 a 18 se declara la clase `GatoSimple`. En la línea 30 se inicializa una referencia a un `GatoSimple` con los resultados obtenidos al llamar a `LaFuncion()`, la cual se declara en la línea 26 para regresar una referencia a un `GatoSimple`.

El cuerpo de `LaFuncion()` crea un objeto local de la clase `GatoSimple` e inicializa su edad y peso. Luego regresa por referencia ese objeto local. Algunos compiladores lo dejarán ejecutar el programa, pero los resultados serán impredecibles.

Cuando `LaFuncion()` regrese, el objeto local llamado `Pelusa` será destruido (sin dolor, se lo aseguro). La referencia regresada por esta función será un alias para un objeto inexistente, y esto es algo malo.

## Cómo regresar una referencia a un objeto en el heap

Podría estar tentado a solucionar el problema del listado 9.13 haciendo que `LaFuncion()` cree a `Pelusa` en el heap. De esa forma, cuando usted regrese de `LaFuncion()`, `Pelusa` aún existirá.

El problema de este método es: ¿Qué se hace con la memoria asignada para `Pelusa` al terminar de utilizarlo? El listado 9.14 ejemplifica este problema.

### ENTRADA LISTADO 9.14 Fugas de memoria

```
1: // Listado 9.14
2: // Solución para fugas de memoria
3:
4: #include <iostream.h>
5:
6: class GatoSimple
7: {
8: public:
9:     GatoSimple (int edad, int peso);
10:    ~GatoSimple() {}
11:    int ObtenerEdad()
12:        { return suEdad; }
13:    int ObtenerPeso()
14:        { return suPeso; }
15: private:
16:     int suEdad;
```

*continúa*

**LISTADO 9.14** CONTINUACIÓN

```

17:     int suPeso;
18: };
19:
20: GatoSimple::GatoSimple(int edad, int peso)
21: {
22:     suEdad = edad;
23:     suPeso = peso;
24: }
25:
26: GatoSimple & LaFuncion();
27:
28: int main()
29: {
30:     GatoSimple & rGato = LaFuncion();
31:     int edad = rGato.ObtenerEdad();
32:     cout << "irGato tiene " << edad << " años de edad!\n";
33:     cout << "&rGato: " << &rGato << endl;
34:     // ¿Cómo nos deshacemos de esa memoria?
35:     GatoSimple * apGato = &rGato;
36:     delete apGato;
37:     // Hmmmm?, ¿a quién se refiere ahora rGato??
38:     return 0;
39: }
40:
41: GatoSimple & LaFuncion()
42: {
43:     GatoSimple * apPelusa = new GatoSimple(5, 9);
44:     cout << "apPelusa: " << apPelusa << endl;
45:     return *apPelusa;
46: }

```

**SALIDA**

apPelusa: 0x8049b88  
 irGato tiene 5 años de edad!  
 &rGato: 0x8049b88

**Precaución**

El código mostrado en el listado 9.14 se compila, se enlaza y parece funcionar perfectamente. Pero es una bomba de tiempo esperando activarse ya que hay una fuga de memoria oculta.

**ANÁLISIS**

LaFuncion() se ha cambiado de forma que ya no regrese una referencia a una variable local. La memoria se reserva en el heap y se asigna a un apuntador en la línea 43. Se imprime la dirección que guarda ese apuntador, y luego el apuntador es desreferenciado y el objeto GatoSimple se regresa por referencia.

En la línea 30, el valor de retorno de LaFuncion() se asigna a una referencia a GatoSimple, y ese objeto se utiliza para obtener la edad del gato, la cual se imprime en la línea 32.

Para probar que la referencia declarada en `main()` está haciendo referencia al objeto de `LaFuncion()` colocado en el heap, se aplica el operador de dirección a `rGato`. Ciertamente muestra la dirección del objeto al que está haciendo referencia, y esto concuerda con la dirección del objeto que se encuentra en el heap.

Hasta ahora todo está bien. Pero, ¿cuánta memoria se liberará? No se puede usar `delete` sobre la referencia. Una solución inteligente es crear otro apuntador e inicializarlo con la dirección obtenida de `rGato`. Esto sí elimina la memoria y tapa la fuga de memoria. Sin embargo, hay un pequeño problema: ¿A quién hace referencia `rGato` después de la línea 36? Como se dijo anteriormente, una referencia siempre debe ser un alias para un objeto actual; si hace referencia a un objeto nulo (como lo hace ahora), el programa no es válido.

9

### Nota

Nunca está de más recalcar que un programa que tenga una referencia a un objeto nulo tal vez se pueda compilar, pero no es válido y su desempeño será impredecible.

Existen tres soluciones para este problema. La primera es declarar un objeto `GatoSimple` en la línea 30 y regresar por valor ese gato de `LaFuncion()`. La segunda es declarar el `GatoSimple` en el heap desde `LaFuncion()`, pero haciendo que `LaFuncion()` regrese un apuntador a esa memoria. Entonces la función que hace la llamada puede eliminar el apuntador cuando deje de utilizarlo.

La tercera solución funcional, que es la correcta, es declarar el objeto en la función que hace la llamada y luego pasarlo por referencia a `LaFuncion()`.

## ¿A quién pertenece el apuntador?

Cuando un programa asigna memoria en heap, se regresa un apuntador. Es imperativo que mantenga un apuntador a esa memoria, ya que si pierde el apuntador, no puede liberar la memoria y se convierte en una fuga de memoria.

Este bloque de memoria se pasa entre funciones, y alguien tiene que ser el "dueño" del apuntador. Por lo general, el valor que se encuentra en el bloque se pasará por medio de referencias, y la función que creó la memoria es la que la elimina. Pero ésta es una regla general, no es rigurosa.

Es peligroso que una función cree memoria y que otra la libere. La imprecisión sobre quién es dueño del apuntador puede conducir a uno de dos problemas: olvidar eliminar un apuntador o eliminarlo dos veces. Cualquiera de las dos situaciones puede ocasionar graves problemas en el programa. Es más seguro crear las funciones de forma que liberan la memoria que reservan.

Si está escribiendo una función que necesita reservar memoria y luego pasársela de regreso a la función que hizo la llamada, considere cambiar su interfaz. Haga que la función que hace la llamada asigne la memoria y luego pásela por referencia a su función. Esto deja fuera toda la administración de memoria de su programa y la pasa a la función que está preparada para eliminarla.

| DEBE                                                                                                                                       | NO DEBE                                                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>DEBE</b> pasar parámetros por valor cuando sea necesario.</p> <p><b>DEBE</b> regresar parámetros por valor cuando sea necesario.</p> | <p><b>NO DEBE</b> pasar por referencia si el elemento al que se hace referencia puede quedar fuera de alcance.</p> <p><b>NO DEBE</b> utilizar referencias a objetos nulos.</p> |

## Resumen

Hoy aprendió lo que son las referencias y cómo se comparan con los apuntadores. También vio que debe inicializar las referencias para referirse a un objeto existente, y que no las puede asignar a ninguna otra cosa. Cualquier acción realizada sobre una referencia en realidad afecta al objeto destino de la referencia. La prueba de esto es que si se toma la dirección de una referencia, se regresa la dirección del destino.

Vio que puede ser más eficiente pasar objetos por referencia que pasárselos por valor. Pasar por referencia también permite que la función llamada cambie el valor de los argumentos de la función que hace la llamada.

También vio que los argumentos de las funciones y los valores regresados de las funciones se pueden pasar por referencia, y que esto se puede implementar con apuntadores o con referencias.

Vio cómo utilizar apuntadores a objetos constantes y referencias constantes para pasar con seguridad valores entre funciones, al tiempo que logra la eficiencia que se obtiene al pasar por referencia.

## Preguntas y respuestas

**P** ¿Por qué tener referencias si los apuntadores pueden hacer todo lo que hacen las referencias?

**R** Las referencias son más fáciles de usar y de entender. La indirección está oculta, y no hay necesidad de desreferenciar repetidamente la variable.

- P** ¿Por qué tener apuntadores si las referencias son más sencillas?
- R** Las referencias no pueden ser nulas, y no se pueden reasignar. Los apuntadores ofrecen mayor flexibilidad pero son un poco más difíciles de utilizar.
- P** ¿Para qué podría utilizar alguna vez el regreso por valor de una función?
- R** Si el objeto que va a regresar es local, debe regresarlo por valor o estará regresando una referencia a un objeto inexistente.
- P** Dado el peligro existente al regresar por referencia, ¿por qué no regresar siempre por valor?
- R** Se logra una mayor eficiencia al regresar por referencia. Se ahorra memoria y el programa se ejecuta más rápido.

9

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, "Respuestas a los cuestionarios y ejercicios", y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Cuál es la diferencia entre una referencia y un apuntador?
2. ¿Cuándo debe utilizar un apuntador en vez de una referencia?
3. ¿Qué regresa new si no hay memoria suficiente para crear el nuevo objeto?
4. ¿Qué es una referencia constante?
5. ¿Cuál es la diferencia entre pasar por referencia y pasar una referencia?

### Ejercicios

1. Escriba un programa que declare un int, una referencia a un int, y un apuntador a un int. Use el apuntador y la referencia para manipular el valor contenido en el int.
2. Escriba un programa que declare un apuntador constante a un entero constante. Inicialice el apuntador a una variable entera, varUno. Asigne el valor 6 a varUno. Use el apuntador para asignar el valor 7 a varUno. Cree otra variable de tipo entero que se llame varDos. Reasigne el apuntador a varDos. No compile todavía este ejercicio.
3. Ahora compile el programa del ejercicio 2. ¿Qué es lo que produce errores? ¿Qué es lo que produce advertencias?

4. Escriba un programa que origine un apuntador perdido.
5. Corrija el programa del ejercicio 4.
6. Escriba un programa que origine una fuga de memoria.
7. Corrija el programa del ejercicio 6.
8. **CAZA ERRORES:** ¿Qué está mal en este programa?

```
1:      #include <iostream.h>
2:
3:      class GATO
4:      {
5:          public:
6:              GATO(int edad) { suEdad = edad; }
7:              ~GATO(){}
8:              int ObtenerEdad() const { return suEdad; }
9:          private:
10:             int suEdad;
11:         };
12:
13:     GATO & CrearGato(int edad);
14:     int main()
15:     {
16:         int edad = 7;
17:         GATO Silvestre = CrearGato(edad);
18:         cout << "Silvestre tiene " << Silvestre.ObtenerEdad()
19:             << " años de edad\n";
20:         return 0;
21:     }
22:
23:     GATO & CrearGato(int edad)
24:     {
25:         GATO * apGato = new GATO(edad);
26:         return *apGato;
27:     }
```

9. Corrija el programa del ejercicio 8.

# SEMANA 2

## DÍA 10

### Funciones avanzadas

En el día 5, “Funciones”, conoció los fundamentos del trabajo con funciones. Ahora que sabe cómo funcionan los apuntadores y las referencias, puede hacer más cosas con las funciones. Hoy aprenderá lo siguiente:

- Cómo sobrecargar funciones miembro
- Cómo sobrecargar operadores
- Cómo escribir funciones para soportar clases con variables asignadas en forma dinámica

#### Sobrecarga de funciones miembro

En el día 5 aprendió cómo implementar el polimorfismo de funciones, conocido también como sobrecarga de funciones, escribiendo dos o más funciones con el mismo nombre pero con distintos parámetros. Las funciones miembro (o métodos) de una clase también se pueden sobrecargar, en forma muy parecida.

La clase `Rectangulo` que se muestra en el listado 10.1 tiene dos funciones `DibujarFigura()`. Una, que no lleva parámetros, dibuja el rectángulo con base en los valores actuales del objeto. La otra toma dos valores, `ancho` y `altura`, y dibuja el rectángulo con base en esos valores, ignorando los valores actuales del objeto.

**ENTRADA****LISTADO 10.1** Sobrecarga de funciones miembro

```
1: // Listado 10.1 Sobrecarga de funciones
2: // miembro de una clase
3:
4: #include <iostream.h>
5:
6: // Declaración de la clase Rectangulo
7: class Rectangulo
8: {
9: public:
10:    // constructores
11:    Rectangulo(int ancho, int altura);
12:    -Rectangulo(){}
13:    // función DibujarFigura() sobrecargada de la clase
14:    void DibujarFigura() const;
15:    void DibujarFigura(int unAncho, int unaAltura) const;
16: private:
17:    int suAncho;
18:    int suAltura;
19: };
20:
21: //Implementación del constructor
22: Rectangulo::Rectangulo(int ancho, int altura)
23: {
24:    suAncho = ancho;
25:    suAltura = altura;
26: }
27:
28: // Función DibujarFigura sobrecargada - no toma valores
29: // Dibuja con base en los valores miembro actuales de la clase
30: void Rectangulo::DibujarFigura() const
31: {
32:    DibujarFigura(suAncho, suAltura);
33: }
34:
35: // Función DibujarFigura sobrecargada - toma dos valores
36: // dibuja la figura con base en los parámetros
37: void Rectangulo::DibujarFigura(int ancho, int altura) const
38: {
39:    for (int i = 0; i < altura; i++)
40:    {
41:        for (int j = 0; j < ancho; j++)
42:        {
43:            cout << "*";
44:        }
45:        cout << "\n";
46:    }
47: }
```

```
49: // Programa controlador para mostrar las funciones sobrecargadas
50: int main()
51: {
52:     // inicializar un rectángulo con 30,5
53:     Rectangulo elRect(30, 5);
54:     cout << "DibujarFigura(): \n";
55:     elRect.DibujarFigura();
56:     cout << "\nDibujarFigura(40, 2): \n";
57:     elRect.DibujarFigura(40, 2);
58:     return 0;
59: }
```

SALIDA

10

ANÁLISIS

El listado 10.1 representa una versión simplificada del proyecto del repaso de la semana 1. La prueba de valores no válidos se ha omitido para ahorrar espacio.

al igual que algunas de las funciones de acceso. El programa principal ha sido reducido a un simple programa controlador, en lugar de un menú.

Sin embargo, la parte importante del código se encuentra en las líneas 14 y 15, en donde se sobrecarga la función miembro `DibujarFigura()`. La implementación de estos métodos sobrecargados de la clase se encuentra en las líneas 30 a 47. Observe que la versión de `DibujarFigura()` que no toma parámetros simplemente llama a la versión que toma dos parámetros, y le pasa los valores miembro actuales. Trate de no duplicar código en dos funciones. Si no puede, tratar de mantenerlas sincronizadas cuando haga cambios a una u otra será muy difícil, y estaré propenso a cometer errores.

El programa controlador de las líneas 50 a 59 crea un objeto rectángulo y luego llama a `DibujarFigura()`, primero sin pasar parámetros, y luego pasando dos enteros.

El compilador decide cuál método llamar con base en el número y tipo de parámetros proporcionados. Podríamos imaginarnos una tercera función sobrecargada llamada `DibujarFigura()` que tome una dimensión y una enumeración, ya sea para el ancho o la altura, esto a elección del usuario.

## Uso de valores predeterminados

Así como las funciones que no pertenecen a una clase pueden tener uno o más valores predeterminados, también las funciones miembro de una clase pueden tenerlos. Las mismas reglas se aplican para declarar los valores predeterminados, como se muestra en el listado 10.2.

### ENTRADA LISTADO 10.2 Uso de valores predeterminados

---

```
1: //Listado 10.2 Valores predeterminados en funciones miembro
2:
3: #include <iostream.h>
4:
5: // Declaración de la clase Rectangulo
6: class Rectangulo
7: {
8: public:
9:     // constructores
10:    Rectangulo(int ancho, int altura);
11:    ~Rectangulo(){}
12:    void DibujarFigura(int unAncho, int unaAltura,
13:                      bool UsarValsActuales = false) const;
14: private:
15:    int suAncho;
16:    int suAltura;
17: };
18:
19: //Implementación del constructor
20: Rectangulo::Rectangulo(int ancho, int altura):
21:     suAncho(ancho), // inicializaciones
22:     suAltura(altura)
23: {} // cuerpo vacío
24:
25:
26: // valores predeterminados usados para el tercer parámetro
27: void Rectangulo::DibujarFigura(int ancho, int altura,
28:                                 bool UsarValActual) const
29: {
30:     int imprimeAncho;
31:     int imprimeAltura;
32:
33:     if (UsarValActual == true)
34:     {
35:         // usar los valores actuales de la clase
36:         imprimeAncho = suAncho;
37:         imprimeAltura = suAltura;
38:     }
39:     else
40:     {
```

```

41:      // usar valores de los parámetros
42:      imprimeAncho = ancho;
43:      imprimeAltura = altura;
44:  }
45:  for (int i = 0; i < imprimeAltura; i++)
46:  {
47:      for (int j = 0; j < imprimeAncho; j++)
48:      {
49:          cout << "*";
50:      }
51:      cout << "\n";
52:  }
53: }
54:
55: // Programa controlador para mostrar las funciones sobrecargadas
56: int main()
57: {
58:     // inicializar un rectángulo con 30,5
59:     Rectangulo elRect(30, 5);
60:     cout << "DibujarFigura(0, 0, true)... \n";
61:     elRect.DibujarFigura(0, 0, true);
62:     cout << "DibujarFigura(40, 2)... \n";
63:     elRect.DibujarFigura(40, 2);
64:     return 0;
65: }

```

10

**SALIDA**

```

DibujarFigura(0, 0, true)...
*****
*****
*****
*****
*****
*****
DibujarFigura(40, 2)...
*****
*****
```

**ANÁLISIS**

El listado 10.2 reemplaza a la función sobrecargada `DibujarFigura()` con una función sencilla con parámetros predeterminados. La función se declara en la línea 12 para recibir tres parámetros. Los dos primeros, `unAncho` y `unaAltura`, son variables de tipo entero, y el tercero, `UsarValsActuales`, es un tipo `bool` que tiene `false` como valor predeterminado.

La implementación para esta función un poco rara empieza en la línea 27. El tercer parámetro, `UsarValActual`, es evaluado. Si es verdadero, se usarán las variables miembro `suAncho` y `suAltura` para asignar un valor a las variables locales `imprimeAncho` e `imprimeAltura`, respectivamente.

Si `UsarValActual` es falso, ya sea porque es su valor predeterminado o porque así lo asignó el usuario, se utilizan los dos primeros parámetros para asignar un valor a `imprimeAncho` e `imprimeAltura`.

Observe que si `UsarValActual` es verdadero, los valores de los otros dos parámetros se ignoran por completo.

Las líneas 21 y 22, que inicializan variables de datos miembro privados, se explican en la sección “Inicialización de objetos” que se verá más adelante en este día.

## Cómo elegir entre valores predeterminados y funciones sobrecargadas

Los listados 10.1 y 10.2 logran lo mismo, pero las funciones sobrecargadas del listado 10.1 son más fáciles de entender y se utilizan con más naturalidad. Además, si se necesita una tercera variación (tal vez el usuario quiera proporcionar el ancho o la altura, pero no ambas) es fácil extender las funciones sobrecargadas. Sin embargo, el valor predeterminado pronto se volverá inusualmente complejo a medida que se agreguen nuevas variaciones.

¿Cómo elegir entre usar la sobrecarga de funciones o los valores predeterminados? Una buena regla empírica es utilizar la sobrecarga de funciones bajo las siguientes condiciones:

- Que no exista un valor predeterminado razonable.
- Cuando necesite distintos algoritmos.
- Siempre que necesite parámetros de distintos tipos en las funciones o métodos (diferentes tipos de datos o clases para diferentes llamadas).

## Constructores predeterminados

Como se dijo en el día 6, “Clases base”, si no declara explícitamente un constructor para una clase, se crea un constructor predeterminado que no lleva parámetros y no hace nada. No obstante, usted puede hacer su propio constructor predeterminado que no lleve parámetros pero que “configure” su objeto como lo necesite.

El constructor que le ofrece el compilador se conoce como el constructor “predeterminado”, pero por convención así se llama también cualquier constructor que no lleve parámetros. Esto puede ser un poco confuso, pero por lo general es más claro viéndolo desde el contexto que cada uno quiera dar a entender.

Tome en cuenta que si codifica algún constructor, el compilador no creará el constructor predeterminado. Así es que si quiere un constructor que no lleve parámetros y ya tiene creado algún otro constructor, ¡deberá codificar su propio constructor predeterminado!

## Sobrecarga de constructores

El objetivo de un constructor es establecer el objeto; por ejemplo, el objetivo del constructor de un Rectángulo es crear un rectángulo. Antes de que el constructor se ejecute no existe ningún rectángulo, sólo un área de memoria. Al terminar el constructor, se tiene un objeto rectángulo completo y listo para utilizarlo.

Los constructores, al igual que todas las funciones miembro, se pueden sobrecargar. La capacidad para sobrecargar constructores es muy poderosa y flexible.

Por ejemplo, podría tener un objeto rectángulo que tenga dos constructores: El primero recibe una longitud y una anchura y crea un rectángulo de ese tamaño. El segundo no recibe valores y crea un rectángulo de un tamaño predeterminado. El listado 10.3 implementa esta idea.

10

### ENTRADA LISTADO 10.3 Sobre carga del constructor

```
1: // Listado 10.3
2: // Sobre carga de constructores
3:
4: #include <iostream.h>
5:
6: class Rectangulo
7: {
8: public:
9:     Rectangulo();
10:    Rectangulo(int ancho, int longitud);
11:    ~Rectangulo() {}
12:    int ObtenerAncho() const
13:        { return suAncho; }
14:    int ObtenerLongitud() const
15:        { return suLongitud; }
16: private:
17:     int suAncho;
18:     int suLongitud;
19: };
20:
21: Rectangulo::Rectangulo()
22: {
23:     suAncho = 5;
24:     suLongitud = 10;
25: }
26:
27: Rectangulo::Rectangulo (int ancho, int longitud)
28: {
29:     suAncho = ancho;
30:     suLongitud = longitud;
31: }
```

continua

**LISTADO 10.3** CONTINUACIÓN

```

33: int main()
34: {
35:     Rectangulo Rect1;
36:     cout << "Ancho de Rect1: ";
37:     cout << Rect1.ObtenerAncho() << endl;
38:     cout << "Longitud de Rect1: ";
39:     cout << Rect1.ObtenerLongitud() << endl;
40:
41:     int unAncho, unaLongitud;
42:     cout << "Escriba un ancho: ";
43:     cin >> unAncho;
44:     cout << "\nEscriba una longitud: ";
45:     cin >> unaLongitud;
46:
47:     Rectangulo Rect2(unAncho, unaLongitud);
48:     cout << "\nAncho de Rect2: ";
49:     cout << Rect2.ObtenerAncho() << endl;
50:     cout << "Longitud de Rect2: ";
51:     cout << Rect2.ObtenerLongitud() << endl;
52:     return 0;
53: }
```

**SALIDA**

Ancho de Rect1: 5  
 Longitud de Rect1: 10  
 Escriba un ancho: 20

Escriba una longitud: 50

Ancho de Rect2: 20  
 Longitud de Rect2: 50

**ANÁLISIS**

La clase `Rectangulo` se declara en las líneas 6 a 19. Se declaran dos constructores: el “constructor predeterminado” (en la línea 9) y el constructor que toma dos variables de tipo entero como parámetros (línea 10).

En la línea 35 se crea un rectángulo utilizando el constructor predeterminado, y se imprimen sus valores en las líneas 37 y 39. En las líneas 41 a 45 se pide al usuario un ancho y una longitud, y en la línea 47 se llama al constructor que lleva dos parámetros. Finalmente, el ancho y la longitud de este rectángulo se imprimen en las líneas 49 y 51.

Al igual que con cualquier función sobrecargada, el compilador elige el constructor correcto con base en el número y el tipo de los parámetros.

## Inicialización de objetos

Con la excepción de las líneas 21 y 22 del listado 10.2, hasta ahora las variables miembro de los objetos se establecían en el cuerpo del constructor. Sin embargo, los constructores se invocan en dos etapas: la etapa de inicialización y la del cuerpo.

Casi todas las variables se pueden establecer en cualquier etapa, ya sea en la etapa de inicialización o asignándoles un valor en el cuerpo del constructor. Es más limpio y eficiente inicializar variables miembro en la etapa de inicialización. El siguiente ejemplo muestra cómo inicializar variables miembro:

```
GATO():           // nombre del constructor y lista
    suEdad(5),    // de inicialización de parámetros
    suPeso(8)
{ }                // cuerpo del constructor
```

Después de los paréntesis de cierre de la lista de parámetros del constructor se escribe el símbolo de dos puntos (:). Luego se escribe el nombre de la variable miembro y un par de paréntesis. Dentro de los paréntesis se escribe la expresión que se va a utilizar para inicializar esa variable miembro. Si existe más de una inicialización, se debe separar cada una con una coma. El listado 10.4 muestra la definición de los constructores del listado 10.3, con la inicialización de las variables miembro en lugar de su asignación.

10

### Precaución

El archivo del listado 10.4 que viene en el CD-ROM no compilará pues sólo es una pieza de código (*segmento de código*), no un programa.

#### **ENTRADA** LISTADO 10.4 Un segmento de código que muestra la inicialización de las variables miembro

```
1: Rectangulo::Rectangulo():
2:     suAncho(5),
3:     suLongitud(10)
4: {}
5:
6: Rectangulo::Rectangulo(int ancho, int longitud):
7:     suAncho(ancho),
8:     suLongitud(longitud)
9: {}
10:
```

#### SALIDA

No hay salida.

Algunas variables se deben inicializar y no se les puede asignar un valor, como a las referencias y constantes. Es común tener otras asignaciones o instrucciones de acción en el cuerpo del constructor; sin embargo, es mejor utilizar la inicialización tanto como sea posible.

## Uso del constructor de copia

Además de proporcionar un constructor y un destructor predeterminados, el compilador proporciona un constructor de copia predeterminado. Este constructor se llama cada vez que se crea la copia de un objeto.

Cuando se pasa un objeto por valor, ya sea a una función o como el valor de retorno de una función, se crea una copia temporal de ese objeto. Si el objeto es definido por el usuario, se llama al constructor de copia de la clase. Ayer, en los listados 9.5 y 9.6, vio cómo se crean las copias temporales de las variables comunes. Hoy verá cómo funciona esto con los objetos.

Todos los constructores de copia toman como parámetro una referencia a un objeto de la misma clase. Es una buena idea hacerla una referencia constante porque el constructor no tendrá que alterar el objeto que se está pasando. Por ejemplo:

```
GATO(const GATO & elGato);
```

Aquí, el constructor `GATO` toma una referencia constante a un objeto `GATO` existente. El objetivo del constructor de copia es crear una copia de `elGato`.

El constructor de copia predeterminado simplemente copia cada variable miembro del objeto que se pasa como parámetro a las variables miembro del nuevo objeto. Esto se conoce como *copia de los datos miembro* (o superficial), y aunque esto está bien para la mayoría de las variables miembro, no es adecuado para las variables miembro que son apuntadores a objetos que se encuentran en el heap.

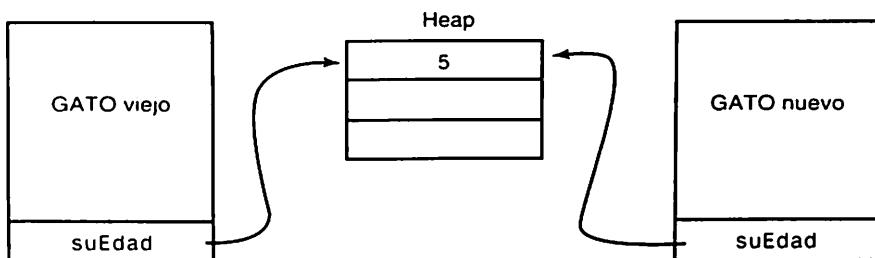
Una copia superficial o de los datos miembro copia en otro objeto los valores exactos de las variables miembro de un objeto. Los apuntadores de ambos objetos terminan apuntando a la misma dirección de memoria. Una copia profunda copia en memoria recién asignada los valores asignados en el heap.

Si la clase `GATO` incluye una variable miembro llamada `suEdad` que apunta a un entero en el heap, el constructor de copia predeterminado copiará en la variable miembro `suEdad` del nuevo `GATO`, la variable miembro `suEdad` del objeto `GATO` original. Ahora los dos objetos apuntarán a la misma dirección de memoria, como se muestra en la figura 10.1.

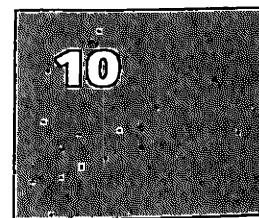
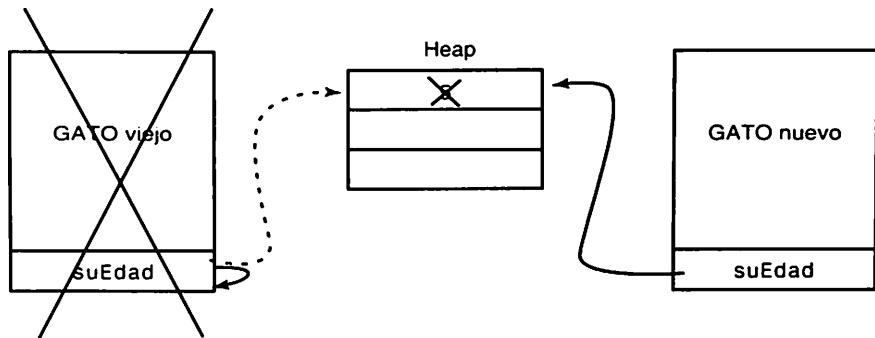
Esto conducirá a un desastre cuando cualquiera de los dos objetos `GATO` queden fuera de alcance. Como se mencionó en el día 8, “Apuntadores”, el trabajo del destructor es limpiar esta memoria. Si el destructor del `GATO` original libera esta memoria y el nuevo `GATO` aún sigue apuntando a esa memoria, se crea un apuntador perdido, y el programa estará en peligro mortal. La figura 10.2 ilustra este problema.

**FIGURA 10.1**

*Uso del constructor de copia predeterminado.*

**FIGURA 10.2**

*Creación de un apuntador perdido.*



La solución a este problema es crear su propio constructor de copia y asignar la memoria según sus necesidades. Después de asignar la memoria, los viejos valores se pueden copiar en la nueva memoria. Esto se conoce como copia profunda. El listado 10.5 muestra cómo hacer esto.

### ENTRADA LISTADO 10.5 Constructores de copia

```

1: // Listado 10.5
2: // Constructores de copia
3:
4: #include <iostream.h>
5:
6: class GATO
7: {
8: public:
9:     GATO(); // constructor predeterminado
10:    GATO (const GATO &); // constructor de copia
11:    ~GATO(); // destructor
12:    int ObtenerEdad() const
13:        { return *suEdad; }
14:    int ObtenerPeso() const
15:        { return *suPeso; }
16:    void AsignarEdad(int edad)
17:        { *suEdad = edad; }
18:
19: private:
20:     int * suEdad;
21:     int * suPeso;
22: };

```

*continua*

**LISTADO 10.5** CONTINUACIÓN

---

```
23:
24:     GATO::GATO()
25:     {
26:         suEdad = new int;
27:         suPeso = new int;
28:         *suEdad = 5;
29:         *suPeso = 9;
30:     }
31:
32:     GATO::GATO(const GATO & rhs)
33:     {
34:         suEdad = new int;
35:         suPeso = new int;
36:         *suEdad = rhs.ObtenerEdad(); // acceso público
37:         *suPeso = *(rhs.suPeso); // acceso privado
38:     }
39:
40:     GATO::~GATO()
41:     {
42:         delete suEdad;
43:         suEdad = NULL;
44:         delete suPeso;
45:         suPeso = NULL;
46:     }
47:
48:     int main()
49:     {
50:         GATO pelusa;
51:         cout << "edad de pelusa: ";
52:         cout << pelusa.ObtenerEdad() << endl;
53:         cout << "Establecer la edad de pelusa en 6...\n";
54:         pelusa.AsignarEdad(6);
55:         cout << "Crear a silvestre a partir de pelusa\n";
56:         GATO silvestre(pelusa);
57:         cout << "edad de pelusa: ";
58:         cout << pelusa.ObtenerEdad() << endl;
59:         cout << "edad de silvestre: ";
60:         cout << silvestre.ObtenerEdad() << endl;
61:         cout << "establecer edad de pelusa en 7...\n";
62:         pelusa.AsignarEdad(7);
63:         cout << "edad de pelusa: ";
64:         cout << pelusa.ObtenerEdad() << endl;
65:         cout << "edad de silvestre: ";
66:         cout << silvestre.ObtenerEdad() << endl;
67:         return 0;
68:     }
```

**SALIDA**

```
edad de pelusa: 5
Establecer la edad de pelusa en 6...
Crear a silvestre a partir de pelusa
edad de pelusa: 6
edad de silvestre: 6
establecer edad de pelusa en 7...
edad de pelusa: 7
edad de silvestre: 6
```

**ANÁLISIS**

En las líneas 6 a 22 se declara la clase GATO. Observe que en la línea 9 se declara un constructor predeterminado, y en la línea 10 se declara un constructor de copia.

En las líneas 20 y 21 se declaran dos variables miembro, cada una como apuntador a un entero. Por lo general, habría muy pocos motivos para que una clase guardara variables miembro de tipo `int` como apuntadores, pero se hizo esto para mostrar cómo se manejan las variables miembro en el heap.

El constructor predeterminado que está en las líneas 24 a 30 asigna espacio en el heap para dos variables de tipo `int` y luego les asigna valores.

El constructor de copia empieza en la línea 32. Observe que el parámetro es `rhs`. Es común referirse al parámetro para un constructor de copia como `rhs`, que significa “lado derecho” (right-hand side). Al analizar las asignaciones de las líneas 36 y 37 se ve que el objeto que se pasa como parámetro se encuentra al lado derecho del signo de igual. Esto funciona así:

En las líneas 34 y 35 se asigna memoria en el heap. Luego, en las líneas 36 y 37 los valores del GATO existente se asignan al valor que se encuentra en la nueva ubicación de memoria.

El parámetro `rhs` es un GATO que se pasa al constructor de copia como referencia constante. Como un objeto de la clase GATO, `rhs` tiene todas las variables miembro de cualquier otro objeto GATO.

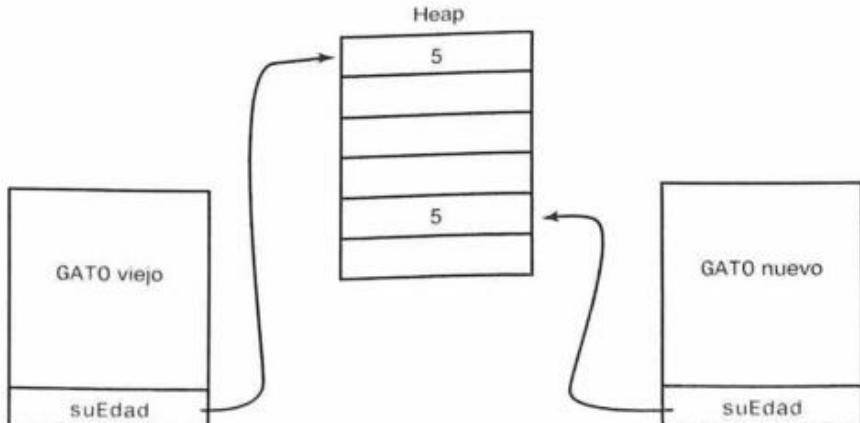
Cualquier objeto GATO puede tener acceso a las variables miembro privadas de cualquier otro objeto GATO; sin embargo, es una buena práctica de programación utilizar métodos de acceso públicos siempre que sea posible. La función miembro `rhs.ObtenerEdad()` regresa el valor guardado en la memoria a la que apunta la variable miembro `suEdad` de `rhs`.

La figura 10.3 muestra un diagrama de lo que está ocurriendo aquí. Los valores a los que apuntan las variables miembro del GATO existente se copian en la memoria asignada para el nuevo GATO.

10

**FIGURA 10.3**

*Ilustración de una copia profunda.*



En la línea 50 se crea un GATO llamado `pelusa`. Se imprime la edad de `pelusa` y luego, en la línea 54, se establece su edad en 6. En la línea 56 se crea un nuevo GATO llamado `silvestre`, usando el constructor de copia y pasando a `pelusa` como parámetro. Si `pelusa` se hubiera pasado como parámetro a una función, el compilador hubiera hecho la misma llamada al constructor de copia.

En las líneas 58 y 60 se imprimen las edades de ambos objetos GATO. Evidentemente, `silvestre` tiene la edad de `pelusa`, es decir 6, no la edad predeterminada de 5. En la línea 62 se establece la edad de `pelusa` en 7, y luego se imprimen otra vez las edades. Esta vez la edad de `pelusa` es 7, pero la edad de `silvestre` aún es 6, lo que demuestra que se guardan en áreas separadas de memoria.

Cuando los objetos GATO quedan fuera de alcance, sus destructores se invocan automáticamente. La implementación del destructor GATO se muestra en las líneas 40 a 46. En ambos apuntadores, `suEdad` y `suPeso`, se llama a `delete`, lo que regresa al heap la memoria asignada. Además, por seguridad, los apuntadores se reasignan a `NULL`.

## Sobrecarga de operadores

C++ tiene varios tipos de datos integrados, incluyendo `int`, `double`, `char`, etc. Cada uno de estos tipos tiene una variedad de operadores integrados, como el de suma (+) y el de multiplicación (\*). C++ también le permite agregar estos operadores a sus propias clases.

Para explorar completamente la sobrecarga de operadores, el listado 10.6 crea una nueva clase llamada `Contador`. Se utilizará un objeto `Contador` para contar (*;sorpresa!*) en los ciclos y en otras aplicaciones en donde se deba incrementar, decrementar o rastrear de alguna otra forma un número.

---

**ENTRADA LISTADO 10.6 La clase Contador**

---

```
1: // Listado 10.6
2: // La clase Contador
3:
4: #include <iostream.h>
5:
6: class Contador
7: {
8: public:
9:     Contador();
10:    ~Contador(){}
11:    int ObtenerSuVal()const
12:        { return suVal; }
13:    void AsignarSuVal(int x)
14:        { suVal = x; }
15: private:
16:     int suVal;
17: };
18:
```

```

19: Contador::Contador():
20:     suVal(0)
21: {}
22:
23: int main()
24: {
25:     Contador i;
26:     cout << "El valor de i es ";
27:     cout << i.ObtenerSuVal() << endl;
28:     return 0;
29: }

```

**SALIDA** El valor de i es 0

**ANÁLISIS** Así como se ve, es una clase bastante inútil. Se define en las líneas 6 a 17. Su única variable miembro es un `int`. El constructor predeterminado, el cual se declara en la línea 9 y cuya implementación se encuentra en la línea 19, inicializa esa única variable miembro, `suVal`, con cero.

10

A diferencia de un `int` común definido por el compilador, el objeto `Contador` no se puede incrementar, decrementar, sumar, asignarle un valor ni se puede manipular de ninguna otra forma. Ésta es una característica, no un problema. A cambio de esto, ¡hace que su valor sea más difícil de imprimir! La impresión es más difícil debido a que no existe un formato definido por el compilador, por lo que usted tiene que crearlo.

## Cómo escribir una función de incremento

La sobrecarga de operadores restaura la mayor parte de la funcionalidad que se ha quitado de esta clase. Por ejemplo, existen dos formas de agregar la capacidad de incrementar un objeto `Contador`. La primera es escribir un método de incremento, como se muestra en el listado 10.7.

**ENTRADA** **LISTADO 10.7** Cómo agregar un operador de incremento

```

1: // Listado 10.7
2: // La clase Contador
3:
4:
5: #include <iostream.h>
6:
7: class Contador
8: {
9: public:
10:    Contador();
11:    ~Contador(){}
12:    int ObtenerSuVal()const
13:        { return suVal; }
14:    void AsignarSuVal(int x)

```

continua

**LISTADO 10.7** CONTINUACIÓN

```

15:     { suVal = x; }
16:     void Incremento()
17:     { ++suVal; }
18: private:
19:     int suVal;
20: };
21:
22: Contador::Contador():
23:     suVal(0)
24: {}
25:
26: int main()
27: {
28:     Contador i;
29:     cout << "El valor de i es ";
30:     cout << i.ObtenerSuVal() << endl;
31:     i.Incremento();
32:     cout << "El valor de i es ";
33:     cout << i.ObtenerSuVal() << endl;
34:     return 0;
35: }
```

**SALIDA**

El valor de i es 0  
El valor de i es 1

**ANÁLISIS**

El listado 10.7 agrega la función Incremento, definida en las líneas 16 y 17. Aunque esto funciona, es incómodo de usar. El programa pide a gritos la capacidad de agregar el operador ++ y, por supuesto, se puede hacer.

## Sobrecarga del operador de prefijo

Los operadores de prefijo se pueden sobrecargar al declarar funciones con la siguiente forma:

`tipoDeRetorno operator op (parámetros)`

Aquí, op es el operador a sobrecargar. Por lo tanto, el operador ++ se puede sobrecargar con la siguiente sintaxis:

`void operator++ ()`

El listado 10.8 muestra esta alternativa.

**ENTRADA****LISTADO 10.8** Sobre carga del operador ++

```

1: // Listado 10.8
2: // La clase Contador
3:
4: #include <iostream.h>
5:
```

```

6: class Contador
7: {
8: public:
9:     Contador();
10:    -Contador(){}
11:    int ObtenerSuVal()const
12:    { return suVal; }
13:    void AsignarSuVal(int x)
14:    {suVal = x; }
15:    void Incremento()
16:    { ++suVal; }
17:    void operator++ ()
18:    { ++suVal; }
19: private:
20:     int suVal;
21: };
22:
23: Contador::Contador():
24:     suVal(0)
25: {}
26:
27: int main()
28: {
29:     Contador i;
30:     cout << "El valor de i es ";
31:     cout << i.ObtenerSuVal() << endl;
32:     i.Incremento();
33:     cout << "El valor de i es ";
34:     cout << i.ObtenerSuVal() << endl;
35:     ++i;
36:     cout << "El valor de i es ";
37:     cout << i.ObtenerSuVal() << endl;
38:     return 0;
39: }

```

**SALIDA**

El valor de i es 0  
 El valor de i es 1  
 El valor de i es 2

**ANÁLISIS**

En la línea 17 se sobrecarga el operador `++` y se utiliza en la línea 35. Esto es lo más cercano a la sintaxis que se esperaría con el objeto `Contador`. En este punto podría considerar colocar las capacidades adicionales por las que se creó `Contador` en primer lugar, como detectar el momento en el que `Contador` sobrepase su tamaño máximo.

Sin embargo, hay un defecto considerable en la manera en que se escribió el operador de incremento. Si quiere colocar el `Contador` en el lado derecho de una asignación, esto fallará. Por ejemplo:

`Contador a = ++i;`

Este código trata de crear un nuevo Contador llamado a, y luego asignarle el valor de i después de incrementar i. El constructor de copia integrado se encargará de la asignación, pero el operador de incremento actual no regresa un objeto Contador. Regresa void. Usted no puede asignar un objeto void a un objeto Contador. ¡No puede crear algo de la nada!

## Cómo regresar tipos en funciones con operadores sobrecargados

Evidentemente, lo que usted necesita es regresar un objeto Contador para que pueda asignarlo a otro objeto Contador. ¿Cuál objeto debe regresar? Un método sería crear un objeto temporal y regresar ese objeto. El listado 10.9 ejemplifica este método.

**ENTRADA****LISTADO 10.9** Cómo regresar un objeto temporal

```
1: // Listado 10.9
2: // operator++ regresa un objeto temporal
3:
4: #include <iostream.h>
5:
6: class Contador
7: {
8: public:
9:     Contador();
10:    ~Contador(){}
11:    int ObtenerSuVal()const
12:    { return suVal; }
13:    void AsignarSuVal(int x)
14:    { suVal = x; }
15:    void Incremento()
16:    { ++suVal; }
17:    Contador operator++ ();
18: private:
19:     int suVal;
20: };
21:
22: Contador::Contador():
23:     suVal(0)
24: {}
25:
26: Contador Contador::operator++()
27: {
28:     ++suVal;
29:     Contador temp;
30:
31:     temp.AsignarSuVal(suVal);
32:     return temp;
33: }
34:
35: int main()
36: {
```

```

37:     Contador i;
38:     cout << "El valor de i es ";
39:     cout << i.ObtenerSuVal() << endl;
40:     i.Incremento();
41:     cout << "El valor de i es ";
42:     cout << i.ObtenerSuVal() << endl;
43:     ++i;
44:     cout << "El valor de i es ";
45:     cout << i.ObtenerSuVal() << endl;
46:     Contador a = ++i;
47:     cout << "El valor de a: " << a.ObtenerSuVal();
48:     cout << " y de i: " << i.ObtenerSuVal() << endl;
49:     return 0;
50: }

```

**SALIDA**

```

El valor de i es 0
El valor de i es 1
El valor de i es 2
El valor de a: 3 y de i: 3

```

**10****ANÁLISIS**

En esta versión se ha declarado el operador `++` en la línea 17 para que regrese un objeto de tipo `Contador`. En la línea 29 se crea una variable temporal llamada `temp`, y su valor se asigna de forma que concuerde con el del objeto actual. Esa variable temporal se regresa y se asigna inmediatamente a `a` en la línea 46.

## Cómo regresar objetos temporales sin nombre

En realidad, no hay necesidad de nombrar el objeto temporal creado en la línea 29. Si `Contador` tuviera un constructor que tomara un valor, usted podría simplemente regresar el resultado de ese constructor como valor de retorno del operador de incremento. El listado 10.10 ejemplifica esta idea.

**ENTRADA****LISTADO 10.10** Cómo regresar un objeto temporal sin nombre

```

1: // Listado 10.10
2: // operator++ regresa un objeto temporal sin nombre
3:
4: #include <iostream.h>
5:
6: class Contador
7: {
8: public:
9:     Contador();
10:    Contador(int val);
11:    ~Contador(){}
12:    int ObtenerSuVal()const
13:    { return suVal; }
14:    void AsignarSuVal(int x)
15:    { suVal = x; }
16:    void Incremento()
17:    { ++suVal; }

```

*continua*

**LISTADO 10.10** CONTINUACIÓN

```

18:     Contador operator++ ();
19: private:
20:     int suVal;
21: };
22:
23: Contador::Contador():
24:     suVal(0)
25: {}
26:
27: Contador::Contador(int val):
28:     suVal(val)
29: {}
30:
31: Contador Contador::operator++()
32: {
33:     ++suVal;
34:     return Contador (suVal);
35: }
36:
37: int main()
38: {
39:     Contador i;
40:     cout << "El valor de i es ";
41:     cout << i.ObtenerSuVal() << endl;
42:     i.Incremento();
43:     cout << "El valor de i es ";
44:     cout << i.ObtenerSuVal() << endl;
45:     ++i;
46:     cout << "El valor de i es ";
47:     cout << i.ObtenerSuVal() << endl;
48:     Contador a = ++i;
49:     cout << "El valor de a: " << a.ObtenerSuVal();
50:     cout << " y de i: " << i.ObtenerSuVal() << endl;
51:     return 0;
52: }
```

**SALIDA**

El valor de i es 0  
 El valor de i es 1  
 El valor de i es 2  
 El valor de a: 3 y de i: 3

**ANÁLISIS**

En la línea 10 se declara un nuevo constructor, el cual toma un `int` como parámetro. La implementación se encuentra en las líneas 27 a 29, y en ésta se inicializa a `suVal` con el valor que recibe.

Ahora la implementación de `operator++` está simplificada. En la línea 33, `suVal` se incrementa. Luego, en la línea 34 se crea un objeto `Contador` temporal, se inicializa con el valor contenido en `suVal`, y luego se regresa como resultado de `operator++`.

Esto es más elegante, pero surge una pregunta: “¿Por qué crear un objeto temporal?” Recuerde que cada objeto temporal debe ser construido y posteriormente destruido (cada uno es una operación costosa en potencia). Además, el objeto *i* ya existe y ya tiene el valor correcto, así que, ¿por qué no regresarlo? Podemos resolver este problema utilizando el apuntador *this*.

## Uso del apuntador *this*

El apuntador *this*, como se dijo en el día 9, “Referencias”, se puede pasar a la función miembro *operator++* al igual que a todas las demás funciones miembro. El apuntador *this* apunta a *i*, y si es desreferenciado, regresará el objeto *i*, el cual ya tiene el valor correcto en su variable miembro *suVal*. El listado 10.11 ilustra el regreso del apuntador *this* desreferenciado, evitando así la creación de un objeto temporal innecesario.

10

**ENTRADA LISTADO 10.11** Regreso del apuntador *this*

```
1: // Listado 10.11
2: // Cómo regresar el apuntador this desreferenciado
3:
4: #include <iostream.h>
5:
6: class Contador
7: {
8: public:
9:     Contador();
10:    ~Contador(){}
11:    int ObtenerSuVal()const
12:        { return suVal; }
13:    void AsignarSuVal(int x)
14:        { suVal = x; }
15:    void Incremento()
16:        { ++suVal; }
17:    const Contador & operator++ ();
18: private:
19:     int suVal;
20: };
21:
22: Contador::Contador():
23:     suVal(0)
24: {};
25:
26: const Contador & Contador::operator++()
27: {
28:     ++suVal;
29:     return *this;
30: }
```

**LISTADO 10.11** CONTINUACIÓN

```

32: int main()
33: {
34:     Contador i;
35:     cout << "El valor de i es ";
36:     cout << i.ObtenerSuVal() << endl;
37:     i.Incremento();
38:     cout << "El valor de i es ";
39:     cout << i.ObtenerSuVal() << endl;
40:     ++i;
41:     cout << "El valor de i es ";
42:     cout << i.ObtenerSuVal() << endl;
43:     Contador a = ++i;
44:     cout << "El valor de a: " << a.ObtenerSuVal();
45:     cout << " y de i: " << i.ObtenerSuVal() << endl;
46:     return 0;
47: }
```

**SALIDA**

El valor de i es 0  
 El valor de i es 1  
 El valor de i es 2  
 El valor de a: 3 y de i: 3

**ANÁLISIS**

La implementación de `operator++`, líneas 26 a 30, ha sido cambiada para desreferenciar el apuntador `this` y regresar el objeto actual. Esto hace que se asigne un objeto `Contador` al objeto `a`. Como se dijo anteriormente, si el objeto `Contador` asignara memoria, sería importante evadir al constructor de copia. En este caso, el constructor de copia predeterminado funciona perfectamente.

Observe que el valor regresado es una referencia a `Contador`, lo que evita la creación de un objeto temporal adicional. Es una referencia `const` porque el valor no debe ser cambiado por la función que usa este `Contador` (es decir, no se debe asignar esta referencia a otro objeto `Contador`).

## Sobrecarga del operador de posfijo

Hasta ahora, ha sobrecargado el operador de prefijo. ¿Qué pasa si quiere sobrecargar el operador de incremento de posfijo? El compilador tiene un problema aquí: ¿Cómo distinguir entre prefijo y posfijo? Por convención, para la declaración del operador se proporciona una variable de tipo entero como parámetro. El valor del parámetro se ignora; sólo es una indicación de que es el operador de posfijo. Esto se muestra en el listado 10.12.

## Cuáles son las diferencias entre prefijo y posfijo

Antes de poder escribir el operador de posfijo, debe entender qué diferencia existe entre éste y el operador de prefijo. En el día 4, “Expresiones e instrucciones”, analizamos esto con detalle (vea el listado 4.3).

Para recordar, prefijo significa “incrementar y luego utilizar”, pero posfijo dice “utilizar y luego incrementar”.

Por lo tanto, aunque el operador de prefijo puede simplemente incrementar el valor y luego regresar el objeto mismo, el de posfijo debe regresar el valor que existía antes de ser incrementado. Para hacer esto, debemos crear un objeto temporal que guarde el valor original, incrementar el valor del objeto original, y luego regresar el objeto temporal.

Repasemos eso otra vez. Considere la siguiente línea de código:

```
a = x++;
```

Si x fuera 5, después de esta instrucción a es 5, pero x es 6. Por lo tanto, regresamos el valor de x y lo asignamos a a, y luego incrementamos el valor de x. Si x es un objeto, su operador de incremento de postfijo debe guardar el valor original (5) en un objeto temporal, incrementar el valor de x a 6, y luego regresar ese valor temporal para asignar su valor a a.

Observe que debido a que estamos regresando el valor temporal, debemos regresarlo por valor y no por referencia, ya que el valor temporal quedará fuera de alcance tan pronto como termine la función.

El listado 10.12 muestra el uso de ambos operadores, prefijo y posfijo.

#### ENTRADA LISTADO 10.12 Operadores de prefijo y posfijo

```
1: // Listado 10.12
2: // Operadores de prefijo y posfijo
3:
4: #include <iostream.h>
5:
6: class Contador
7: {
8: public:
9:     Contador();
10:    ~Contador(){}
11:    int ObtenerSuVal()const
12:    { return suVal; }
13:    void AsignarSuVal(int x)
14:    { suVal = x; }
15:    const Contador & operator++ (); // prefijo
16:    const Contador operator++ (int); // posfijo
17: private:
18:     int suVal;
19: };
20:
21: Contador::Contador():
22:     suVal(0)
23: {}
24:
```

continúa

**LISTADO 10.12** CONTINUACIÓN

```
25: const Contador & Contador::operator++()
26: {
27:     ++suVal;
28:     return *this;
29: }
30:
31: const Contador Contador::operator++(int x)
32: {
33:     Contador temp(*this);
34:     ++suVal;
35:     return temp;
36: }
37:
38: int main()
39: {
40:     Contador i;
41:     cout << "El valor de i es ";
42:     cout << i.ObtenerSuVal() << endl;
43:     i++;
44:     cout << "El valor de i es ";
45:     cout << i.ObtenerSuVal() << endl;
46:     ++i;
47:     cout << "El valor de i es ";
48:     cout << i.ObtenerSuVal() << endl;
49:     Contador a = ++i;
50:     cout << "El valor de a: " << a.ObtenerSuVal();
51:     cout << " y de i: " << i.ObtenerSuVal() << endl;
52:     a = i++;
53:     cout << "El valor de a: " << a.ObtenerSuVal();
54:     cout << " y de i: " << i.ObtenerSuVal() << endl;
55:     return 0;
56: }
```

**SALIDA**

```
El valor de i es 0
El valor de i es 1
El valor de i es 2
El valor de a: 3 y de i: 3
El valor de a: 3 y de i: 4
```

**ANÁLISIS**

El operador de posfijo se declara en la línea 16 y se implementa en las líneas 31 a 36. Observe que el prototipo del operador de prefijo (línea 15) no incluye el indicador de entero (x), sino que se utiliza con su sintaxis normal. El operador de posfijo sí utiliza ese indicador (x) para establecer que es el posfijo y no el prefijo. Sin embargo, el valor del indicador (x) nunca se utiliza.

### Sobrecarga de operadores: operadores unarios

Un operador sobrecargado se declara de la misma forma que una función. Utilice la palabra reservada `operator`, seguida del operador que se va a sobrecargar. Las funciones de operadores unarios no llevan parámetros, con la excepción del incremento y decremento de posfijo, que toman un entero como indicador.

#### Ejemplo 1

```
const Contador& Contador::operator++ () ;
```

#### Ejemplo 2

```
Contador Contador::operator--(int) ;
```

10

## Uso del operador de suma

El operador de incremento es un operador unario. Funciona sólo con un objeto. El operador de suma (+) es un operador binario, en el que se involucran dos objetos. ¿Cómo se implementa la sobrecarga del operador + para `Contador`?

El objetivo es poder declarar dos variables `Contador` y luego sumarlas, como en el siguiente ejemplo:

```
Contador varUno, varDos, varTres;
VarTres = VarUno + VarDos;
```

Una vez más, podría empezar escribiendo una función llamada `Sumar()`, que tomaría un `Contador` como su argumento, sumaría los valores y luego regresaría un `Contador` con el resultado. El listado 10.13 ejemplifica este método.

### ENTRADA LISTADO 10.13 La función Sumar()

```
1: // Listado 10.13
2: // Función Sumar
3:
4: #include <iostream.h>
5:
6: class Contador
7: {
8: public:
9:     Contador();
10:    Contador(int valorInicial);
11:    -Contador(){}
12:    int ObtenerSuVal()const
13:        { return suVal; }
14:    void AsignarSuVal(int x)
15:        { suVal = x; }
16:    Contador Sumar(const Contador &);
```

continua

**LISTADO 10.13** CONTINUACIÓN

---

```
17: private:
18:     int suVal;
19: };
20:
21: Contador::Contador(int valorInicial):
22:     suVal(valorInicial)
23: {}
24:
25: Contador::Contador():
26:     suVal(0)
27: {}
28:
29: Contador Contador::Sumar(const Contador & rhs)
30: {
31:     return Contador(suVal + rhs.ObtenerSuVal());
32: }
33:
34: int main()
35: {
36:     Contador varUno(2), varDos(4), varTres;
37:
38:     varTres = varUno.Sumar(varDos);
39:     cout << "varUno: " << varUno.ObtenerSuVal() << endl;
40:     cout << "varDos: " << varDos.ObtenerSuVal() << endl;
41:     cout << "varTres: " << varTres.ObtenerSuVal() << endl;
42:     return 0;
43: }
```

---

**SALIDA**

```
varUno: 2
varDos: 4
varTres: 6
```

**ANÁLISIS**

La función `Sumar()` se declara en la línea 16. Toma una referencia constante a `Contador`, que es el número a sumar al objeto actual. Regresa un objeto `Contador`, que es el resultado que se va a asignar en el lado izquierdo de la instrucción de asignación, como se muestra en la línea 38. Es decir, `varUno` es el objeto, `varDos` es el parámetro para la función `Sumar()`, y el resultado se asigna a `varTres`.

Para poder crear `varTres` sin tener que inicializarla con un valor, se requiere un constructor predeterminado. El constructor predeterminado inicializa `suVal` con `0`, como se muestra en las líneas 25 a 27. Como `varUno` y `varDos` necesitan inicializarse con un valor distinto de cero, se creó otro constructor, como se muestra en las líneas 21 a 23. Otra solución para este problema es proporcionar el valor predeterminado `0` al constructor declarado en la línea 10.

## Cómo sobrecargar a operator+

La función `Sumar()` se muestra en las líneas 29 a 32. Funciona, pero su uso es muy sofisticado. Sobrecargar el operador `+` ofrecería un uso más natural de la clase `Contador`. El listado 10.14 ejemplifica esto.

### ENTRADA LISTADO 10.14 operator+

```
1: // Listado 10.14
2: //Sobrecargar el operador de suma (+)
3:
4: #include <iostream.h>
5:
6: class Contador
7: {
8: public:
9:     Contador();
10:    Contador(int valorInicial);
11:    ~Contador(){}
12:    int ObtenerSuVal()const
13:        { return suVal; }
14:    void AsignarSuVal(int x)
15:        { suVal = x; }
16:    Contador operator+ (const Contador &);
17: private:
18:     int suVal;
19: };
20:
21: Contador::Contador(int valorInicial):
22:     suVal(valorInicial)
23: {}
24:
25: Contador::Contador():
26:     suVal(0)
27: {}
28:
29: Contador Contador::operator+ (const Contador & rhs)
30: {
31:     return Contador(suVal + rhs.ObtenerSuVal());
32: }
33:
34: int main()
35: {
36:     Contador varUno(2), varDos(4), varTres;
37:
38:     varTres = varUno + varDos;
39:     cout << "varUno: " << varUno.ObtenerSuVal()<< endl;
40:     cout << "varDos: " << varDos.ObtenerSuVal() << endl;
41:     cout << "varTres: " << varTres.ObtenerSuVal() << endl;
42:     return 0;
43: }
```

**SALIDA**

```
varUno: 2
varDos: 4
varTres: 6
```

**ANÁLISIS**

En la línea 16 se declara a `operator+`, y se define en las líneas 29 a 32. Compare estas líneas con la declaración y la definición de la función `Sumar()` del listado anterior; son casi idénticas. Sin embargo, la sintaxis de su uso es bastante diferente. Es más natural decir esto:

```
varTres = varUno + varDos;
```

que decir:

```
varTres = varUno.Sumar(varDos);
```

No es un gran cambio, pero es suficiente para que el programa sea más fácil de usar y de entender.

**Nota**

Las técnicas utilizadas para sobrecargar el operador `++` se pueden aplicar a los demás operadores unarios, como el operador `--`.

**Sobrecarga de operadores: operadores binarios**

Los operadores binarios se crean igual que los operadores unarios, sólo que los binarios si llevan un parámetro. El parámetro es una referencia constante a un objeto del mismo tipo.

**Ejemplo 1**

```
Contador Contador::operator+ ( const Contador & rhs );
```

**Ejemplo 2**

```
Contador Contador::operator- ( const Contador & rhs );
```

**Cuestiones adicionales relacionadas con la sobrecarga de operadores**

Los operadores sobrecargados pueden ser funciones miembro, como se describe en esta lección, o funciones no miembro. Estas últimas se describirán en el día 14, "Clases y funciones especiales", cuando hablemos sobre las funciones amigas.

Los únicos operadores que deben ser miembros de la clase son los operadores de asignación (`=`), de subíndice de arreglos (`[ ]`), de llamada a función (`()`) y de flecha (`->`).

El operador `[ ]` se verá en el día 11, "Herencia", al hablar sobre los arreglos.

## Limitaciones de la sobrecarga de operadores

Los operadores de tipos de datos integrados (como `int`) no se pueden sobrecargar. El orden de precedencia no se puede cambiar, y la aridad del operador, es decir, si es unario o binario, tampoco se puede cambiar. No puede crear nuevos operadores, por lo que no puede declarar `**` como operador "de potencia".

La aridad se refiere a la cantidad de términos que se utilizan en el operador. Algunos operadores de C++ son unarios y sólo utilizan un término (`miValor++`). Otros son binarios y utilizan dos términos (`a + b`). Sólo un operador es ternario y utiliza tres términos. El operador `? :` se denomina comúnmente operador ternario porque es el único operador ternario en C++ (`a > b ? x : y`).

## Qué se debe sobrecargar

La sobrecarga de operadores es uno de los aspectos de C++ de los que más abusan los programadores novatos. Es tentador crear nuevos e interesantes usos para algunos de los operadores más confusos, pero esto produce invariablemente un código confuso y difícil de leer.

Claro que hacer que el operador `+` reste y el operador `*` sume podría ser divertido, pero ningún programador profesional haría eso. El mayor peligro recae en el bien intencionado pero idiosincrásico uso de un operador (usar `+` para indicar que se deben concatenar una serie de letras, o usar `/` para indicar que se debe dividir una cadena de caracteres). Existe un buen motivo para considerar estos usos, pero hay un mejor motivo para proceder con cautela. Recuerde, el objetivo de la sobrecarga de los operadores es aumentar su uso y su comprensión.

10

| DEBE                                                                                                | No DEBE                                           |
|-----------------------------------------------------------------------------------------------------|---------------------------------------------------|
| <b>DEBE</b> utilizar la sobrecarga de operadores cuando esto ayude a que el programa sea más claro. | <b>NO DEBE</b> crear operadores contraintuitivos. |
| <b>DEBE</b> regresar un objeto de la misma clase para la cual sobrecarga los operadores .           |                                                   |

## Uso del operador de asignación

La cuarta y última función que proporciona el compilador, si usted no especifica una, es el operador de asignación (`operator=()`). Este operador se llama siempre que se asigna algo a un objeto. Por ejemplo:

```
GATO gatoUno(5,7);
GATO gatoDos(3,4);
// ... aquí puede ir más código
gatoDos = gatoUno;
```

Aquí se crea `gatoUno` y se inicializa con `suEdad` igual a 5 y `suPeso` igual a 7. Después se crea `gatoDos` y se le asignan los valores 3 y 4.

Después, los valores contenidos en `gatoUno` se asignan a `gatoDos`. Aquí surgen dos cuestiones: ¿Qué pasa si `suEdad` es un apuntador, y qué pasa con los valores originales contenidos en `gatoDos`?

Cuando examinamos el constructor de copia, vio el manejo de variables miembro que guardan sus valores en el heap. Las mismas cuestiones surgen aquí, como vio en las figuras 10.1 y 10.2.

Los programadores de C++ distinguen entre una copia superficial, o de datos miembro, por una parte, y una copia profunda por la otra. Una copia superficial sólo copia los miembros, y ambos objetos terminan apuntando a la misma área de memoria. Una copia profunda asigna la memoria necesaria. Esto se ilustra en la figura 10.3.

Sin embargo, hay un detalle adicional relacionado con el operador de asignación. El objeto `gatoDos` ya existe y tiene memoria asignada. Esa memoria se debe eliminar si no se desea una fuga de memoria. Pero, ¿qué ocurre si se asigna `gatoDos` a sí mismo?

```
gatoDos = gatoDos;
```

Es muy poco probable que alguien quiera hacer esto intencionalmente, pero el programa debe ser capaz de manejarlo. Lo que es más importante, es posible que esto ocurra accidentalmente si las referencias y los apuntadores desreferenciados ocultan el hecho de que la asignación es al objeto mismo.

Si no manejara cuidadosamente este problema, `gatoDos` eliminaría su asignación de memoria. Luego, cuando estuviera listo para copiar en la memoria del lado derecho de la asignación, tendría un problema muy grande. La memoria ya no estaría ahí.

Para protegerse contra esto, su operador de asignación debe verificar si lo que está de su lado derecho es el objeto mismo. El operador hace esto examinando el apuntador `this`. El listado 10.15 muestra una clase con un operador de asignación.

---

**ENTRADA LISTADO 10.15** Un operador de asignación

```
1: // Listado 10.15
2: // Constructores de copia
3:
4: #include <iostream.h>
5:
6: class GATO
7: {
8: public:
9:     GATO(); // constructor predeterminado
10:    // iconstructor y destructor de copia suprimidos!
11:    int ObtenerEdad() const
```

```
12:         { return *suEdad; }
13:     int ObtenerPeso() const
14:         { return *suPeso; }
15:     void AsignarEdad(int edad)
16:         { *suEdad = edad; }
17:     GATO & operator=(const GATO &);
18: private:
19:     int * suEdad;
20:     int * suPeso;
21: };
22:
23: GATO::GATO()
24: {
25:     suEdad = new int;
26:     suPeso = new int;
27:     *suEdad = 5;
28:     *suPeso = 9;
29: }
30:
31: GATO & GATO::operator=(const GATO & rhs)
32: {
33:     if (this == &rhs)
34:         return *this;
35:     *suEdad = rhs.ObtenerEdad();
36:     *suPeso = rhs.ObtenerPeso();
37:     return *this;
38: }
39:
40:
41: int main()
42: {
43:     GATO pelusa;
44:
45:     cout << "edad de pelusa: ";
46:     cout << pelusa.ObtenerEdad() << endl;
47:     cout << "Estableciendo edad de pelusa en 6...\n";
48:     pelusa.AsignarEdad(6);
49:     GATO bigotes;
50:     cout << "edad de bigotes: ";
51:     cout << bigotes.ObtenerEdad() << endl;
52:     cout << "copiando pelusa a bigotes...\n";
53:     bigotes = pelusa;
54:     cout << "edad de bigotes: ";
55:     cout << bigotes.ObtenerEdad() << endl;
56:     return 0;
57: }
```

10

**SALIDA**

```
edad de pelusa: 5
Estableciendo edad de pelusa en 6...
edad de bigotes: 5
copiando pelusa a bigotes...
edad de bigotes: 6
```

**ANÁLISIS**

El listado 10.15 utiliza de nuevo la clase GATO y omite el constructor y el destructor de copia para ahorrar espacio. En la línea 17 se declara el operador de asignación, y se define en las líneas 31 a 38.

En la línea 33 se prueba el objeto actual (el GATO que va a ser asignado) para ver si es igual al GATO al que se va a asignar. Esto se hace comprobando si la dirección de rhs es la misma que la dirección guardada en el apuntador this.

Claro que el operador relacional de igualdad (==) también se puede sobrecargar, lo que le permite determinar por usted mismo lo que significa que sus objetos sean iguales.

## Operadores de conversión

¿Qué pasa cuando trata de asignar una variable de un tipo de datos integrado, como int o unsigned short, a un objeto de una clase definida por el usuario? El listado 10.16 vuelve a utilizar la clase Contador e intenta asignar una variable de tipo int a un objeto Contador.


**Precaución**

¡El listado 10.16 no compilará!

**ENTRADA****LISTADO 10.16** Intento de asignar un int a un Contador

```

1: // Listado 10.16
2: // ¡Este código no compilará!
3:
4: #include <iostream.h>
5:
6: class Contador
7: {
8: public:
9:     Contador();
10:    ~Contador(){}
11:    int ObtenerSuVal()const
12:        { return suVal; }
13:    void AsignarSuVal(int x)
14:        { suVal = x; }
15: private:
16:     int suVal;
17: };
18:
19: Contador::Contador():
20:     suVal(0)
21: {}
22:
```

```

23: int main()
24: {
25:     int elShort = 5;
26:     Contador elCtr = elShort;
27:     cout << "elCtr: ";
28:     cout << elCtr.ObtenerSuVal() << endl;
29:     return 0;
30: }

```

**SALIDA**

lst10-16.cxx: In function 'int main ()':  
 lst10-16.cxx:26: conversion from 'int' to non-scalar type 'Contador'  
 requested

10

**ANÁLISIS**

La clase `Contador` declarada en las líneas 6 a 17 sólo tiene un constructor predefinido. No declara un método específico para convertir un `int` en un objeto `Contador`, por lo que la línea 26 produce un error de compilación. El compilador no puede saber, a menos que usted se lo indique, que si hay un `int`, debe asignar ese valor a la variable miembro `suVal`.

La mayoría de los compiladores proporciona un mensaje más corto, algo así como “Unable to convert int to Contador” o “conversion from ‘int’ to non-scalar type ‘Contador’ requested”.

El listado 10.17 corrige esto mediante la creación de un operador de conversión: un constructor que toma un `int` y produce un objeto `Contador`.

**ENTRADA****LISTADO 10.17** Conversión de `int` a `Contador`

```

1: // Listado 10.17
2: // Un constructor como operador de conversión
3:
4: #include <iostream.h>
5:
6: class Contador
7: {
8: public:
9:     Contador();
10:    Contador(int val);
11:    ~Contador(){}
12:    int ObtenerSuVal()const
13:        { return suVal; }
14:    void AsignarSuVal(int x)
15:        { suVal = x; }
16: private:
17:     int suVal;
18: };
19:
20: Contador::Contador():

```

continua

**LISTADO 10.17** CONTINUACIÓN

```
21:     suVal(0)
22: }
23:
24: Contador::Contador(int val):
25:     suVal(val)
26: {
27:
28: int main()
29: {
30:     int elShort = 5;
31:     Contador elCtr = elShort;
32:     cout << "elCtr: ";
33:     cout << elCtr.ObtenerSuVal() << endl;
34:     return 0;
35: }
```

**SALIDA** elCtr: 5

**ANÁLISIS**

El cambio importante se encuentra en la línea 10, en donde el constructor se sobrecarga para tomar un `int`, y en las líneas 24 a 26, en las que se implementa el constructor. El efecto de este constructor es crear un `Contador` a partir de un `int`.

Con esto, el compilador puede llamar al constructor que toma un `int` como argumento. He aquí la forma de hacerlo:

*Paso 1: Crear un contador llamado elCtr.*

Esto es como decir: `int x = 5;`, lo cual crea una variable de tipo entero llamada `x` y luego la inicializa con el valor 5. En este caso, creamos un objeto `Contador` llamado `elCtr` y lo inicializamos con la variable de tipo `int` llamada `elShort`.

*Paso 2: Asignar a elCtr el valor de elShort.*

¡Pero `elShort` es de tipo `int`, no un `contador`! Primero tenemos que convertirlo en un `Contador`. El compilador tratará de hacer ciertas conversiones por usted en forma automática, pero tiene que enseñarle cómo. Para esto, debe crear un constructor para `Contador` que, por ejemplo, tome como su único parámetro un `int`:

```
class Contador
{
    Contador (int x);
    // ...
};
```

Este constructor crea objetos `Contador` a partir de valores de tipo `int`. Hace esto mediante la creación de un *contador* temporal sin nombre. Para fines ilustrativos, suponga que el objeto temporal `Contador` que creamos a partir del tipo `int` se llama `fueShort`.

*Paso 3: Asignar `fueShort` a `elCtr`, lo que equivale a*

```
"elCtr = fueShort";
```

En este paso, `fueShort` (el objeto temporal creado cuando se ejecutó el constructor) se substituye por lo que estaba del lado derecho del operador de asignación. Es decir, ahora que el compilador ha creado una copia temporal por usted, inicializa `elCtr` con ese valor temporal.

Para comprender esto, debe entender que TODAS las sobrecargas de operadores funcionan de la misma forma: se declara un operador sobrecargado mediante el uso de la palabra reservada `operator`. Con los operadores binarios (como `= o +`) la variable del lado derecho se convierte en el parámetro. Esto lo realiza el constructor. Por lo tanto,

```
a = b;
```

se convierte en

```
a.operator=(b);
```

Sin embargo, ¿qué ocurre si trata de invertir la asignación con lo siguiente?

```
1: Contador elCtr(5);
2: int elShort = elCtr;
3: cout << "elShort : " << elShort << endl;
```

De nuevo, esto generará un error de compilación. Aunque el compilador ahora sabe cómo crear un `Contador` a partir de un `int`, no sabe cómo invertir el proceso (a menos que usted le indique cómo).

10

## Cómo crear sus propios operadores de conversión

Para resolver éste y otros problemas similares, C++ proporciona operadores de conversión que se pueden agregar a una clase. Esto permite que la clase especifique cómo hacer conversiones implícitas a tipos de datos integrados. El listado 10.18 muestra esto. Sin embargo, hay una observación: los operadores de conversión no especifican un valor de retorno, aunque realmente sí regresan un valor convertido.

### ENTRADA LISTADO 10.18 Conversión de Contador a unsigned short()

```
1: // Listado 10.18
2: // Uso del operador de conversión
3:
4: #include <iostream.h>
5:
```

continua

**LISTADO 10.18** CONTINUACIÓN

---

```
6: class Contador
7: {
8: public:
9:     Contador();
10:    Contador(int val);
11:    ~Contador(){}
12:    int ObtenerSuVal()const
13:        { return suVal; }
14:    void AsignarSuVal(int x)
15:        { suVal = x; }
16:    operator unsigned short();
17: private:
18:     int suVal;
19: };
20:
21: Contador::Contador():
22:     suVal(0)
23: {}
24:
25: Contador::Contador(int val):
26:     suVal(val)
27: {}
28:
29: Contador::operator unsigned short ()
30: {
31:     return (int (suVal));
32: }
33:
34: int main()
35: {
36:     Contador ctr(5);
37:     int elShort = ctr;
38:
39:     cout << "elShort: " << elShort << endl;
40:     return 0;
41: }
```

---

**SALIDA**

elShort: 5

**ANÁLISIS**

En la línea 16 se declara el operador de conversión. Observe que no tiene valor

de retorno. La implementación de esta función se encuentra en las líneas 29 a 32.

La línea 31 regresa el valor de suVal convertido en un int.

Ahora el compilador sabe cómo convertir variables de tipo int en objetos Contador y viceversa, y se pueden asignar entre sí con libertad.

## Resumen

Hoy aprendió cómo sobrecargar funciones miembro de sus clases. También aprendió cómo proporcionar valores predeterminados para las funciones y cómo decidir cuándo utilizar valores predeterminados y cuándo sobrecargar.

Sobrecargar los constructores de una clase le permite crear clases flexibles que se pueden crear a partir de otras clases. La inicialización de objetos ocurre en la etapa de inicialización del constructor y es más eficiente que asignar valores en el cuerpo del constructor.

El compilador proporciona el constructor de copia y el operador de asignación, si usted no crea los suyos, pero éstos realizan una copia de los datos miembro de la clase. En clases en las que los datos miembro incluyan apuntadores a objetos, variables o funciones almacenadas en el heap, estos métodos se deben pasar por alto y usted deberá asignar la memoria para el objeto destino.

Casi todos los operadores de C++ se pueden sobrecargar, aunque debe tener cuidado de no crear operadores cuyo uso sea contraintuitivo o ambiguo. No puede cambiar la aridad de los operadores, ni puede inventar nuevos operadores.

El apuntador `this` hace referencia al objeto actual y es un parámetro invisible para todas las funciones miembro. A menudo los operadores sobrecargados regresan el apuntador `this` desreferenciado.

Los operadores de conversión le permiten crear clases que se pueden utilizar en expresiones que esperan un tipo distinto de objeto. Con ellos se rompe la regla que establece que todas las funciones regresan un valor explícito; al igual que los constructores y los destructores, no tienen tipo de valor de retorno.

10

## Preguntas y respuestas

- P ¿Por qué utilizar valores predeterminados si se puede sobrecargar una función?**
- R** Es más sencillo mantener una función que mantener dos, y es más fácil entender una función con parámetros predeterminados que estudiar los cuerpos de dos funciones. Además, actualizar una de las funciones sin actualizar la otra es una causa común de errores.
- P Dados los problemas con funciones sobrecargadas, ¿por qué no mejor utilizar siempre valores predeterminados?**
- R** Las funciones sobrecargadas ofrecen capacidades que no están disponibles con las variables predeterminadas; una de ellas es la capacidad de cambiar la lista de parámetros entre diversos tipos, en lugar de omitir algunos de ellos.
- P Al escribir un constructor de clase, ¿cómo decide qué debe colocar en la inicialización y qué debe colocar en el cuerpo del constructor?**
- R** Una regla empírica sencilla es hacer todo lo que sea posible en la fase de inicialización, es decir, inicializar ahí todas las variables miembro. Algunas cosas, como los cálculos y las instrucciones de impresión, deben ir en el cuerpo del constructor.

**P ¿Puede una función sobrecargada tener un parámetro predeterminado?**

**R** Sí. No hay razón alguna para no combinar estas poderosas características. Una o más de las funciones sobrecargadas pueden tener sus propios valores predeterminados, siguiendo las reglas comunes para variables predeterminadas en cualquier función.

**P ¿Por qué algunas funciones miembro se definen dentro de la declaración de la clase y otras no?**

**R** Definir la implementación de una función miembro dentro de la declaración la convierte en función en línea. Por lo general, esto se realiza sólo si la función es extremadamente sencilla. Tome en cuenta que también puede convertir una función miembro en función en línea si utiliza la palabra reservada `inline`, incluso si la función se declara fuera de la declaración de la clase.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. Al sobrecargar funciones miembro de una clase, ¿de qué manera deben diferir?
2. ¿Cuál es la diferencia entre una declaración y una definición?
3. ¿Cuándo se llama al constructor de copia?
4. ¿Cuándo se llama al destructor?
5. ¿Qué diferencia hay entre el constructor de copia y el operador de asignación (=)?
6. ¿Qué es el apuntador `this`?
7. ¿Cómo puede diferenciar entre la sobrecarga de los operadores de incremento de prefijo y los de posfijo?
8. ¿Puede sobrecargar el `operator+` para el tipo de datos `short int`?
9. ¿Es válido en C++ sobrecargar el `operator++` para que decremente un valor de su clase?
10. ¿Qué valor de retorno deben tener los operadores de conversión en sus declaraciones?

## Ejercicios

1. Escriba la declaración de una clase llamada `CirculoSencillo` (únicamente la declaración) con una variable miembro: `suRadio`. Incluya un constructor predeterminado, un destructor y métodos de acceso para la variable `suRadio`.
2. Usando la clase que creó en el ejercicio 1, escriba la implementación del constructor predeterminado, e inicialice `suRadio` con el valor 5.
3. Usando la misma clase, agregue un segundo constructor que tome un valor como parámetro y asigne ese valor a `suRadio`.
4. Cree un operador de incremento de prefijo y uno de posfijo para su clase `CirculoSencillo`, que incrementen `suRadio`.
5. Cambie la clase `CirculoSencillo` para que guarde el dato miembro `suRadio` en el heap, y corrija los métodos existentes.
6. Proporcione un constructor de copia para `CirculoSencillo`.
7. Proporcione un operador de asignación para `CirculoSencillo`.
8. Escriba un programa que cree dos objetos `CirculoSencillo`. Utilice el constructor predeterminado en uno y cree una instancia con el otro que tenga el valor 9. Llame al operador de incremento para que actúe sobre cada uno y luego imprima sus valores. Por último, asigne el segundo al primero e imprima sus valores.
9. **CAZA ERRORES:** ¿Qué está mal en esta implementación del operador de asignación?

```
CUADRADO CUADRADO::operator=(const CUADRADO & rhs)
{
    suLado = new int;
    *suLado = rhs.ObtenerLado();
    return *this;
}
```

10. **CAZA ERRORES:** ¿Qué está mal en esta implementación del operador de suma?

```
MuyCorto MuyCorto::operator+ (const MuyCorto & rhs)
{
    suVal += rhs.ObtenerSuVal();
    return *this;
}
```

10

TABLE II  
Effect of Temperature on the Properties of Poly(1,3-butadiene) Gels

| Temperature, °C. | Mechanical Properties                |                        |
|------------------|--------------------------------------|------------------------|
|                  | Tensile Strength, kg/cm <sup>2</sup> | Elongation at Break, % |
| 25               | 10.0                                 | 100                    |
| 30               | 10.0                                 | 100                    |
| 35               | 10.0                                 | 100                    |
| 40               | 10.0                                 | 100                    |
| 45               | 10.0                                 | 100                    |
| 50               | 10.0                                 | 100                    |
| 55               | 10.0                                 | 100                    |
| 60               | 10.0                                 | 100                    |
| 65               | 10.0                                 | 100                    |
| 70               | 10.0                                 | 100                    |
| 75               | 10.0                                 | 100                    |
| 80               | 10.0                                 | 100                    |
| 85               | 10.0                                 | 100                    |
| 90               | 10.0                                 | 100                    |
| 95               | 10.0                                 | 100                    |
| 100              | 10.0                                 | 100                    |
| 105              | 10.0                                 | 100                    |
| 110              | 10.0                                 | 100                    |
| 115              | 10.0                                 | 100                    |
| 120              | 10.0                                 | 100                    |
| 125              | 10.0                                 | 100                    |
| 130              | 10.0                                 | 100                    |
| 135              | 10.0                                 | 100                    |
| 140              | 10.0                                 | 100                    |
| 145              | 10.0                                 | 100                    |
| 150              | 10.0                                 | 100                    |
| 155              | 10.0                                 | 100                    |
| 160              | 10.0                                 | 100                    |
| 165              | 10.0                                 | 100                    |
| 170              | 10.0                                 | 100                    |
| 175              | 10.0                                 | 100                    |
| 180              | 10.0                                 | 100                    |
| 185              | 10.0                                 | 100                    |
| 190              | 10.0                                 | 100                    |
| 195              | 10.0                                 | 100                    |
| 200              | 10.0                                 | 100                    |
| 205              | 10.0                                 | 100                    |
| 210              | 10.0                                 | 100                    |
| 215              | 10.0                                 | 100                    |
| 220              | 10.0                                 | 100                    |
| 225              | 10.0                                 | 100                    |
| 230              | 10.0                                 | 100                    |
| 235              | 10.0                                 | 100                    |
| 240              | 10.0                                 | 100                    |
| 245              | 10.0                                 | 100                    |
| 250              | 10.0                                 | 100                    |
| 255              | 10.0                                 | 100                    |
| 260              | 10.0                                 | 100                    |
| 265              | 10.0                                 | 100                    |
| 270              | 10.0                                 | 100                    |
| 275              | 10.0                                 | 100                    |
| 280              | 10.0                                 | 100                    |
| 285              | 10.0                                 | 100                    |
| 290              | 10.0                                 | 100                    |
| 295              | 10.0                                 | 100                    |
| 300              | 10.0                                 | 100                    |
| 305              | 10.0                                 | 100                    |
| 310              | 10.0                                 | 100                    |
| 315              | 10.0                                 | 100                    |
| 320              | 10.0                                 | 100                    |
| 325              | 10.0                                 | 100                    |
| 330              | 10.0                                 | 100                    |
| 335              | 10.0                                 | 100                    |
| 340              | 10.0                                 | 100                    |
| 345              | 10.0                                 | 100                    |
| 350              | 10.0                                 | 100                    |
| 355              | 10.0                                 | 100                    |
| 360              | 10.0                                 | 100                    |
| 365              | 10.0                                 | 100                    |
| 370              | 10.0                                 | 100                    |
| 375              | 10.0                                 | 100                    |
| 380              | 10.0                                 | 100                    |
| 385              | 10.0                                 | 100                    |
| 390              | 10.0                                 | 100                    |
| 395              | 10.0                                 | 100                    |
| 400              | 10.0                                 | 100                    |
| 405              | 10.0                                 | 100                    |
| 410              | 10.0                                 | 100                    |
| 415              | 10.0                                 | 100                    |
| 420              | 10.0                                 | 100                    |
| 425              | 10.0                                 | 100                    |
| 430              | 10.0                                 | 100                    |
| 435              | 10.0                                 | 100                    |
| 440              | 10.0                                 | 100                    |
| 445              | 10.0                                 | 100                    |
| 450              | 10.0                                 | 100                    |
| 455              | 10.0                                 | 100                    |
| 460              | 10.0                                 | 100                    |
| 465              | 10.0                                 | 100                    |
| 470              | 10.0                                 | 100                    |
| 475              | 10.0                                 | 100                    |
| 480              | 10.0                                 | 100                    |
| 485              | 10.0                                 | 100                    |
| 490              | 10.0                                 | 100                    |
| 495              | 10.0                                 | 100                    |
| 500              | 10.0                                 | 100                    |
| 505              | 10.0                                 | 100                    |
| 510              | 10.0                                 | 100                    |
| 515              | 10.0                                 | 100                    |
| 520              | 10.0                                 | 100                    |
| 525              | 10.0                                 | 100                    |
| 530              | 10.0                                 | 100                    |
| 535              | 10.0                                 | 100                    |
| 540              | 10.0                                 | 100                    |
| 545              | 10.0                                 | 100                    |
| 550              | 10.0                                 | 100                    |
| 555              | 10.0                                 | 100                    |
| 560              | 10.0                                 | 100                    |
| 565              | 10.0                                 | 100                    |
| 570              | 10.0                                 | 100                    |
| 575              | 10.0                                 | 100                    |
| 580              | 10.0                                 | 100                    |
| 585              | 10.0                                 | 100                    |
| 590              | 10.0                                 | 100                    |
| 595              | 10.0                                 | 100                    |
| 600              | 10.0                                 | 100                    |
| 605              | 10.0                                 | 100                    |
| 610              | 10.0                                 | 100                    |
| 615              | 10.0                                 | 100                    |
| 620              | 10.0                                 | 100                    |
| 625              | 10.0                                 | 100                    |
| 630              | 10.0                                 | 100                    |
| 635              | 10.0                                 | 100                    |
| 640              | 10.0                                 | 100                    |
| 645              | 10.0                                 | 100                    |
| 650              | 10.0                                 | 100                    |
| 655              | 10.0                                 | 100                    |
| 660              | 10.0                                 | 100                    |
| 665              | 10.0                                 | 100                    |
| 670              | 10.0                                 | 100                    |
| 675              | 10.0                                 | 100                    |
| 680              | 10.0                                 | 100                    |
| 685              | 10.0                                 | 100                    |
| 690              | 10.0                                 | 100                    |
| 695              | 10.0                                 | 100                    |
| 700              | 10.0                                 | 100                    |
| 705              | 10.0                                 | 100                    |
| 710              | 10.0                                 | 100                    |
| 715              | 10.0                                 | 100                    |
| 720              | 10.0                                 | 100                    |
| 725              | 10.0                                 | 100                    |
| 730              | 10.0                                 | 100                    |
| 735              | 10.0                                 | 100                    |
| 740              | 10.0                                 | 100                    |
| 745              | 10.0                                 | 100                    |
| 750              | 10.0                                 | 100                    |
| 755              | 10.0                                 | 100                    |
| 760              | 10.0                                 | 100                    |
| 765              | 10.0                                 | 100                    |
| 770              | 10.0                                 | 100                    |
| 775              | 10.0                                 | 100                    |
| 780              | 10.0                                 | 100                    |
| 785              | 10.0                                 | 100                    |
| 790              | 10.0                                 | 100                    |
| 795              | 10.0                                 | 100                    |
| 800              | 10.0                                 | 100                    |
| 805              | 10.0                                 | 100                    |
| 810              | 10.0                                 | 100                    |
| 815              | 10.0                                 | 100                    |
| 820              | 10.0                                 | 100                    |
| 825              | 10.0                                 | 100                    |
| 830              | 10.0                                 | 100                    |
| 835              | 10.0                                 | 100                    |
| 840              | 10.0                                 | 100                    |
| 845              | 10.0                                 | 100                    |
| 850              | 10.0                                 | 100                    |
| 855              | 10.0                                 | 100                    |
| 860              | 10.0                                 | 100                    |
| 865              | 10.0                                 | 100                    |
| 870              | 10.0                                 | 100                    |
| 875              | 10.0                                 | 100                    |
| 880              | 10.0                                 | 100                    |
| 885              | 10.0                                 | 100                    |
| 890              | 10.0                                 | 100                    |
| 895              | 10.0                                 | 100                    |
| 900              | 10.0                                 | 100                    |
| 905              | 10.0                                 | 100                    |
| 910              | 10.0                                 | 100                    |
| 915              | 10.0                                 | 100                    |
| 920              | 10.0                                 | 100                    |
| 925              | 10.0                                 | 100                    |
| 930              | 10.0                                 | 100                    |
| 935              | 10.0                                 | 100                    |
| 940              | 10.0                                 | 100                    |
| 945              | 10.0                                 | 100                    |
| 950              | 10.0                                 | 100                    |
| 955              | 10.0                                 | 100                    |
| 960              | 10.0                                 | 100                    |
| 965              | 10.0                                 | 100                    |
| 970              | 10.0                                 | 100                    |
| 975              | 10.0                                 | 100                    |
| 980              | 10.0                                 | 100                    |
| 985              | 10.0                                 | 100                    |
| 990              | 10.0                                 | 100                    |
| 995              | 10.0                                 | 100                    |
| 1000             | 10.0                                 | 100                    |

# SEMANA 2

DÍA 11

## Herencia

Un aspecto fundamental de la inteligencia humana es buscar, reconocer y crear relaciones entre conceptos. Creamos jerarquías, matrices, redes y otras interrelaciones para explicar y entender la forma en que interactúan las cosas. El lenguaje C++ intenta capturar esto en las jerarquías de herencia. Hoy aprenderá lo siguiente:

- Qué es la herencia
- Cómo derivar una clase de otra
- Qué es el acceso protegido y cómo utilizarlo
- Qué son las funciones virtuales

### Qué es la herencia

¿Qué es un perro? Cuando ve a su mascota, ¿qué es lo que ve? Yo veo cuatro patas al servicio de un hocico. Un biólogo ve una red de órganos que interactúan entre sí, un físico ve átomos y fuerzas trabajando, y un taxónomo ve un representante de la especie *Canis domesticus*.

Esta última valoración es la que nos interesa en este momento. Un perro es un tipo de canino, un canino es un tipo de mamífero, y así sucesivamente. Los taxónomos dividen el mundo de cosas vivientes en reino, filo, clase, orden, familia, género y especie.

Esta jerarquía establece una relación *es un*. Un miembro de la especie *Homo sapiens* *es un* tipo de primate. Esta relación se ve en todas partes: Una vagoneta *es un* tipo de auto, el cual *es un* tipo de vehículo. Una nieve *es un* tipo de postre, el cual *es un* tipo de comida.

Cuando decimos que algo *es un* tipo de otro algo, damos a entender que *es una especialización* de eso. Por ejemplo, un auto *es un* tipo especial de vehículo.

Cualquier objeto que sea del tipo *es un* tiene algunas características de la descripción de orden mayor. Un auto *es un* vehículo. Un camión también *lo es*. Un auto *no es* un camión. Pero ambos tienen ciertas características similares que comparten con la descripción común de *vehículo*. Algunas de estas similitudes heredadas son las llantas, un chasis, un tipo de motor, etc.

## Herencia y derivación

El concepto perro hereda (es decir, obtiene automáticamente) todas las características de un mamífero. Ya que *es un* mamífero, sabemos que *se mueve* y *que respira aire*. Por naturaleza, todos los mamíferos *se mueven* y *respiran aire*. El concepto de un perro añade la idea de ladrar, mover su cola, comerse mis revisiones de esta lección justo cuando ya había terminado, ladrar cuando estoy tratando de dormir, etc.

Podemos dividir a los perros en perros de trabajo, de deporte y Terriers (como el Terrier escocés); y podemos dividir a los perros de deporte en Retrievers (cobradores), Spaniels (como el Cocker Spaniel), y así sucesivamente. Por último, cada uno de éstos se puede especializar aún más; por ejemplo, los Retrievers se pueden subdividir en Labradores y Goldens (dorados).

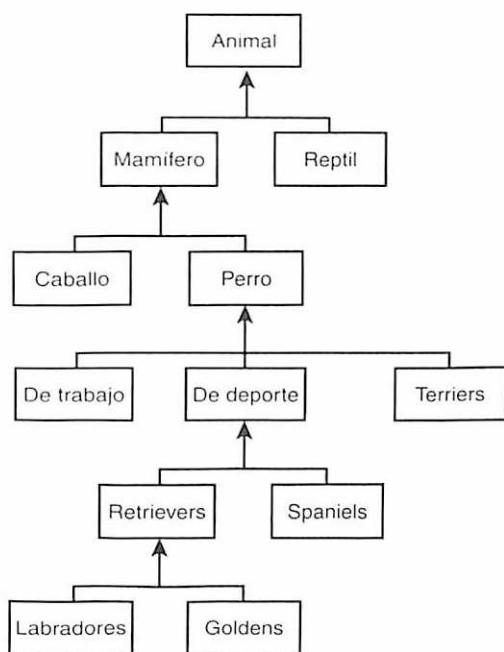
Un Golden *es un* tipo de Retriever, el cual *es un* tipo de perro de deporte, y por lo tanto también *es un* tipo de mamífero, que *es un* tipo de animal y, por ende *un tipo de cosa viviente*. Esta jerarquía se representa en la figura 11.1 usando el UML (Lenguaje de Modelado Uniformado). Las flechas apuntan desde los tipos más especializados a los tipos más generales.

El lenguaje C++ intenta representar estas relaciones permitiendo que usted defina clases que se deriven de otras clases. La derivación es una forma de expresar la relación *es un*. Usted deriva una nueva clase llamada **Perro** de la clase **Mamifero**. No tiene que declarar explícitamente que los perros *se mueven* porque han heredado eso de la clase **Mamifero**.

Se dice que una clase que agrega nueva funcionalidad a una clase existente se deriva de esa clase original. Se dice que la clase original es la clase base de la nueva clase.

Si la clase **Perro** se deriva de la clase **Mamifero**, entonces **Mamifero** es una clase base de **Perro**. Las clases derivadas son superconjuntos de sus clases base. Así como un perro agrega ciertas características a la idea proyectada por un mamífero, la clase **Perro** agrega ciertos métodos o datos a la clase **Mamifero**.

**FIGURA 11.1**  
*Jerarquía de animales.*



Por lo general, una clase base tiene más de una clase derivada. Como los perros, gatos y caballos son tipos de mamíferos, sus clases se derivan de la clase Mamífero.

11

## Cómo crear clases que representen animales

Para facilitar la discusión de la derivación y la herencia, esta lección se enfocará en las relaciones existentes entre una variedad de clases que representan animales. Imagine que le han pedido que diseñe un juego para niños (una simulación de una granja).

A su tiempo desarrollará un conjunto completo de animales de granja, incluyendo caballos, vacas, perros, gatos, ovejas, etc. Creará métodos para estas clases, de forma que puedan actuar como los niños podrían esperar, pero por ahora puede llenar cada método con una simple instrucción de impresión.

Rellenar una función significa que usted escribe sólo lo suficiente para mostrar que la función fue llamada, dejando los detalles para cuando tenga más tiempo. Siéntase libre de extender el poco código proporcionado en esta lección para hacer que los animales actúen en forma más realista.

## La sintaxis de la derivación

Al declarar una clase, puede indicar de qué clase se deriva escribiendo el símbolo de dos puntos (:) después del nombre de la clase, seguido del tipo de derivación (público u otro) y de la clase de la que se deriva. A continuación se muestra un ejemplo:

```
class Perro : public Mamifero
```

Hablaremos sobre el tipo de derivación más adelante en esta lección. Por ahora, utilice siempre el tipo público. Debió declarar con anterioridad la clase de la que deriva, o se generará un error de compilación. El listado 11.1 muestra cómo declarar la clase **Perro** que se deriva de la clase **Mamifero**.

**ENTRADA LISTADO 11.1 Herencia simple**

---

```
1:  //Listado 11.1 Herencia simple
2:
3:  #include <iostream.h>
4:
5:  enum RAZA { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
6:
7:  class Mamifero
8:  {
9:  public:
10:    //constructores
11:    Mamifero();
12:    -Mamifero();
13:    //métodos de acceso
14:    int ObtenerEdad() const;
15:    void AsignarEdad(int);
16:    int ObtenerPeso() const;
17:    void AsignarPeso();
18:    //Otros métodos
19:    void Hablar() const;
20:    void Dormir() const;
21:  protected:
22:    int suEdad;
23:    int suPeso;
24:  };
25:
26:  class Perro : public Mamifero
27:  {
28:  public:
29:    // Constructores
30:    Perro();
31:    -Perro();
32:    // Métodos de acceso
33:    RAZA ObtenerRaza() const;
34:    void AsignarRaza(RAZA);
35:    // Otros métodos
36:    void MoverCola();
37:    void PedirAlimento();
38:  protected:
39:    RAZA suRaza;
40:  };
```

---

**SALIDA**

El programa anterior no tiene salida ya que es sólo un conjunto de declaraciones de clases sin sus implementaciones. No es un programa completo. Sin embargo, hay mucho que ver aquí.

**ANÁLISIS**

En las líneas 7 a 24 se declara la clase `Mamifero`. Observe que en este ejemplo `Mamifero` no se deriva de ninguna otra clase. En el mundo real, los mamíferos sí se derivan de otra clase (es decir, los mamíferos son tipos de animales). En un programa de C++ sólo se puede representar una fracción de la información que se tiene acerca de un objeto dado. La realidad es demasiado compleja como para capturarla en su totalidad, por lo que cada jerarquía de C++ es una representación arbitraria de los datos disponibles. El truco de un buen diseño es representar las áreas de importancia en una forma que se asemeje a la realidad lo más fielmente posible.

La jerarquía tiene que empezar en algún lado; este programa empieza con la clase `Mamifero`. Debido a esta decisión, algunas variables miembro que bien podrían pertenecer a una clase base más alta, ahora se representan aquí. Es evidente que todos los animales tienen una edad y un peso, por ejemplo, si `Mamifero` se deriva de `Animal`, es de esperarse que herede esos atributos. Como no es así, los atributos aparecen en la clase `Mamifero`.

Para mantener el programa sencillo y manejable, sólo se han colocado seis métodos en la clase `Mamifero` (cuatro métodos de acceso, así como `Hablar()` y `Dormir()`).

La clase `Perro` hereda de `Mamifero`, como se indica en la línea 26. Cada objeto `Perro` tendrá tres variables miembro: `suEdad`, `suPeso` y `suRaza`. Observe que la declaración de clase para `Perro` no incluye las variables miembro `suEdad` y `suPeso`. Los objetos `Perro` heredan estas variables de la clase `Mamifero`, junto con todos los métodos de `Mamifero`, excepto el operador de copia, los constructores y el destructor.

**Nota**

Recuerde que las variables miembro también se conocen como datos miembro. Cuando se crea una instancia de la clase, los datos miembro de esa instancia se conocen como propiedades. En el ejemplo anterior, si creamos un objeto de la clase `Perro`, sus propiedades son los valores almacenados en `suPeso`, `suEdad` y `suRaza`. Algunos autores utilizan indistintamente los términos datos miembro, variables miembro y propiedades.

11

## Comparación entre privado y protegido

Tal vez se haya dado cuenta que se introdujo una nueva palabra reservada de acceso, `protected`, en las líneas 21 y 38 del listado 11.1. Anteriormente, los datos de una clase se habían declarado como privados. Sin embargo, los miembros privados no están disponibles para las clases derivadas. Podría hacer que `suEdad` y `suPeso` fueran públicas, pero eso no es deseable. No queremos que otras clases tengan acceso directo a estos datos miembro.

Lo que queremos es una designación que diga: “Hacer estos datos visibles para esta clase y para las clases que se deriven de esta clase”. Esta designación es protegida. Los datos y funciones miembro protegidos son completamente visibles para las clases derivadas, pero en cuanto a lo demás son privados.

En total, existen tres especificadores de acceso: público, protegido y privado. Si una función tiene un objeto de su clase, puede tener acceso a todos los datos y funciones miembro públicos. Las funciones miembro, a su vez, pueden tener acceso a todos los datos y funciones miembro privados de su propia clase, y todos los datos y funciones miembro protegidos de cualquier clase de la que se deriven.

Por lo tanto, la función `Perro::MoverCola()` puede tener acceso al dato protegido `suRaza` y a los datos protegidos de la clase `Mamifero`.

Incluso si hay otras clases interpuestas entre `Mamifero` y `Perro` (por ejemplo, `Animales-Domesticos`), la clase `Perro` aún podrá tener acceso a los miembros protegidos de `Mamifero`, asumiendo que estas otras clases utilicen herencia pública. La herencia privada se explica en el día 15, “Herencia avanzada”.

El listado 11.2 muestra cómo crear objetos de la clase `Perro` y tener acceso a los datos y funciones de esa clase.

---

**ENTRADA LISTADO 11.2 Uso de un objeto derivado**

---

```
1: //Listado 11.2 Uso de un objeto derivado
2:
3: #include <iostream.h>
4:
5: enum RAZA { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
6:
7: class Mamifero
8: {
9: public:
10:    // constructores
11:    Mamifero() : suEdad(2), suPeso(5){}
12:    ~Mamifero(){}
13:    //métodos de acceso
14:    int ObtenerEdad() const
15:        { return suEdad; }
16:    void AsignarEdad(int edad)
17:        { suEdad = edad; }
18:    int ObtenerPeso() const
19:        { return suPeso; }
20:    void AsignarPeso(int peso)
21:        { suPeso = peso; }
22:    //Otros métodos
23:    void Hablar()const
```

```
24:         { cout << "¡Sonido de mamífero!\n"; }
25:     void Dormir()const
26:         { cout << "shhh. Estoy durmiendo.\n"; }
27: protected:
28:     int suEdad;
29:     int suPeso;
30: };
31:
32: class Perro : public Mamifero
33: {
34: public:
35:     // Constructores
36:     Perro() : suRaza(GOLDEN){}
37:     ~Perro(){}
38:     // Métodos de acceso
39:     RAZA ObtenerRaza() const
40:         { return suRaza; }
41:     void AsignarRaza(RAZA raza)
42:         { suRaza = raza; }
43:     // Otros métodos
44:     void MoverCola() const
45:         { cout << "Moviendo la cola...\n"; }
46:     void PedirAlimento() const
47:         { cout << "Pidiendo alimento...\n"; }
48: private:
49:     RAZA suRaza;
50: };
51:
52: int main()
53: {
54:     Perro fido;
55:     fido.Hablar();
56:     fido.MoverCola();
57:     cout << "fido tiene ";
58:     cout << fido.ObtenerEdad() << " años de edad\n";
59:     return 0;
60: }
```

11

**SALIDA**

```
¡Sonido de mamífero!
Moviendo la cola...
fido tiene 2 años de edad
```

**ANÁLISIS**

En las líneas 7 a 30 se declara la clase **Mamifero** (todas sus funciones son en línea para ahorrar espacio aquí). En las líneas 32 a 50 se declara la clase **Perro** como clase derivada de **Mamifero**. Así, debido a estas declaraciones, todos los miembros de la clase **Perro** tienen una edad, un peso y una raza.

En la línea 54 se declara un **Perro** llamado **fido**. **fido** hereda todos los atributos de un **Mamifero**, así como todos los atributos de un **Perro**. Por lo tanto, **fido** sabe cómo **MoverCola()**, pero también sabe cómo **Hablar()** y **Dormir()**.

## Constructores y destructores

Los objetos Perro son objetos Mamífero. Ésta es la esencia de la relación *es un*. Al crear a fido, se llama primero a su constructor base, con lo que se crea un Mamífero. Luego se llama al constructor de Perro, lo que completa la construcción del objeto Perro. Como no le dimos parámetros a Perro, en cada caso se llamó al constructor predeterminado. fido existe hasta que está completamente construido, lo que significa que se deben construir sus partes correspondientes a Mamífero y a Perro. Por lo tanto, es necesario llamar a ambos constructores.

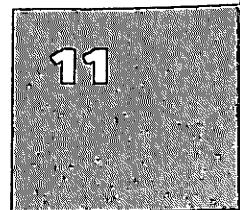
Al destruir a fido, primero se llamará al destructor de Perro y luego al destructor de la parte correspondiente a Mamífero de fido. Cada destructor tiene oportunidad de limpiar su propia parte de fido. ¡Recuerde siempre limpiar todo lo que haga su Perro! El listado 11.3 muestra esto.

### ENTRADA LISTADO 11.3 Llamadas a los constructores y destructores

---

```
1: // Listado 11.3 Llamadas a los constructores y destructores.
2:
3: #include <iostream.h>
4:
5: enum RAZA { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
6:
7: class Mamifero
8: {
9: public:
10:    // constructores
11:    Mamifero();
12:    ~Mamifero();
13:    //métodos de acceso
14:    int ObtenerEdad() const
15:        { return suEdad; }
16:    void AsignarEdad(int edad)
17:        { suEdad = edad; }
18:    int ObtenerPeso() const
19:        { return suPeso; }
20:    void AsignarPeso(int peso)
21:        { suPeso = peso; }
22:    //Otros métodos
23:    void Hablar() const
24:        { cout << "¡Sonido de mamífero!\n"; }
25:    void Dormir() const
26:        { cout << "shhh. Estoy durmiendo.\n"; }
27: protected:
28:    int suEdad;
29:    int suPeso;
30: };
31:
32: class Perro : public Mamifero
```

```
33:  {
34:  public:
35:    // Constructores
36:    Perro();
37:    ~Perro();
38:    // Métodos de acceso
39:    RAZA ObtenerRaza() const
40:      { return suRaza; }
41:    void AsignarRaza(RAZA raza)
42:      { suRaza = raza; }
43:    // Otros métodos
44:    void MoverCola() const
45:      { cout << "Moviendo la cola...\n"; }
46:    void PedirAlimento() const
47:      { cout << "Pidiendo alimento...\n"; }
48: private:
49:    RAZA suRaza;
50: };
51:
52: Mamifero::Mamifero():
53:   suEdad(1),
54:   suPeso(5)
55: {
56:   cout << "Constructor de Mamifero...\n";
57: }
58:
59: Mamifero::~Mamifero()
60: {
61:   cout << "Destructor de Mamifero...\n";
62: }
63:
64: Perro::Perro():
65:   suRaza(GOLDEN)
66: {
67:   cout << "Constructor de Perro...\n";
68: }
69:
70: Perro::~Perro()
71: {
72:   cout << "Destructor de Perro...\n";
73: }
74:
75: int main()
76: {
77:   Perro fido;
78:   fido.Hablar();
79:   fido.MoverCola();
80:   cout << "fido tiene ";
81:   cout << fido.ObtenerEdad() << " años de edad\n";
82:   return 0;
83: }
```



**SALIDA**

Constructor de Mamifero...  
 Constructor de Perro...  
 ¡Sonido de mamífero!  
 Moviendo la cola...  
 fido tiene 1 años de edad  
 Destructor de Perro...  
 Destructor de Mamifero...

**ANÁLISIS**

El listado 11.3 es igual que el listado 11.2, con la excepción de que los constructores y destructores ahora imprimen en la pantalla cuando son llamados. Primero se llama al constructor de Mamifero, y luego al de Perro. En ese momento, el Perro existe en su totalidad, y sus métodos pueden ser llamados. Cuando fido queda fuera de alcance, se hace una llamada al destructor de Perro, seguida de una llamada al destructor de Mamifero.

## Paso de argumentos a los constructores base

Es posible que usted necesite sobrecargar el constructor de Mamifero para que tenga una edad específica, y sobrecargar el constructor de Perro para que tenga una raza. ¿Cómo pasa los parámetros de edad y peso al constructor adecuado de Mamifero? ¿Qué pasa si los objetos Perro quieren inicializar su peso, pero los objetos Mamifero no quieren hacerlo?

La inicialización de la clase base se puede realizar durante la inicialización de la clase, escribiendo el nombre de la clase base, seguido de los parámetros que espera esa clase base. El listado 11.4 muestra esto.

**ENTRADA****LISTADO 11.4** Sobrecarga de constructores de clases derivadas

```

1: //Listado 11.4 Sobrecarga de constructores de clases derivadas
2:
3: #include <iostream.h>
4:
5: enum RAZA { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
6:
7: class Mamifero
8: {
9: public:
10:    // constructores
11:    Mamifero();
12:    Mamifero(int edad);
13:    ~Mamifero();
14:    //métodos de acceso
15:    int ObtenerEdad() const
16:    { return suEdad; }
17:    void AsignarEdad(int edad)
18:    { suEdad = edad; }
19:    int ObtenerPeso() const
20:    { return suPeso; }
21:    void AsignarPeso(int peso)
22:    { suPeso = peso; }
```

```
23:      //Otros métodos
24:      void Hablar() const
25:          { cout << "¡Sonido de mamífero!\n"; }
26:      void Dormir() const
27:          { cout << "shhh. Estoy durmiendo.\n"; }
28:  protected:
29:      int suEdad;
30:      int suPeso;
31:  };
32:
33: class Perro:public Mamífero
34: {
35: public:
36:     // Constructores
37:     Perro();
38:     Perro(int edad);
39:     Perro(int edad, int peso);
40:     Perro(int edad, RAZA raza);
41:     Perro(int edad, int peso, RAZA raza);
42:     ~Perro();
43:     // Métodos de acceso
44:     RAZA ObtenerRaza() const
45:         { return suRaza; }
46:     void AsignarRaza(RAZA raza)
47:         { suRaza = raza; }
48:     // Otros métodos
49:     void MoverCola() const
50:         { cout << "Moviendo la cola...\n"; }
51:     void PedirAlimento() const
52:         { cout << "Pidiendo alimento...\n"; }
53: private:
54:     RAZA suRaza;
55: };
56:
57: Mamífero::Mamífero():
58:     suEdad(1),
59:     suPeso(5)
60: {
61:     cout << "Constructor de Mamífero...\n";
62: }
63:
64: Mamífero::Mamífero(int edad):
65:     suEdad(edad),
66:     suPeso(5)
67: {
68:     cout << "Constructor de Mamífero(int)... \n";
69: }
70:
71: Mamífero::~Mamífero()
72: {
73:     cout << "Destructor de Mamífero...\n";
74: }
```

11

continúa

**LISTADO 11.4** CONTINUACIÓN

```
75:  
76: Perro::Perro():  
77:     Mamifero(),  
78:     suRaza(GOLDEN)  
79: {  
80:     cout << "Constructor de Perro...\n";  
81: }  
82:  
83: Perro::Perro(int edad):  
84:     Mamifero(edad),  
85:     suRaza(GOLDEN)  
86: {  
87:     cout << "Constructor de Perro(int)... \n";  
88: }  
89:  
90: Perro::Perro(int edad, int peso):  
91:     Mamifero(edad),  
92:     suRaza(GOLDEN)  
93: {  
94:     suPeso = peso;  
95:     cout << "Constructor de Perro(int, int)... \n";  
96: }  
97:  
98: Perro::Perro(int edad, int peso, RAZA raza):  
99:     Mamifero(edad),  
100:    suRaza(raza)  
101: {  
102:     suPeso = peso;  
103:     cout << "Constructor de Perro(int, int, RAZA)... \n";  
104: }  
105:  
106: Perro::Perro(int edad, RAZA raza):  
107:     Mamifero(edad),  
108:     suRaza(raza)  
109: {  
110:     cout << "Constructor de Perro(int, RAZA)... \n";  
111: }  
112:  
113: Perro::~Perro()  
114: {  
115:     cout << "Destructor de Perro...\n";  
116: }  
117:  
118: int main()  
119: {  
120:     Perro fido;  
121:     Perro rover(5);  
122:     Perro buster(6, 8);  
123:     Perro yorkie (3, GOLDEN);  
124:     Perro dobbie (4, 20, DOBERMAN);  
125:     fido.Hablar();  
126:     rover.MoverCola();
```

```
127:     cout << "yorkie tiene ";
128:     cout << yorkie.ObtenerEdad() << " años de edad\n";
129:     cout << "dobbie pesa ";
130:     cout << dobbie.ObtenerPeso() << " libras\n";
131:     return 0;
132: }
```

**Nota**

La salida se ha numerado a fin de que en el análisis se tenga una referencia para cada linea.

**SALIDA**

```
1: Constructor de Mamifero...
2: Constructor de Perro...
3: Constructor de Mamifero(int)...
4: Constructor de Perro(int)...
5: Constructor de Mamifero(int)...
6: Constructor de Perro(int, int)...
7: Constructor de Mamifero(int)...
8: Constructor de Perro(int, RAZA)...
9: Constructor de Mamifero(int)...
10: Constructor de Perro(int, int, RAZA)...
11: iSonido de mamifero!
12: Moviendo la cola...
13: yorkie tiene 3 años de edad
14: dobbie pesa 20 libras
15: Destructor de Perro...
16: Destructor de Mamifero...
17: Destructor de Perro...
18: Destructor de Mamifero...
19: Destructor de Perro...
20: Destructor de Mamifero...
21: Destructor de Perro...
22: Destructor de Mamifero...
23: Destructor de Perro...
24: Destructor de Mamifero...
```

**11****ANÁLISIS**

En el listado 11.4 se ha sobrecargado el constructor de **Mamifero** (línea 12) para que tome un entero, que es la edad del **Mamifero**. La implementación de las líneas 64 a 69 inicializa **suEdad** con el valor que se pasa al constructor, e inicializa **suPeso** con 5.

**Perro** ha sobrecargado cinco constructores en las líneas 37 a 41. El primero es el constructor predeterminado. El segundo toma la edad, que es el mismo parámetro que toma el constructor de **Mamifero**. El tercer constructor toma la edad y el peso, el cuarto toma la edad y la raza, y el quinto toma la edad, el peso y la raza.

Observe que en la línea 77, el constructor predeterminado de **Perro** llama al constructor predeterminado de **Mamifero**. Aunque no es estrictamente necesario hacer esto, sirve como información que se haya intentado llamar al constructor base, el cual no toma parámetros.

El constructor base se llamaría de cualquier forma, pero hacerlo en el programa hace que sus intenciones sean explícitas.

La implementación para el constructor de **Perro**, que toma un entero, se encuentra en las líneas 83 a 88. En su fase de inicialización (líneas 84 y 85), **Perro** inicializa su clase base, pasa el parámetro y luego inicializa su raza.

Hay otro constructor de **Perro** en las líneas 90 a 96. Éste toma dos parámetros. Una vez más inicializa su clase base llamando al constructor apropiado, pero esta vez también asigna un peso a la variable **suPeso** de la clase base. Observe que en la fase de inicialización no puede hacer una asignación a la variable de la clase base. Como **Mamifero** no tiene un constructor que tome este parámetro, usted debe hacer esto dentro del cuerpo del constructor de **Perro**.

Analice los constructores restantes para asegurarse de que le gusta la forma en que funcionan. Tome nota de las variables miembro que se inicializan y de las que deben esperar hasta el cuerpo del constructor.

Se ha numerado la salida para que cada línea pueda tener una referencia en este análisis. Las dos primeras líneas de salida representan la instancia de **fido**, usando el constructor predeterminado.

Las líneas 3 y 4 representan la creación de **rover**. Las líneas 5 y 6 representan a **buster**. Observe que el constructor de **Mamifero** que se llamó es el constructor que toma un entero, pero el constructor de **Perro** es el constructor que toma dos enteros.

Después de crear todos los objetos, se utilizan y luego quedan fuera de alcance. A medida que se destruye cada objeto, primero se llama al destructor de **Perro** y luego al de **Mamifero**, siendo en total cinco de cada uno.

## Redefinición de funciones

Un objeto **Perro** tiene acceso a todos los métodos (funciones miembro) de la clase **Mamifero**, así como a cualquier método que la declaración de la clase **Perro** pueda agregar (por ejemplo, **MoverCola()**). Ese objeto también puede redefinir una función de la clase base. Esto significa que la implementación de la función de la clase base cambia en una clase derivada. Cuando se crea un objeto de la clase derivada, se llama a la función apropiada.

Cuando una clase derivada crea una función con el mismo tipo de valor de retorno y de firma que una función miembro de la clase base, pero con una nueva implementación, se dice que está redefiniendo ese método.

Al redefinir una función, debe concordar con el tipo de retorno y de firma de la función de la clase base. La firma es el prototipo de la función sin el tipo de valor de retorno: es decir, el nombre de la función, la lista de parámetros y la palabra reservada **const**, si se utiliza.

El listado 11.5 muestra lo que ocurre cuando la clase **Perro** redefine el método **Hablar()** de **Mamifero**. Para ahorrar espacio, se han omitido las funciones de acceso de estas clases.

**ENTRADA****LISTADO 11.5 Redefinición de un método de la clase base en una clase derivada**

```
1: //Listado 11.5 Redefinición de un método
2: // de la clase base en una clase derivada
3:
4: #include <iostream.h>
5:
6: enum RAZA { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
7:
8: class Mamifero
9: {
10: public:
11:     // constructores
12:     Mamifero()
13:         { cout << "Constructor de Mamifero...\n"; }
14:     ~Mamifero()
15:         { cout << "Destructor de mamifero...\n"; }
16:     //Otros métodos
17:     void Hablar()const
18:         { cout << "¡Sonido de mamifero!\n"; }
19:     void Dormir()const
20:         { cout << "shhh. Estoy durmiendo.\n"; }
21: protected:
22:     int suEdad;
23:     int suPeso;
24: };
25:
26: class Perro : public Mamifero
27: {
28: public:
29:     // Constructores
30:     Perro()
31:         { cout << "Constructor de Perro...\n"; }
32:     ~Perro()
33:         { cout << "Destructor de Perro...\n"; }
34:     // Otros métodos
35:     void MoverCola() const
36:         { cout << "Moviendo la cola...\n"; }
37:     void PedirAlimento() const
38:         { cout << "Pidiendo alimento...\n"; }
39:     void Hablar() const
40:         { cout << "¡Guau!\n"; }
41: private:
42:     RAZA suRaza;
43: };
44:
45: int main()
46: {
47:     Mamifero animalGrande;
48:     Perro fido;
49:
50:     animalGrande.Hablar();
51:     fido.Hablar();
52:     return 0;
53: }
```

**SALIDA**

```

Constructor de Mamifero...
Constructor de Mamifero...
Constructor de Perro...
¡Sonido de mamifero!
¡Guau!
Destructor de Perro...
Destructor de mamifero...
Destructor de mamifero...

```

**ANÁLISIS**

En la línea 39, la clase Perro redefine el método `Hablar()`, lo que ocasiona que los objetos Perro “digan” ¡Guau! cuando se llama al método `Hablar()`. En la línea 47 se crea un objeto de la clase Mamifero llamado `animalGrande`, lo que produce la primera línea de salida cuando se hace la llamada al constructor de Mamifero. En la línea 48 se crea un objeto de la clase Perro llamado `fido`, lo que produce las siguientes dos líneas de salida, en donde se llama al constructor de Mamifero y luego al de Perro. En la línea 50, el objeto `animalGrande` llama a su método `Hablar()`; luego, en la línea 51, el objeto `fido` llama a su método `Hablar()`. La salida refleja que se llamó a los métodos correctos. Por último, los dos objetos quedan fuera de alcance y se llama a los destructores.

**Sobrecarga en comparación con redefinición**

Estos términos son similares, y hacen cosas similares. Al sobrecargar un método, se crea más de un método con el mismo nombre, pero con diferente firma. Al redefinir un método, se crea un método en una clase derivada con el mismo nombre que un método de la clase base, y con la misma firma.

**Cómo ocultar el método de la clase base**

En el listado anterior, el método `Hablar()` de la clase Perro oculta al método de la clase base. Esto es lo que se quiere, pero se pueden tener resultados inesperados. Si Mamifero tiene un método llamado `Mover()`, el cual está sobrecargado, y Perro redefine ese método, el método de Perro ocultará todos los métodos de la clase Mamifero que tengan ese nombre.

Si Mamifero sobrecarga a `Mover()` con tres métodos (uno que no lleve parámetros, uno que tome un entero y otro que tome un entero y una dirección) y Perro redefine sólo el método `Mover()` que no lleva parámetros, no será fácil tener acceso a los otros dos métodos usando un objeto Perro. El listado 11.6 ilustra este problema.

**ENTRADA****LISTADO 11.6** Ocultamiento de métodos

```

1: //Listado 11.6 Ocultamiento de métodos
2:
3: #include <iostream.h>
4:
5: class Mamifero

```

```

6:  {
7:  public:
8:    void Mover() const
9:      { cout << "Mamifero se mueve un paso\n"; }
10:   void Mover(int distancia) const
11:   {
12:     cout << "Mamifero se mueve ";
13:     cout << distancia << " pasos.\n";
14:   }
15: protected:
16:   int suEdad;
17:   int suPeso;
18: };
19:
20: class Perro : public Mamifero
21: {
22: public:
23:   // iOtros compiladores tal vez emitan una advertencia
24:   // de que se está ocultando una función!
25:   void Mover() const
26:     { cout << "Perro se mueve 5 pasos.\n"; }
27: };
28:
29: int main()
30: {
31:   Mamifero animalGrande;
32:   Perro fido;
33:
34:   animalGrande.Mover();
35:   animalGrande.Mover(2);
36:   fido.Mover();
37:   // fido.Mover(10);
38:   return 0;
39: }

```

11

**SALIDA**

Mamifero se mueve un paso  
 Mamifero se mueve 2 pasos.  
 Perro se mueve 5 pasos.

**ANÁLISIS**

Todos los métodos y datos adicionales se han omitido en estas clases. En las líneas 8 y 10, la clase `Mamifero` declara los métodos `Mover()` sobrecargados. En la línea 25, `Perro` redefine la versión del método `Mover()` que no lleva parámetros. Estos métodos se invocan en las líneas 34 a 36, y la salida refleja esto a medida que se ejecuta el programa.

Sin embargo, la línea 37 se convirtió en comentario porque produce un error en tiempo de compilación. Aunque la clase `Perro` podría haber llamado al método `Mover(int)` si no hubiera redefinido la versión del `Mover()` que no lleva parámetros, ahora que lo ha hecho debe redefinir ambos si quiere usarlos. De no ser así, *ocultará* el método que no redefina. Esto es un recordatorio de la regla que establece que si usted proporciona un constructor, el compilador no proporcionará un constructor predeterminado. Los compiladores GNU producen el siguiente mensaje si no convierte la línea 37 en comentario:

```
lst11-06.cxx: In function `int main()':  
lst11-06.cxx:37: too many arguments for method `void Perro::Mover() const'
```

La regla es ésta: cuando redefine cualquier método sobrecargado, las demás sobrecargas de ese método quedan ocultas. Si no quiere que se oculten, debe redefinirlas a todas.

Es un error común ocultar un método de la clase base cuando se trata de redefinirlo, al olvidar incluir la palabra reservada `const`. `const` es parte de la firma, y al omitirla cambia la firma y, por consecuencia, se oculta el método en lugar de redefinirlo.

#### Redefinición en comparación con ocultamiento

En la siguiente sección se describen los métodos virtuales. La redefinición de un método virtual soporta el polimorfismo (si se oculta, se debilita el polimorfismo). Verá más sobre esto muy pronto.

## Cómo llamar al método base

Si ha redefinido el método base, aún puede llamarlo especificando completamente el nombre del método. Esto se hace escribiendo el nombre base, seguido de dos símbolos de dos puntos (:) y del nombre del método; por ejemplo: `Mamifero::Mover()`.

Se podría modificar la línea 37 del el listado 11.6 para que pudiera compilar, escribiendo lo siguiente:

```
37:     fido.Mamifero::Mover(10);
```

Esto llama explícitamente al método de `Mamifero`. El listado 11.7 ilustra detalladamente esta idea.

#### ENTRADA

#### LISTADO 11.7 Cómo llamar al método base desde el método redefinido

```
1: //Listado 11.7 Cómo llamar al método base desde el método redefinido.  
2:  
3: #include <iostream.h>  
4:  
5: class Mamifero  
6: {  
7: public:  
8:     void Mover() const  
9:     { cout << "Mamifero se mueve un paso\n"; }  
10:    void Mover(int distancia) const  
11:    {  
12:        cout << "Mamifero se mueve " << distancia;  
13:        cout << " pasos.\n";  
14:    }  
15: protected:
```

```

16:     int suEdad;
17:     int suPeso;
18: };
19:
20: class Perro : public Mamifero
21: {
22: public:
23:     void Mover()const;
24: };
25:
26: void Perro::Mover() const
27: {
28:     cout << "En perro se mueve...\n";
29:     Mamifero::Mover(3);
30: }
31:
32: int main()
33: {
34:     Mamifero animalGrande;
35:     Perro fido;
36:
37:     animalGrande.Mover(2);
38:     fido.Mamifero::Mover(6);
39:     return 0;
40: }
```

11

**SALIDA**

Mamifero se mueve 2 pasos.  
 Mamifero se mueve 6 pasos.

**ANÁLISIS**

En la línea 34 se crea un Mamifero llamado `animalGrande`, y en la línea 35 se crea un Perro llamado `fido`. La llamada al método de la línea 37 invoca al método `Mover()` de `Mamifero`, el cual toma un `int` como parámetro.

El programador quería invocar a `Mover(int)` en el objeto `Perro`, pero tuvo un problema. `Perro` redefine el método `Mover()`, pero no lo sobrecarga y no proporciona una versión que tome un `int` como parámetro. Esto se resuelve por medio de la llamada explícita al método `Mover(int)` de la clase base, el cual se encuentra en la línea 38.

**DEBE**

**DEBE** extender la funcionalidad de clases ya probadas mediante la derivación.

**DEBE** cambiar el comportamiento de ciertas funciones de la clase derivada, redefiniendo los métodos de la clase base.

**NO DEBE**

**NO DEBE** ocultar una función de la clase base cambiando la firma de la función.

## Métodos virtuales

Esta lección recalca el hecho de que el objeto Perro es un objeto Mamífero. Hasta ahora eso sólo ha significado que el objeto Perro ha heredado los atributos (datos) y capacidades (métodos) de su clase base. Sin embargo, en C++ la relación *es un* va más allá de eso.

C++ extiende su polimorfismo para permitir que apuntadores a clases base se asignen a objetos de las clases derivadas. Por lo tanto, puede escribir

```
Mamifero * apMamifero = new Perro;
```

Esto crea un nuevo objeto Perro en el heap y regresa un apuntador a ese objeto, el cual se asigna a un apuntador a Mamífero. Esto está bien porque un perro es un mamífero.

### Nota

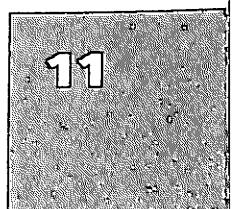
Ésta es la esencia del polimorfismo. Por ejemplo, podría crear muchos tipos de ventanas, incluyendo cuadros de diálogo, ventanas desplazables y cuadros de lista, y darle a cada una de ellas un método virtual llamado dibujar(). Al crear un apuntador a una ventana, y asignarle a ese apuntador cuadros de diálogo y otros tipos derivados, puede llamar a dibujar() sin preocuparse por el tipo en tiempo de ejecución del objeto al que se apunta. Se llamará a la función dibujar() apropiada.

Entonces puede usar este apuntador para invocar a cualquier método de Mamífero. Lo que quiere es que esos métodos que están redefinidos en Perro() llamen a la función correcta. Las funciones virtuales le permiten hacer esto. El listado 11.8 muestra la forma en que esto funciona, y lo que ocurre con métodos que no son virtuales.

### ENTRADA LISTADO 11.8 Uso de métodos virtuales

```
1: //Listado 11.8 Uso de métodos virtuales
2:
3: #include <iostream.h>
4:
5: class Mamifero
6: {
7: public:
8:     Mamifero() : suEdad(1)
9:     { cout << "Constructor de Mamifero...\n"; }
10:    virtual ~Mamifero()
11:    { cout << "Destructor de Mamifero...\n"; }
12:    void Mover() const
13:    { cout << "Mamifero se mueve un paso\n"; }
14:    virtual void Hablar() const
15:    { cout << "¡Mamifero habla!\n"; }
16: protected:
17:     int suEdad;
```

```
18:  };
19:
20: class Perro : public Mamifero
21: {
22: public:
23:     Perro()
24:         { cout << "Constructor de Perro...\n"; }
25:     virtual ~Perro()
26:         { cout << "Destructor de Perro...\n"; }
27:     void MoverCola()
28:         { cout << "Moviendo la cola...\n"; }
29:     void Hablar()const
30:         { cout << "¡Guau!\n"; }
31:     void Mover()const
32:         { cout << "Perro se mueve 5 pasos...\n"; }
33: };
34:
35: int main()
36: {
37:
38:     Mamifero * apPerro = new Perro;
39:
40:     apPerro->Mover();
41:     apPerro->Hablar();
42:     return 0;
43: }
```

11**SALIDA**

```
Constructor de Mamifero...
Constructor de Perro...
Mamifero se mueve un paso
¡Guau!
```

**ANÁLISIS**

En la línea 14 se proporciona un método virtual, `Hablar()`, a la clase `Mamifero`.

Con esto, el diseñador de esta clase indica que espera que esta clase se convierta eventualmente en el tipo base de otra clase. Probablemente, la clase derivada necesitará redefinir esta función.

En la línea 38 se crea un apuntador a `Mamifero` (`apPerro`), pero se le asigna la dirección de un nuevo objeto `Perro`. Como un perro es un mamífero, ésta es una asignación válida. El apuntador se utiliza entonces para llamar a la función `Mover()`. Como el compilador sabe que `apPerro` sólo es un `Mamifero`, busca el método `Mover()` en el objeto `Mamifero`.

En la línea 41, el apuntador llama al método `Hablar()`. Como `Hablar()` es virtual, se invoca al método redefinido `Hablar()` de la clase `Perro`.

Esto es casi mágico. En lo que a la función respecta, tiene un apuntador a `Mamifero`, pero aquí se llamó a un método de `Perro`. De hecho, si se tuviera un arreglo de apuntadores a `Mamifero`, y cada uno apuntara a una subclase de `Mamifero`, se podrían llamar uno por uno, y cada vez se llamaría a la función correcta. El listado 11.9 ilustra esta idea.

**ENTRADA LISTADO 11.9 Varias funciones virtuales llamadas una por una**

```
1:  //Listado 11.9 Varias funciones virtuales llamadas una por una
2:
3:  #include <iostream.h>
4:
5:  class Mamifero
6:  {
7:  public:
8:      Mamifero() : suEdad(1) {}
9:      virtual ~Mamifero() {}
10:     virtual void Hablar() const
11:         { cout << "iMamifero habla!\n"; }
12: protected:
13:     int suEdad;
14: };
15:
16: class Perro : public Mamifero
17: {
18: public:
19:     void Hablar()const
20:         { cout << "iGuau!\n"; }
21: };
22:
23: class Gato : public Mamifero
24: {
25: public:
26:     void Hablar()const
27:         { cout << "iMiau!\n"; }
28: };
29:
30: class Caballo : public Mamifero
31: {
32: public:
33:     void Hablar()const
34:         { cout << "iYihii!\n"; }
35: };
36:
37: class Cerdo: public Mamifero
38: {
39: public:
40:     void Hablar()const
41:         { cout << "iOink!\n"; }
42: };
43:
44: int main()
45: {
46:     Mamifero * elArreglo[ 5 ];
47:     Mamifero * aptr;
48:     int opcion, i;
49:
50:     for (i = 0; i < 5; i++)
51:     {
52:         cout << "(1)perro (2)gato (3)caballo (4)cerdo: ";
53:         cin >> opcion;
```

```
54:         switch (opcion)
55:     {
56:         case 1:
57:             aptr = new Perro;
58:             break;
59:         case 2:
60:             aptr = new Gato;
61:             break;
62:         case 3:
63:             aptr = new Caballo;
64:             break;
65:         case 4:
66:             aptr = new Cerdo;
67:             break;
68:         default:
69:             aptr = new Mamifero;
70:             break;
71:     }
72:     elArreglo[ i ] = aptr;
73: }
74: for (i = 0; i < 5; i++)
75:     elArreglo[ i ]->Hablar();
76: return 0;
77: }
```

11

**SALIDA**

```
(1)perro (2)gato (3)caballo (4)cerdo: 1
(1)perro (2)gato (3)caballo (4)cerdo: 2
(1)perro (2)gato (3)caballo (4)cerdo: 3
(1)perro (2)gato (3)caballo (4)cerdo: 4
(1)perro (2)gato (3)caballo (4)cerdo: 5
¡Guau!
¡Miau!
¡Yihii!
¡Oink!
¡Mamifero habla!
```

**ANÁLISIS**

Este programa simplificado que proporciona sólo la funcionalidad más básica para cada clase, muestra un ejemplo de las funciones virtuales en su forma más pura. Se declaran cuatro clases: *Perro*, *Gato*, *Caballo* y *Cerdo*, todas derivadas de *Mamifero*.

En la línea 10 se declara como virtual la función *Hablar()* de *Mamifero*. En las líneas 19, 26, 33 y 40, las cuatro clases derivadas redefinen la implementación de *Hablar()*.

Se pide al usuario que elija los objetos a crear, y los apuntadores se agregan al arreglo en las líneas 50 a 73.

**Nota**

En tiempo de compilación, es imposible saber cuáles objetos se crearán, y por ende cuáles métodos *Hablar()* se invocarán. El apuntador *aptr* está ligado a su objeto en tiempo de compilación. Esto se conoce como **vinculación dinámica** o **vinculación en tiempo de ejecución**, a diferencia de la **vinculación estática** o **vinculación en tiempo de compilación**.

**Preguntas frecuentes**

**FAQ:** Si marco un método miembro como virtual en la clase base, ¿necesito marcarlo también como virtual en clases derivadas?

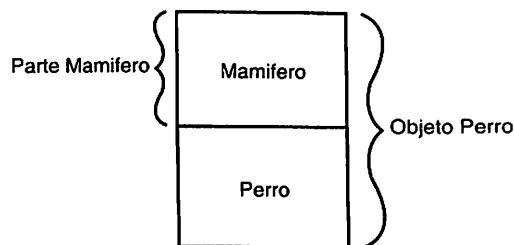
**Respuesta:** No. Cuando un método es virtual, si lo redefine en clases derivadas, sigue siendo virtual. Es una buena idea (aunque no es obligatorio) seguir marcándolo como virtual; esto hace que el código sea más fácil de entender.

## Cómo trabajan las funciones virtuales

Cuando se crea un objeto derivado, por ejemplo, un objeto Perro, primero se llama al constructor de la clase base, y luego se llama al constructor de la clase derivada. La figura 11.2 muestra cómo se ve el objeto Perro después de ser creado. Observe que la parte Mamífero del objeto está junto a la memoria de la parte Perro.

**FIGURA 11.2**

*El objeto Perro después de ser creado.*



Cuando se crea una función virtual dentro de un objeto, el objeto debe estar al pendiente de esa función. Muchos compiladores crean una *tabla de funciones virtuales*, conocida como tabla v. Se mantiene una de éstas para cada tipo, y cada objeto de ese tipo mantiene un apuntador de tabla virtual (conocido como aptrv o apuntador v), el cual apunta a esa tabla.

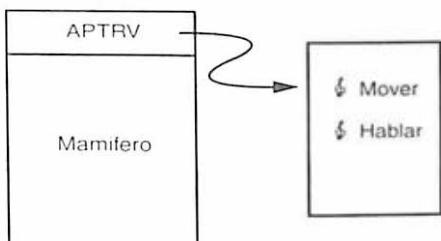
Aunque las implementaciones varían, todos los compiladores deben realizar lo mismo, por lo que no estamos del todo mal con esta descripción.

Cada aptrv de cada objeto apunta a la tabla v que, a su vez, tiene un apuntador a cada una de las funciones virtuales. (Nota: Hablaremos detalladamente de los apuntadores a funciones en el día 14, “Clases y funciones especiales”.) Cuando se crea la parte Mamífero del objeto Perro, el aptrv se inicializa para apuntar a la parte correcta de la tabla v, como se muestra en la figura 11.3.

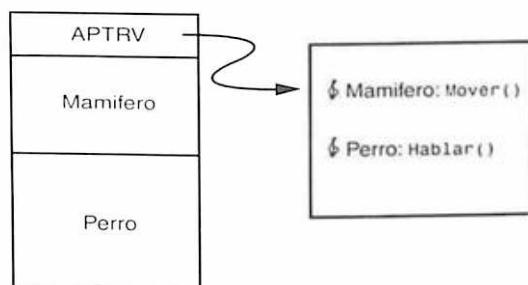
Cuando se llama al constructor de Perro y se agrega la parte Perro de este objeto, el aptrv se ajusta para apuntar a las redefiniciones (si existe alguna) de la función virtual del objeto Perro (vea la figura 11.4).

**FIGURA 11.3**

*La tabla-v de un Mamífero.*

**FIGURA 11.4**

*La tabla-v de un Perro.*



Cuando se utiliza un apuntador a un *Mamífero*, el *aptrv* sigue apuntando a la función correcta, dependiendo del tipo “real” del objeto. Por lo tanto, cuando se invoca a la función *Hablar()*, se invoca a la función correcta.

11

### No puede llegar allá desde aquí

Si el objeto *Perro* tuviera un método llamado *MoverCola()* que no existiera en la clase *Mamífero*, no podría utilizar el apuntador a *Mamífero* para tener acceso a ese método (a menos que lo convierta en apuntador a *Perro*). Como *MoverCola()* no es una función virtual, y como no se encuentra en un objeto *Mamífero*, usted no puede llegar a la función sin un objeto *Perro* o sin un apuntador a *Perro*.

Aunque puede transformar el apuntador a *Mamífero* en apuntador a *Perro*, por lo general existen formas mejores y más seguras de llamar al método *MoverCola()*. C++ desaprueba las conversiones explícitas porque son propensas a errores. Este tema se explicará en detalle cuando hablemos sobre la herencia múltiple en el día 13, “Polimorfismo”, y también cuando hablemos sobre las plantillas en el día 20, “Excepciones y manejo de errores”.

## Partición de datos

La partición de datos es lo que ocurre cuando el compilador selecciona la función virtual que es parte de una clase base en lugar de una función que se encuentre en una clase derivada. Por lo general, necesitaría ejecutar una función con el mismo nombre en una clase derivada. A menudo, este proceso se conoce con el término *magia de la función virtual*.

Hay que tener en cuenta que la magia de la función virtual opera sólo en apunadores y referencias. Si se pasa un objeto por valor, no se podrá invocar a las funciones virtuales. El listado 11.10 ejemplifica este problema.

**ENTRADA LISTADO 11.10 Partición de datos al pasar parámetros por valor**

```
1: //Listado 11.10 Partición de datos al pasar parámetros por valor
2:
3: #include <iostream.h>
4:
5: class Mamifero
6: {
7: public:
8:     Mamifero() : suEdad(1) {}
9:     virtual ~Mamifero() {}
10:    virtual void Hablar() const
11:        { cout << "iMamifero habla!\n"; }
12: protected:
13:     int suEdad;
14: };
15:
16: class Perro : public Mamifero
17: {
18: public:
19:     void Hablar()const
20:         { cout << "iGuau!\n"; }
21: };
22:
23: class Gato : public Mamifero
24: {
25: public:
26:     void Hablar()const
27:         { cout << "iMiau!\n"; }
28: };
29:
30: void ValorFuncion(Mamifero);
31: void AptrFuncion(Mamifero *);
32: void RefFuncion(Mamifero &);
33:
34: int main()
35: {
36:     Mamifero * aptr = NULL;
37:     int opcion;
38:     while (1)
39:     {
40:         bool fSalir = false;
41:         cout << "(1)perro (2)gato (0)Salir: ";
42:         cin >> opcion;
43:         switch (opcion)
44:         {
45:             case 0:
46:                 fSalir = true;
47:                 break;
48:             case 1:
```

```

49:             aptr = new Perro;
50:             break;
51:         case 2:
52:             aptr = new Gato;
53:             break;
54:         default:
55:             aptr = new Mamifero;
56:             break;
57:         }
58:         if (fSalir)
59:             break;
60:         AptrFuncion(aptr);
61:         RefFuncion(*aptr);
62:         ValorFuncion(*aptr);
63:     }
64:     return 0;
65: }
66:
67: void ValorFuncion(Mamifero MamiferoValor)
68: {
69:     MamiferoValor.Hablar();
70: }
71:
72: void AptrFuncion(Mamifero * apMamifero)
73: {
74:     apMamifero->Hablar();
75: }
76:
77: void RefFuncion(Mamifero & rMamifero)
78: {
79:     rMamifero.Hablar();
80: }

```

11

**SALIDA**

```

(1)perro (2)gato (0)Salir: 1
¡Guau!
¡Guau!
¡Mamifero habla!
(1)perro (2)gato (0)Salir: 2
¡Miau!
¡Miau!
¡Mamifero habla!
(1)perro (2)gato (0)Salir: 0

```

**ANÁLISIS**

En las líneas 5 a 28 se declaran versiones simplificadas de las clases Mamifero.

**Perro** y **Gato**. Se declaran tres funciones, **AptrFuncion()**, **RefFuncion()** y **ValorFuncion()**. Toman un apuntador a Mamifero, una referencia a Mamifero y un objeto Mamifero, respectivamente. Las tres funciones hacen lo mismo: llaman al método **Hablar()**.

Se pide al usuario que elija un **Perro** o un **Gato**, y con base en la opción que elija, se crea un apuntador al tipo correcto en las líneas 48 a 53.

En la primera línea de la salida, el usuario elige Perro. El objeto Perro se crea en el heap en la línea 49. Luego se pasa como apuntador, como referencia y por valor a las tres funciones. El apuntador y la referencia invocan a las funciones virtuales, y se invoca a la función miembro Perro->Hablar(). Esto se muestra en las dos primeras líneas de salida después de la elección del usuario.

Sin embargo, el apuntador desreferenciado se pasa por valor. La función espera recibir un objeto Mamifero, por lo que el compilador parte el objeto Perro dejando sólo la parte Mamifero. En ese momento, se hace una llamada al método Hablar() de Mamifero, como se refleja en la tercera línea de salida después de la elección del usuario.

Este experimento se repite para el objeto Gato, con resultados similares.

## Destuctores virtuales

Es válido y común pasar un apuntador a un objeto derivado cuando se espera un apuntador a un objeto base. ¿Qué pasa cuando ese apuntador a un objeto derivado se elimina? Si el destructor es virtual, como debe ser, ocurre lo correcto (se llama al destructor de la clase derivada). Debido a que el destructor de la clase derivada invocará automáticamente al destructor de la clase base, todo el objeto se destruirá de manera apropiada.

La regla empírica es esta: si alguna de las funciones de su clase es virtual, el destructor también debe ser virtual.

## Constructores virtuales de copia

Los constructores no pueden ser virtuales, así que, técnicamente, no existe un constructor virtual de copia. Sin embargo, algunas veces su programa necesita con desesperación pasar un apuntador a un objeto base como parámetro y tener una copia del objeto derivado correcto que se crea. Una solución común para este problema es crear un método Clonar() en la clase base y hacerlo virtual. El método Clonar() crea una copia del nuevo objeto de la clase actual y regresa ese objeto.

Como cada clase derivada redefine el método Clonar(), se crea una copia de la clase derivada. El listado 11.11 muestra cómo se utiliza esto.

---

**ENTRADA LISTADO 11.11 Constructor virtual de copia**

---

```
1: //Listado 11.11 Constructor virtual de copia
2:
3: #include <iostream.h>
4:
5: enum ANIMALES { MAMIFERO, PERRO, GATO};
6: const int NumTiposAnimales = 3;
7:
8: class Mamifero
9: {
10: public:
11:     Mamifero() : suEdad(1)
```

```
12:      { cout << "Constructor de Mamifero...\n"; }
13:      virtual ~Mamifero()
14:      { cout << "Destructor de Mamifero...\n"; }
15:      Mamifero(const Mamifero & rhs);
16:      virtual void Hablar() const
17:      { cout << "¡Mamifero habla!\n"; }
18:      virtual Mamifero * Clonar()
19:      { return new Mamifero(*this); }
20:      int ObtenerEdad()const
21:      { return suEdad; }
22: protected:
23:     int suEdad;
24: };
25:
26: Mamifero::Mamifero(const Mamifero & rhs):suEdad(rhs.ObtenerEdad())
27: {
28:     cout << "Constructor de copia de Mamifero...\n";
29: }
30:
31: class Perro:public Mamifero
32: {
33: public:
34:     Perro()
35:     { cout << "Constructor de Perro...\n"; }
36:     virtual ~Perro()
37:     { cout << "Destructor de Perro...\n"; }
38:     Perro(const Perro & rhs);
39:     void Hablar()const
40:     { cout << "¡Guau!\n"; }
41:     virtual Mamifero * Clonar()
42:     { return new Perro(*this); }
43: };
44:
45: Perro::Perro(const Perro & rhs):
46:     Mamifero(rhs)
47: {
48:     cout << "Constructor de copia de Perro...\n";
49: }
50:
51: class Gato:public Mamifero
52: {
53: public:
54:     Gato()
55:     { cout << "Constructor de Gato...\n"; }
56:     ~Gato()
57:     { cout << "Destructor de Gato...\n"; }
58:     Gato(const Gato & );
59:     void Hablar()const
60:     { cout << "¡Miau!\n"; }
61:     virtual Mamifero * Clonar()
62:     { return new Gato(*this); }
63: };
64:
65: Gato::Gato(const Gato & rhs):
66:     Mamifero(rhs)
```

**LISTADO 11.11** CONTINUACIÓN

```
67:  {
68:    cout << "Constructor de copia de Gato...\n";
69:  }
70:
71: int main()
72: {
73:   Mamifero * elArreglo[ NumTiposAnimales ];
74:   Mamifero * aptr;
75:   int opcion, i;
76:
77:   for (i = 0; i < NumTiposAnimales; i++)
78:   {
79:     cout << "(1)perro (2)gato (3)Mamifero: ";
80:     cin >> opcion;
81:     switch (opcion)
82:     {
83:       case PERRITO:
84:         aptr = new Perro;
85:         break;
86:       case GATITO:
87:         aptr = new Gato;
88:         break;
89:       default:
90:         aptr = new Mamifero;
91:         break;
92:     }
93:     elArreglo[ i ] = aptr;
94:   }
95:   Mamifero * OtroArreglo[ NumTiposAnimales ];
96:   for (i = 0; i < NumTiposAnimales; i++)
97:   {
98:     elArreglo[ i ]->Hablar();
99:     OtroArreglo[ i ] = elArreglo[ i ]->Clonar();
100:
101:    for (i = 0; i < NumTiposAnimales; i++)
102:      OtroArreglo[ i ]->Hablar();
103:
104:  }
```

**SALIDA**

```
1: (1)perro (2)gato (3)Mamifero: 1
2: Constructor de Mamifero...
3: Constructor de Perro...
4: (1)perro (2)gato (3)Mamifero: 2
5: Constructor de Mamifero...
6: Constructor de Gato...
7: (1)perro (2)gato (3)Mamifero: 3
8: Constructor de Mamifero...
9: ¡Guau!
10: Constructor de copia de Mamifero...
11: Constructor de copia de Perro...
12: ¡Miau!
13: Constructor de copia de Mamifero...
```

```
14: Constructor de copia de Gato...
15: ¡Mamífero habla!
16: Constructor de copia de Mamífero...
17: ¡Guau!
18: ¡Miau!
19: ¡Mamífero habla!
```

**ANÁLISIS**

El listado 11.11 es muy similar a los dos listados anteriores, con la excepción de que se ha agregado un nuevo método virtual a la clase `Mamífero: Clonar()`. Este método regresa un apuntador a un nuevo objeto `Mamífero` mediante una llamada al constructor de copia, pasándose a sí mismo (`*this`) como referencia `const`.

Tanto `Perro` como `Gato` redefinen el método `Clonar()`, inicializando sus datos y pasando copias de sí mismos a sus propios constructores de copia. Como `Clonar()` es virtual, esto creará en efecto un constructor virtual de copia, como se muestra en la línea 99.

Se pide al usuario que elija entre perros, gatos o mamíferos, los cuales se crean en las líneas 77 a 94. En la línea 95 se guarda en un arreglo un apuntador para cada opción.

A medida que el programa itera sobre el arreglo, se llama a los métodos `Hablar()` y `Clonar()` de cada objeto, uno por uno, en las líneas 98 y 99. El resultado de la llamada a `Clonar()` es un apuntador a una copia del objeto, la cual se guarda a continuación en un segundo arreglo en la línea 99.

En la línea 1 de la salida se pide al usuario un valor y responde con 1, con el que elige crear un perro. Se invoca a los constructores de `Mamífero` y de `Perro`. Esto se repite para `Gato` y para `Mamífero` en las líneas 4 y 7 de la salida.

La línea 9 de la salida representa la llamada a `Hablar()` del primer objeto, el `Perro`. Se llama al método virtual `Hablar()` y se invoca a la versión correcta de `Hablar()`. Luego se llama a la función `Clonar()` y, como también es virtual, se invoca al método `Clonar()` de `Perro`, lo que ocasiona que se llame a los constructores de copia de `Mamífero` y de `Perro`.

En las líneas 12 a 14 se repite lo mismo para `Gato`, y luego para `Mamífero` en las líneas 15 y 16. Por último, se itera el nuevo arreglo, y se invoca al método `Hablar()` de cada uno de los nuevos objetos.

**11**

## El costo de los métodos virtuales

Debido a que los objetos con métodos virtuales deben mantener una tabla v, hay algo de sobrecarga al tener métodos virtuales. Si tiene una clase muy pequeña de la que no espera derivar otras clases, tal vez no exista un motivo para tener métodos virtuales.

Cuando declara a cualquier método como virtual, ya ha pagado la mayor parte del precio de la tabla v (aunque cada entrada agrega una pequeña sobrecarga en la memoria). En ese momento, es mejor que el destructor sea virtual, y debe dar por hecho que todos los demás métodos también son virtuales. Examine cuidadosamente cualquier método que no sea virtual, y asegúrese de entender por qué no es virtual.

| DEBE                                                                              | No DEBE                                            |
|-----------------------------------------------------------------------------------|----------------------------------------------------|
| <b>DEBE</b> utilizar métodos virtuales si espera hacer derivaciones de una clase. | <b>NO DEBE</b> marcar el constructor como virtual. |
| <b>DEBE</b> utilizar un destructor virtual si cualquier método es virtual.        |                                                    |

## Resumen

Hoy aprendió la forma en que las clases derivadas heredan de las clases base. En esta lección hablamos sobre la herencia pública y las funciones virtuales. Las clases heredan todos los datos y funciones públicas y protegidas de sus clases base.

El acceso protegido es público para las clases derivadas y privado para todos los demás objetos. Las clases derivadas no pueden tener acceso a los datos o funciones privadas de sus clases base.

Los constructores se pueden inicializar antes del cuerpo del constructor. En ese momento se invocan los constructores base y se pueden pasar parámetros a la clase base.

Las funciones de la clase base se pueden redefinir en la clase derivada. Si las funciones de la clase base son virtuales, y si se accede al objeto por medio de un apuntador o por referencia, se invocarán las funciones de la clase derivada, con base en el tipo en tiempo de ejecución del objeto al que se apunta.

Los métodos de la clase base se pueden invocar nombrando explícitamente a la función con el prefijo del nombre de la clase base y dos símbolos de dos puntos (::). Por ejemplo, si Perro hereda de Mamífero, se puede llamar al método caminar() de Mamífero con Mamífero::caminar().

En clases que tengan métodos virtuales, el destructor casi siempre debe ser virtual. Un destructor virtual asegura que la parte derivada del objeto se liberará cuando se utilice delete en el apuntador. Los constructores no pueden ser virtuales. Los constructores virtuales de copia se pueden crear al hacer una función miembro virtual que llame al constructor de copia.

## Preguntas y respuestas

- P** ¿Se pasan los miembros y funciones heredados a las generaciones subsecuentes?  
 Si Perro se deriva de Mamífero, y Mamífero se deriva de Animal, ¿hereda Perro las funciones y los datos de Animal?
- R** Sí. Al continuar la derivación, las clases derivadas heredan la suma de todas las funciones y datos que se encuentren en todas sus clases base.

- P Si, en el ejemplo anterior, Mamifero redefine una función de Animal, ¿cuál obtiene Perro, la función original o la redefinida?**
- R Si Perro hereda de Mamifero, recibe la función en el estado en que la tenga Mamifero: la función redefinida.**
- P ¿Puede una clase derivada hacer que la función pública de una clase base sea privada?**
- R Sí, la clase derivada puede redefinir el método y hacerlo privado. Y seguirá siendo privado para cualquier derivación subsecuente.**
- P ¿Por qué no hacer que todas las funciones de una clase sean virtuales?**
- R Ocurre una sobrecarga con la primera función virtual al crear una tabla v. Después de eso, la sobrecarga es trivial. Muchos programadores de C++ sienten que si una función es virtual, todas las demás deben serlo. Otros programadores están en desacuerdo, pues sienten que siempre debe haber un motivo para lo que se va a hacer.**
- P Si una función (UnaFunc()) es virtual en una clase base y también está sobrecargada, de forma que tome ya sea uno o dos enteros, y la clase derivada redefine la que toma un entero, ¿qué se llama cuando un apuntador a un objeto derivado llama a la que toma dos enteros?**
- R La redefinición de la función que toma un int oculta toda la función de la clase base, por lo que se obtendrá un error de compilación que dirá que esa función necesita sólo un int.**

11

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, "Respuestas a los cuestionarios y ejercicios", y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Qué es una tabla v?
2. ¿Qué es un destructor virtual?
3. ¿Cómo se puede mostrar la declaración de un constructor virtual?
4. ¿Cómo se puede crear un constructor virtual de copia?
5. ¿Cómo se invoca a una función miembro de la clase base desde una clase derivada en la que se haya redefinido esa función?
6. ¿Cómo se invoca a una función miembro de la clase base desde una clase derivada en la que no se haya redefinido esa función?

7. Si una clase base declara una función como virtual, y una clase derivada no utiliza el término virtual cuando redefina esa clase, ¿seguirá siendo virtual cuando la herede una clase de tercera generación?
8. ¿Para qué se utiliza la palabra reservada `protected`?

## Ejercicios

1. Muestre la declaración de una función virtual que tome un parámetro entero y regrese `void`.
2. Muestre la declaración de una clase llamada `Cuadrado`, la cual se deriva de `Rectangulo`, que a su vez se deriva de `Figura`.
3. Si, en el ejercicio 2, `Figura` no toma parámetros, `Rectangulo` toma dos (`longitud` y `ancho`), pero `Cuadrado` toma sólo un parámetro (`longitud`), muestre la inicialización del constructor para `Cuadrado`.
4. Escriba un constructor virtual de copia para la clase `Cuadrado` (del ejercicio 3).

5. **CAZA ERRORES:** ¿Qué está mal en este segmento de código?

```
void UnaFuncion (Figura);
Figura * apRect = new Rectangulo;
UnaFuncion(*apRect);
```

6. **CAZA ERRORES:** ¿Qué está mal en este segmento de código?

```
class Figura()
{
public:
    Figura();
    virtual ~Figura();
    virtual Figura(const Figura &);
```

# SEMANA 2

## DÍA 12

### Arreglos, cadenas tipo C y listas enlazadas

En lecciones anteriores se declaraba un `int`, `char`, u otro objeto individual. A menudo es necesario declarar una colección de objetos, por ejemplo, 20 variables de tipo `int` o un conjunto de objetos de la clase `GATO`. Hoy aprenderá lo siguiente:

- Qué son los arreglos y cómo declararlos
- Qué son las cadenas y cómo utilizar arreglos de caracteres para crear cadenas
- La relación existente entre arreglos y apuntadores
- Cómo utilizar aritmética de apuntadores con arreglos

#### Qué es un arreglo

Un *arreglo* es una colección de ubicaciones para guardar datos, cada una de las cuales guarda el mismo tipo de datos. Cada ubicación de almacenamiento es un *elemento* del arreglo.

Un arreglo se declara escribiendo el tipo, seguido del nombre del arreglo y del subíndice. El *subíndice* es el número de elementos del arreglo, y va entre

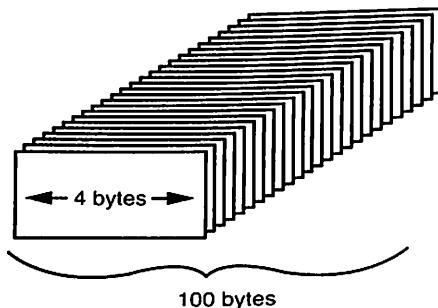
corchetes ([]). Otra palabra para subíndice es *desplazamiento*, ya que representa la distancia desde el principio del arreglo. Por ejemplo:

```
long ArregloLong [25];
```

Este ejemplo declara un arreglo de 25 enteros largos llamado `ArregloLong`. Cuando el compilador ve esta declaración, reserva suficiente memoria para guardar los 25 elementos. Debido a que cada entero largo ocupa 4 bytes, esta declaración separa 100 bytes continuos de memoria, como se muestra en la figura 12.1.

**FIGURA 12.1**

*Declaración de un arreglo.*



## Cómo acceder a los elementos de un arreglo

Puede acceder a cada uno de los elementos del arreglo mediante una referencia al subíndice o desplazamiento que se encuentra entre corchetes en el nombre del arreglo. Los elementos de los arreglos se cuentan desde cero. Por lo tanto, el primer elemento del arreglo es `nombreArreglo[0]`. En el ejemplo del `ArregloLong`, `ArregloLong[0]` es el primer elemento, `ArregloLong[1]` el segundo, y así sucesivamente.

Esto puede ser un poco confuso. El arreglo `UnArreglo[3]` tiene tres elementos. Éstos son `UnArreglo[0]`, `UnArreglo[1]` y `UnArreglo[2]`. Visto en forma más general, `UnArreglo[n]` tiene n elementos numerados desde `UnArreglo[0]` hasta `UnArreglo[n-1]`.

Por lo tanto, `ArregloLong[25]` se numera desde `ArregloLong[0]` hasta `ArregloLong[24]`. El listado 12.1 muestra cómo declarar un arreglo de cinco enteros y llenar cada uno con un valor.

### ENTRADA

### LISTADO 12.1 Uso de un arreglo de enteros

```
1: //Listado 12.1 - Arreglos
2:
3: #include <iostream.h>
4:
5: int main()
6: {
7:     int miArreglo[ 5 ];
8:     int i;
```

```

9:
10:    for (i = 0; i < 5; i++) // 0-4
11:    {
12:        cout << "Valor para miArreglo[" << i << "]: ";
13:        cin >> miArreglo[ i ];
14:    }
15:    for (i = 0; i < 5; i++)
16:        cout << i << ":" << miArreglo[i] << "\n";
17:    return 0;
18: }
```

**SALIDA**

Valor para miArreglo[0]: 3  
 Valor para miArreglo[1]: 6  
 Valor para miArreglo[2]: 9  
 Valor para miArreglo[3]: 12  
 Valor para miArreglo[4]: 15  
 0: 3  
 1: 6  
 2: 9  
 3: 12  
 4: 15

**ANÁLISIS**

La línea 7 declara un arreglo llamado `miArreglo`, el cual guarda cinco variables enteras. La línea 10 establece un ciclo que cuenta desde 0 hasta 4, que es el conjunto apropiado de subíndices o desplazamientos para un arreglo de cinco elementos. Se pide al usuario un valor, y ese valor se guarda en el subíndice correcto del arreglo.

El primer valor se guarda en `miArreglo[0]`, el segundo en `miArreglo[1]`, y así sucesivamente. El segundo ciclo for imprime en la pantalla cada valor.

12

**Nota**

Los arreglos empiezan a contar desde 0, no desde 1. Ésta es la causa de muchos errores en programas de C++ escritos por novatos. Siempre que utilice un arreglo, recuerde que un arreglo con 10 empieza a contar desde `NombreArreglo[0]` hasta `NombreArreglo[9]`. `NombreArreglo[10]` no se utiliza.

## Cómo escribir más allá del fin de un arreglo

Cuando escribe un valor en un elemento de un arreglo, el compilador calcula en dónde se va a guardar el valor con base en el tamaño de cada elemento y del subíndice. Suponga que pide que el valor se escriba en `ArregloLong[5]`, que viene siendo el sexto elemento. El compilador multiplica el desplazamiento (5) por el tamaño de cada elemento (en este caso, 4). Luego mueve todos esos bytes (20) partiendo desde el principio del arreglo y escribe el nuevo valor en esa ubicación.

Recuerde que *subíndice* y *desplazamiento* son sinónimos.

Si pide escribir en `ArregloLong[50]`, el compilador ignora el hecho de que no existe dicho elemento. Calcula qué tan lejos del primer elemento se debe escribir (200 bytes) y luego escribe encima de lo que se encuentre en esa ubicación. Esto puede ser casi cualquier información, y si escribe su nuevo valor ahí, podría tener resultados impredecibles. Si tiene suerte, el programa se detendrá inmediatamente. Si no, eventualmente obtendrá resultados extraños más adelante en el programa, y le será muy difícil descubrir qué salió mal.

En un sistema operativo robusto como Linux, un fallo en el programa no afectará al sistema operativo en sí. En un ambiente que no sea robusto (como MS-DOS), un programa puede sobreescribir porciones del sistema operativo, lo que puede ocasionar que el equipo se congele. Estos problemas son especialmente difíciles de depurar. En casos raros, un programa así podría afectar a Linux, sobre todo si se ejecuta con privilegios de superusuario (root).

El compilador es como un ciego midiendo la distancia entre casas. Empieza en la primera casa, `CallePrincipal[0]`. Si usted le pide que vaya a la sexta casa de la calle principal, él se dice a sí mismo: "Debo avanzar cinco casas más. Cada casa está a cuatro pasos grandes de distancia de la otra. Debo avanzar 20 pasos más". Si le pide que vaya a `CallePrincipal[100]` y la calle principal sólo tiene 25 casas, el medirá 400 pasos. Mucho antes de llegar ahí, sin duda se pondrá enfrente de un autobús en marcha. Así que tenga cuidado hacia dónde lo envía.

El listado 12.2 muestra lo que ocurre cuando se escribe más allá del final de un arreglo.

### Precaución

No ejecute este programa; ¡puede hacer que un sistema que no sea Linux deje de funcionar!

#### ENTRADA LISTADO 12.2 Cómo escribir más allá del final de un arreglo

```
1: //Listado 12.2 Muestra qué pasa cuando se escribe
2: // más allá del fin de un arreglo
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:     // centinelas
9:     long centinelaUno[ 3 ];
10:    long ArregloDestino[ 25 ]; // arreglo que se va a llenar
11:    long centinelaDos[ 3 ];
12:    int i;
13:
14:    for (i = 0; i < 3; i++ )
15:        centinelaUno[ i ] = centinelaDos[ i ] = 0;
```

```
16:     for (i = 0; i < 25; i++ )
17:         ArregloDestino[ i ] = 0;
18:     cout << "Prueba 1: \n"; // probar valores actuales (deben ser 0)
19:     cout << "ArregloDestino[0]: " << ArregloDestino[ 0 ] << "\n";
20:     cout << "ArregloDestino[24]: " << ArregloDestino[ 24 ] << "\n\n";
21:     for (i = 0; i < 3; i++ )
22:     {
23:         cout << "centinelaUno[" << i << "]: ";
24:         cout << centinelaUno[ i ] << "\n";
25:         cout << "centinelaDos[" << i << "]: ";
26:         cout << centinelaDos[ i ]<< "\n";
27:     }
28:     cout << "\nAsignando...";
29:     for (i = 0; i <= 25; i++ )
30:     {
31:         ArregloDestino[ i ] = 20;
32:         cout << "\nPrueba 2: \n";
33:         cout << "ArregloDestino[0]: " << ArregloDestino[ 0 ] << "\n";
34:         cout << "ArregloDestino[24]: " << ArregloDestino[ 24 ] << "\n";
35:         cout << "ArregloDestino[25]: " << ArregloDestino[ 25 ] << "\n\n";
36:         for (i = 0; i < 3; i++ )
37:         {
38:             cout << "centinelaUno[" << i << "]: ";
39:             cout << centinelaUno[ i ]<< "\n";
40:             cout << "centinelaDos[" << i << "]: ";
41:             cout << centinelaDos[ i ]<< "\n";
42:         }
43:     }
44:     return 0;
45: }
```

**SALIDA**

Prueba 1:  
ArregloDestino[0]: 0  
ArregloDestino[24]: 0

centinelaUno[0]: 0  
centinelaDos[0]: 0  
centinelaUno[1]: 0  
centinelaDos[1]: 0  
centinelaUno[2]: 0  
centinelaDos[2]: 0

Asignando...  
Prueba 2:  
ArregloDestino[0]: 20  
ArregloDestino[24]: 20  
ArregloDestino[25]: 20

centinelaUno[0]: 20  
centinelaDos[0]: 0

**12**

```
centinelaUno[1]: 0  
centinelaDos[1]: 0  
centinelaUno[2]: 0  
centinelaDos[2]: 0
```

**ANÁLISIS**

Las líneas 9 y 11 declaran dos arreglos de tres enteros que actúan como centinelas alrededor de `ArregloDestino`. Estos arreglos centinela se inicializan con 0. Si se escribe en memoria que se encuentre más allá del final de `ArregloDestino`, es muy probable que se cambien los centinelas. Algunos compiladores cuentan hacia abajo en la memoria; otros cuentan hacia arriba. Debido a esto, los centinelas se colocan en ambos lados de `ArregloDestino`.

Las líneas 21 a 27 confirman los valores de los centinelas en la prueba 1. En las líneas 29 y 30 se inicializan todos los miembros de `ArregloDestino` con el valor 20, pero el contador cuenta hasta el desplazamiento 25 de `ArregloDestino`, el cual no existe.

Las líneas 32 a 34 imprimen los valores de `ArregloDestino` en la prueba 2. Observe que `ArregloDestino[25]` imprime felizmente un 20. Sin embargo, cuando se imprimen `centinelaUno` y `centinelaDos`, `centinelaUno[0]` revela que su valor ha cambiado. Esto se debe a que la memoria que se encuentra a 25 elementos de distancia de `ArregloDestino[0]` es la misma memoria en la que está `centinelaUno[0]`. Cuando se accedió al `ArregloDestino[25]` no existente, a lo que en realidad se accedió fue a `CentinelaUno[0]`.

Este terrible error puede ser muy difícil de encontrar, debido a que el valor de `centinelaUno[0]` se cambió en una parte del código que no estaba escribiendo en `centinelaUno`.

Este código utiliza “números mágicos” como el 3 para el tamaño de los arreglos centinela y 25 para el tamaño de `ArregloDestino`. Es más seguro utilizar constantes para que se puedan cambiar todos estos valores en un solo lugar.

Tenga en cuenta que como algunos compiladores usan la memoria en forma distinta a otros, sus resultados pueden variar.

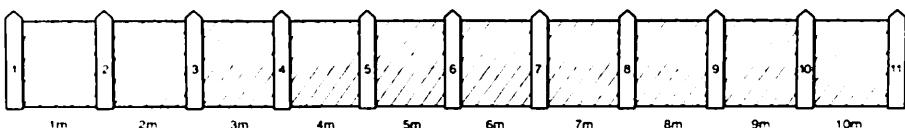
### Errores tipo poste de barda

Debido a que es muy común escribir más allá del final de un arreglo, este error tiene su propio nombre. Se conoce como *error tipo poste de barda*. Esto se refiere al problema producido al contar cuántos postes de barda se necesitan para una barda de 10 metros, si se necesita un poste por cada metro. La mayoría de las personas contesta 10, pero en realidad se necesitan 11. La figura 12.2 ilustra esto.

Este tipo de conteo “menos uno” puede ser la ruina en la vida de cualquier programador. Sin embargo, con el tiempo se acostumbrará a la idea de que un arreglo de 25 elementos llega hasta el elemento 24, y que todo empieza desde 0.

**FIGURA 12.2**

*Errores tipo poste de barda.*



### Nota

Algunos programadores se refieren a `NombreArreglo[0]` como el elemento cero. Tener este hábito es un gran error. Si `NombreArreglo[0]` es el elemento cero, ¿qué es `NombreArreglo[1]`? ¿El primero? Si es así, al ver `NombreArreglo[24]`, ¿se daría cuenta de que no es el 24º elemento, sino el 25º? Es mucho mejor decir que `NombreArreglo[0]` tiene un desplazamiento de cero y es el primer elemento.

Ponga atención a eso: desplazamiento cero y primer elemento (distintos números que describen la misma ubicación). Asegúrese de no intercambiarlos mentalmente.

## Inicialización de arreglos

Puede inicializar un arreglo sencillo de tipos de datos integrados, como enteros y caracteres, al declarar el arreglo. Después del nombre del arreglo, se coloca el signo de igual (=) y una lista de valores separados por comas y encerrados entre llaves. Por ejemplo,

```
int ArregloEntero[5] = { 10, 20, 30, 40, 50 };
```

declara a `ArregloEntero` como un arreglo de cinco enteros. Asigna a `ArregloEntero[0]` el valor 10, a `ArregloEntero[1]` el valor 20, y así sucesivamente.

Si omite el tamaño del arreglo, se crea un arreglo suficientemente grande para guardar los valores de su inicialización. Por lo tanto, si escribe

```
int ArregloEntero[] = { 10, 20, 30, 40, 50 };
```

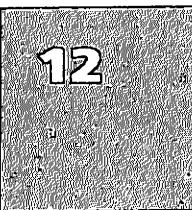
creará el mismo arreglo que el del ejemplo anterior.

Si necesita conocer el tamaño del arreglo, puede pedir al compilador que lo calcule por usted. Por ejemplo,

```
const USHORT LongitudArregloEntero;
LongitudArregloEntero = sizeof(ArregloEntero)/sizeof(ArregloEntero[0]);
```

le asigna a la variable constante de tipo `USHORT`, llamada `LongitudArregloEntero`, el resultado obtenido al dividir el tamaño de todo el arreglo entre el tamaño de cada entrada individual del arreglo. Ese cociente es el número de miembros del arreglo.

No puede inicializar más elementos de los que haya declarado para el arreglo. Por lo tanto,



```
int ArregloEntero[5] = { 10, 20, 30, 40, 50, 60};
```

genera un error de compilación, debido a que declaró un arreglo con cinco elementos e inicializó seis valores. Sin embargo, es válido escribir

```
int ArregloEntero[5] = { 10, 20};
```

Aunque los miembros del arreglo que no se inicialicen no tienen valores garantizados, los agregados se inicializarán con 0. Por lo tanto, si no inicializa un elemento de un arreglo, su valor se establecerá en 0.

### DEBE

DEBE permitir que el compilador establezca el tamaño de los arreglos inicializados.

DEBE dar nombres significativos a los arreglos, como lo haría con cualquier variable.

DEBE recordar que el primer miembro del arreglo se encuentra en el desplazamiento 0.

### NO DEBE

NO DEBE escribir más allá del final del arreglo.

## Declaración de arreglos

Los arreglos pueden tener cualquier nombre de variable válido, pero no pueden tener el mismo nombre que otra variable o arreglo dentro de su mismo alcance. Por lo tanto, usted no puede tener un arreglo llamado `misGatos[5]` y una variable llamada `misGatos` al mismo tiempo.

Puede dimensionar el tamaño del arreglo con un `const` o con una enumeración. El listado 12.3 muestra esto.

### ENTRADA

### LISTADO 12.3 Uso de consts y enums en arreglos

```

1: // Listado 12.3
2: // Cómo dimensionar arreglos con constantes y enumeraciones
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:     enum SemanaDias { Dom, Lun, Mar, Mie, Jue,
9:                       Vie, Sab, DiasDeLaSemana };
10:
11:    int ArregloSemana[ DiasDeLaSemana ] = { 10, 20, 30, 40, 50, 60, 70 };
12:
13:    cout << "El valor de Jueves es: " << ArregloSemana[ Jue ] << endl;
14:
15: }
```

**SALIDA** El valor de Jueves es: 50**ANÁLISIS** La línea 8 crea una enumeración llamada SemanaDias. Tiene ocho elementos. Domingo es igual a 0, y DiasDeLaSemana es igual a 7.

La línea 13 utiliza la constante enumerada Jue como un desplazamiento dentro del arreglo. Como Jue se evalúa en 4, se regresa el quinto elemento del arreglo, ArregloSemana[4], y se imprime en la línea 13.

### Arreglos

Para declarar un arreglo, escriba el tipo del objeto guardado, seguido del nombre del arreglo y un subíndice que indica el número de objetos que se van a guardar en el arreglo.

Ejemplo 1

```
int MiArregloEntero[90];
```

Ejemplo 2

```
long * ArregloDeApunadoresALongs[100];
```

Para tener acceso a los miembros del arreglo, utilice el operador de subíndice.

Ejemplo 1

```
int elNovenoEntero = MiArregloDeEnteros[8];
```

Ejemplo 2

```
long * apLong = ArregloDeApunadoresALongs[8];
```

Los arreglos empiezan a contar desde cero. Un arreglo de  $n$  elementos se numera desde 0 hasta  $n-1$ .

12

## Arreglos de objetos

Cualquier objeto, ya sea integrado o definido por el usuario, se puede guardar en un arreglo. Cuando declara el arreglo, le indica al compilador el tipo de objeto a guardar y el número de objetos a los que debe asignar espacio. El compilador sabe cuánto espacio se necesita para cada objeto, con base en la declaración de la clase. La clase debe tener un constructor predeterminado que no lleve argumentos, para que se puedan crear los objetos cuando se defina el arreglo.

El acceso a los datos miembro de un arreglo de objetos es un proceso de dos pasos. Se identifica el miembro del arreglo utilizando el operador de índice ([ ]). y luego se agrega el operador de punto (.) para tener acceso a la variable miembro específica. El listado 12.4 muestra cómo se crea un arreglo de cinco objetos de tipo GATO.

**ENTRADA****LISTADO 12.4 Creación de un arreglo de objetos**

```

1: // Listado 12.4 - Un arreglo de objetos
2:
3: #include <iostream.h>
4:
5: class GATO
6: {
7:     public:
8:         GATO()
9:             { suEdad = 1; suPeso = 5; }
10:        ~GATO() {}
11:        int ObtenerEdad() const
12:            { return suEdad; }
13:        int ObtenerPeso() const
14:            { return suPeso; }
15:        void AsignarEdad(int edad )
16:            { suEdad = edad; }
17:     private:
18:         int suEdad;
19:         int suPeso;
20:     };
21:
22: int main()
23: {
24:     GATO Camada[ 5 ];
25:     int i;
26:
27:     for (i = 0; i < 5; i++ )
28:         Camada[ i ].AsignarEdad(2 * i + 1 );
29:     for (i = 0; i < 5; i++ )
30:     {
31:         cout << "Gato #" << i + 1 << ":" ;
32:         cout << Camada[ i ].ObtenerEdad() << endl;
33:     }
34:     return 0;
35: }
```

**SALIDA**

Gato #1: 1  
 Gato #2: 3  
 Gato #3: 5  
 Gato #4: 7  
 Gato #5: 9

**ANÁLISIS**

Las líneas 5 a 20 declaran la clase GATO. Esta clase debe tener un constructor predeterminado para que se puedan crear objetos de tipo GATO en un arreglo.

Recuerde que si crea cualquier otro constructor, el compilador no proporcionará el constructor predeterminado.

El primer ciclo for (líneas 27 y 28) establece la edad de cada uno de los cinco objetos de tipo GATO del arreglo. El segundo ciclo for (líneas 29 a 33) accede a cada miembro del arreglo y llama al método ObtenerEdad().

Para llamar al método `ObtenerEdad()` de cada GATO individual, se accede al elemento del arreglo, es decir `Camada[i]`, seguido del operador de punto `(.)` y de la función miembro.

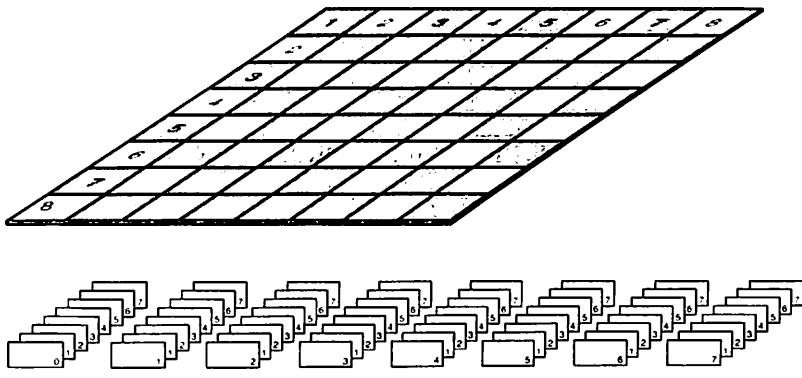
## Trabajo con arreglos multidimensionales

Es posible tener arreglos con más de una dimensión. Cada dimensión se representa como un subíndice del arreglo. Por lo tanto, un arreglo de dos dimensiones tiene dos subíndices; un arreglo de tres dimensiones tiene tres subíndices; y así sucesivamente. Los arreglos pueden tener cualquier número de dimensiones, aunque es muy probable que la mayoría de los arreglos que usted vaya a crear sea de una o dos dimensiones.

Un buen ejemplo de un arreglo de dos dimensiones es un tablero de ajedrez. Una dimensión representa las ocho filas; la otra dimensión representa las ocho columnas. La figura 12.3 ilustra esta idea.

**FIGURA 12.3**

*Un tablero de ajedrez y un arreglo de dos dimensiones.*



Imagine que tiene una clase llamada CUADRO. La declaración de un arreglo llamado Tablero que la represente sería

```
CUADRO Arreglo[8][8];
```

También podría representar los mismos datos con un arreglo de una dimensión de 64 cuadros. Por ejemplo:

```
CUADRO Tablero[64];
```

Esto no representa tan fielmente al objeto real como el arreglo de dos dimensiones. Cuando el juego empieza, el rey está ubicado en la cuarta posición de la primera fila. Si se cuenta desde cero en el arreglo, esta posición corresponde a

```
Arreglo[0][3];
```

dando por hecho que el primer subíndice corresponde a **fila** y el segundo a **columna**. La distribución de las posiciones para todo el tablero se muestra en la figura 12.3.

## Inicialización de arreglos multidimensionales

Los arreglos multidimensionales también se pueden inicializar. Se asigna la lista de valores para los elementos del arreglo en orden, y el último subíndice del arreglo cambia mientras que los otros subíndices permanecen fijos. Por lo tanto, si tiene un arreglo

```
int elArreglo[5][3];
```

los tres primeros elementos van en `elArreglo[0]`; los siguientes tres en `elArreglo[1]`; y así sucesivamente.

Este arreglo se inicializa escribiendo lo siguiente:

```
int elArreglo[5][3] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 };
```

Para que todo sea más claro, las inicializaciones se pueden agrupar con llaves. Por ejemplo:

```
int elArreglo[5][3] = { {1,2,3},  
{4,5,6},  
{7,8,9},  
{10,11,12},  
{13,14,15} };
```

El compilador ignora las llaves de adentro, lo que facilita entender la manera en que se distribuyen los números.

Cada valor debe estar separado por una coma, sin importar las llaves. Todo el conjunto de inicialización debe estar encerrado entre llaves, y debe terminar con punto y coma.

El listado 12.5 crea un arreglo de dos dimensiones. La primera dimensión es el conjunto de números del 0 al 4. La segunda dimensión consiste en el doble de cada valor que se encuentra en la primera dimensión.

### ENTRADA LISTADO 12.5 Creación de un arreglo multidimensional

---

```
1: // Listado 12.5 Muestra el uso de arreglos  
2: // multidimensionales  
3:  
4: #include <iostream.h>  
5:  
6: int main()  
7: {  
8:     int UnArreglo[ 5 ][ 2 ] = { {0, 0}, {1, 2}, {2, 4}, {3, 6}, {4, 8} };  
9:  
10:    for (int i = 0; i < 5; i++ )  
11:        for (int j = 0; j < 2; j++ )  
12:        {  
13:            cout << "UnArreglo[" << i << "][" << j << "]: " ;  
14:            cout << UnArreglo[ i ][ j ]<< endl;  
15:        }  
16:    return 0;  
17: }
```

---

**SALIDA**

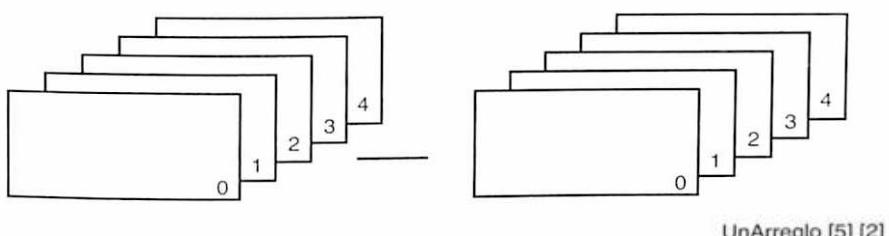
```
UnArreglo[0][0]: 0
UnArreglo[0][1]: 0
UnArreglo[1][0]: 1
UnArreglo[1][1]: 2
UnArreglo[2][0]: 2
UnArreglo[2][1]: 4
UnArreglo[3][0]: 3
UnArreglo[3][1]: 6
UnArreglo[4][0]: 4
UnArreglo[4][1]: 8
```

**ANÁLISIS**

La línea 8 declara un arreglo de dos dimensiones, `UnArreglo`. La primera dimensión consta de cinco enteros; la segunda dimensión consta de dos enteros. Esto crea una cuadrícula de  $5 \times 2$ , como se ve en la figura 12.4.

**FIGURA 12.4**

*Un arreglo de  $5 \times 2$ .*



Los valores se inicializan en pares, aunque también se pueden calcular. Las líneas 10 y 11 crean un ciclo `for` anidado. El ciclo `for` externo avanza por cada elemento de la primera dimensión. Para cada elemento de esa dimensión, el ciclo `for` interno avanza por cada elemento de la segunda dimensión. Esto concuerda con la salida. `UnArreglo[0][1]` sigue a `UnArreglo[0][0]`. La primera dimensión se incrementa sólo después que se incrementa en 1 la segunda dimensión. Luego la segunda dimensión vuelve a empezar.

**12**

## Una palabra sobre los arreglos y la memoria

Cuando declara un arreglo, le está indicando al compilador exactamente cuántos objetos espera guardar en él. El compilador reserva memoria para todos los objetos, incluso si no los utiliza. Éste no es un problema para arreglos en los que tenga una buena idea de cuántos objetos necesitará. Por ejemplo, un tablero de ajedrez tiene 64 cuadros, y los gatos tienen entre 1 y 10 gatitos. Sin embargo, cuando no tiene idea de cuántos objetos va a necesitar, debe usar estructuras de datos más avanzadas.

Este libro trata arreglos de apuntadores, arreglos creados en el espacio libre de almacenamiento (o heap), y otras colecciones más. Verá unas cuantas estructuras de datos avanzadas, y puede aprender más en libros avanzados, como el libro *C++: Cómo Programar*, de Prentice Hall. Dos de las grandes ventajas de la programación son que siempre hay más cosas que aprender, y siempre hay más libros de los que se puede aprender.

## Uso del heap para solucionar problemas relacionados con la memoria

Los arreglos de los que hemos hablado hasta ahora guardan todos sus elementos en la pila. Por lo general, la memoria de la pila está severamente limitada, mientras que la memoria del heap es mucho más grande. El tamaño de los arreglos es de naturaleza estática. Una vez que declara un arreglo y compila el programa, no puede cambiar el tamaño de los arreglos durante la ejecución. Pero existen formas de vencer esta limitación. El aprovechamiento del heap le permite tener estructuras de datos que se comporten como si fueran arreglos dinámicos.

### Arreglos de apuntadores

Es posible declarar cada objeto en el heap y luego guardar sólo un apuntador al objeto del arreglo. Esto reduce considerablemente la cantidad de memoria utilizada de la pila. El listado 12.6 vuelve a utilizar el arreglo del listado 12.4, pero guarda todos los objetos en el heap. Como indicación de la mayor cantidad de memoria que esto permite, el arreglo se expande de 5 a 500, y el nombre se cambia de *Camada* a *Familia*.

**ENTRADA****LISTADO 12.6** Cómo guardar un arreglo en el heap

```
1: // Listado 12.6 - Un arreglo de apuntadores a objetos
2:
3: #include <iostream.h>
4:
5: class GATO
6: {
7: public:
8:     GATO()
9:     { suEdad = 1; suPeso = 5; }
10:    ~GATO() {} // destructor
11:    int ObtenerEdad() const
12:    { return suEdad; }
13:    int ObtenerPeso() const
14:    { return suPeso; }
15:    void AsignarEdad(int edad )
16:    { suEdad = edad; }
17: private:
18:     int suEdad;
19:     int suPeso;
20: };
21:
22: int main()
23: {
24:     GATO * Familia[ 500 ];
25:     int i;
26:     GATO * apGato;
27:
28:     for ( i = 0; i < 500; i++ )
29:     {
```

```

30:         apGato = new GATO;
31:         apGato->AsignarEdad(2 * i + 1 );
32:         Familia[ i ] = apGato;
33:     }
34:     for ( i = 0; i < 500; i++ )
35:     {
36:         cout << "Gato #" << i + 1 << ":" ;
37:         cout << Familia[ i ]->ObtenerEdad() << endl;
38:     }
39:     return 0;
40: }

```

**SALIDA**

Gato #1: 1  
 Gato #2: 3  
 Gato #3: 5  
 ...  
 Gato #499: 997  
 Gato #500: 999

**ANÁLISIS**

El objeto GATO declarado en las líneas 5 a 20 es idéntico al objeto GATO declarado en el listado 12.4. Sin embargo, esta vez el arreglo declarado en la línea 24 se llama **Familia**, y se declara para guardar 500 apuntadores a objetos GATO.

En el ciclo inicial (líneas 28 a 33), se crean 500 nuevos objetos GATO en el heap, y se establece la edad de cada uno al doble del índice mas uno. Por lo tanto, al primer GATO se le asigna un 1, al segundo GATO un 3, al tercer GATO un 5, y así sucesivamente. Por último, se agrega el apuntador al arreglo.

Debido a que el arreglo ha sido declarado para guardar apuntadores, el apuntador (en lugar del valor desreferenciado contenido en él) se agrega al arreglo.

El segundo ciclo (líneas 34 a 38) imprime cada uno de los valores. El apuntador se accede por medio del índice, **Familia[i]**. Esa dirección se utiliza entonces para tener acceso al método **ObtenerEdad()**.

En este ejemplo, el arreglo **Familia** y todos sus apuntadores se guardan en la pila, pero los 500 objetos de tipo GATO que se crean se guardan en el heap.

**12**

## Declaración de arreglos en el heap

Es posible colocar todo el arreglo en el heap. Esto se hace llamando a **new** y utilizando el operador de subíndice. El resultado es un apuntador a un área del heap que guarda el arreglo. Por ejemplo,

```
GATO * Familia = new GATO[ 500 ];
```

declara a **Familia** como apuntador al primer elemento de un arreglo de 500 objetos de tipo GATO. En otras palabras, **Familia** apunta a (o tiene la dirección de) **Familia[0]**.

La ventaja de usar **Familia** de esta manera es que puede utilizar aritmética de apuntadores para tener acceso a cada miembro de **Familia**. Por ejemplo, puede escribir

```
GATO * Familia = new GATO[ 500 ];
GATO * apGato = Familia;           //apGato apunta a Familia[0]
apGato->AsignarEdad(10 );        // asignar un 10 a Familia[0]
apGato++;
apGato-> AsignarEdad(20 );       // avanzar a Familia[1]
                                    // asignar un 20 a Familia[1]
```

Esto declara un nuevo arreglo de 500 objetos de tipo GATO y un apuntador que apunta al inicio del arreglo. Al utilizar ese apuntador, se llama a la función **AsignarEdad()**, con el valor 10, del primer GATO . Luego se incrementa el apuntador para que apunte al siguiente GATO, y se llama al método **AsignarEdad()** del segundo GATO.

## **Uso de un apuntador a un arreglo en comparación con un arreglo de apuntadores**

Examine las tres declaraciones siguientes:

```
1: GATO  FamiliaUno[ 500 ];
2: GATO * FamiliaDos[ 500 ];
3: GATO * FamiliaTres = new GATO[ 500 ];
```

**FamiliaUno** es un arreglo de 500 objetos de tipo GATO. **FamiliaDos** es un arreglo de 500 apuntadores a objetos de tipo GATO. **FamiliaTres** es un apuntador a un arreglo de 500 objetos de tipo GATO.

Las diferencias entre estas tres líneas de código afectan considerablemente la forma en que funcionan estos arreglos . Lo que es aún más sorprendente es que **FamiliaTres** es una variante de **FamiliaUno**, pero es muy diferente de **FamiliaDos**.

Esto trae a la mente la espinosa cuestión de la manera en que se relacionan los apuntadores con los arreglos. En el tercer caso, **FamiliaTres** es un apuntador a un arreglo. Es decir, la dirección contenida en **FamiliaTres** es la dirección del primer elemento del arreglo. Éste es el mismo caso para **FamiliaUno**, con la diferencia de que **FamiliaUno** se crea en la pila y **FamiliaTres** se crea en el heap.

## **Uso de apuntadores con nombres de arreglos**

En C++, el nombre de un arreglo es un apuntador constante al primer elemento del arreglo. Por lo tanto, en la declaración

```
GATO Familia[ 50 ];
```

**Familia** es un apuntador a **&Familia[ 0 ]**, que es la dirección del primer elemento del arreglo **Familia**.

Es válido utilizar nombres de arreglos como apuntadores constantes, y viceversa. Por lo tanto, **Familia + 4** es una forma válida de tener acceso a los datos de **Familia[ 4 ]**.

El compilador se encarga de toda la aritmética cuando se suman, incrementan y decrementan apuntadores. La dirección que se accede al escribir `Familia + 4` no está 4 bytes más allá de la dirección de `Familia`, está 4 objetos más allá. Si cada objeto tiene 4 bytes de largo, `Familia + 4` está 16 bytes más allá del inicio del arreglo. Si cada objeto es un `GATO` que tiene 4 variables miembro de tipo `long` de 4 bytes cada una, y dos variables miembro de tipo `short` de 2 bytes cada una, esto quiere decir que cada `GATO` es de 20 bytes, y `Familia + 4` está 80 bytes más allá del inicio del arreglo.

El listado 12.7 muestra la declaración y el uso de un arreglo en el heap.

**ENTRADA** **LISTADO 12.7** Creación de un arreglo por medio de new

```
1: // Listado 12.7 - Un arreglo en el heap
2:
3: #include <iostream.h>
4:
5: class GATO
6: {
7: public:
8:     GATO()
9:     { suEdad = 1; suPeso = 5; }
10:    ~GATO();
11:    int ObtenerEdad() const
12:    { return suEdad; }
13:    int ObtenerPeso() const
14:    { return suPeso; }
15:    void AsignarEdad(int edad)
16:    { suEdad = edad; }
17: private:
18:     int suEdad;
19:     int suPeso;
20: };
21:
22: GATO::~GATO()
23: {
24:     // cout << "¡Se llamó al destructor!\n";
25: }
26:
27: int main()
28: {
29:     GATO * Familia = new GATO[ 500 ];
30:     int i;
31:
32:     for (i = 0; i < 500; i++ )
33:     {
34:         Familia[ i ].AsignarEdad(2 * i + 1 );
35:     }
36:     for (i = 0; i < 500; i++ )
```

12

**LISTADO 12.7** CONTINUACIÓN

```

37:      {
38:          cout << "Gato #" << i + 1 << ": ";
39:          cout << Familia[ i ].ObtenerEdad() << endl;
40:      }
41:      delete [] Familia;
42:      return 0;
43:  }

```

**SALIDA**

```

Gato #1: 1
Gato #2: 3
Gato #3: 5
...
Gato #499: 997
Gato #500: 999

```

**ANÁLISIS**

La línea 29 declara el arreglo `Familia`, el cual guarda 500 objetos tipo `GATO`. Todo el arreglo se crea en el heap con la llamada a `new GATO[500]`.

## Eliminación de arreglos en el heap

¿Qué ocurre con la memoria asignada para estos objetos de tipo `GATO` cuando se destruye el arreglo? ¿Existe una probabilidad de fuga de memoria? Al eliminar `Familia` se regresa automáticamente toda la memoria reservada para el arreglo si se utiliza el operador `delete [ ]`, recordando siempre incluir los corchetes. El compilador tiene suficiente inteligencia para destruir cada objeto del arreglo y luego liberar la memoria que éste ocupaba en el heap.

Para ver eso, cambie el tamaño del arreglo de 500 a 10 en las líneas 29, 32 y 36. Luego modifique la instrucción `cout` de la línea 24 para que se pueda ejecutar y deje de ser comentario. Cuando se llegue a la línea 41 y se destruya el arreglo, se llamará al destructor de cada objeto `GATO`.

Cuando cree un elemento en el heap por medio de `new`, siempre debe eliminar ese elemento y liberar su memoria con `delete`. De la misma manera, cuando cree un arreglo por medio de `new <clase>[tamaño]`, debe eliminar ese arreglo y liberar toda su memoria con `delete[]`. Los corchetes le indican al compilador que se va a eliminar un arreglo.

Si omite los corchetes, sólo se eliminará el primer objeto del arreglo. Puede probar esto usted mismo si quita los corchetes de la línea 41. Si modificara la línea 24 para que el destructor imprimiera, entonces vería sólo un objeto `GATO` destruido. ¡Felicitaciones! Acaba de crear una fuga de memoria.

| DEBE                                                                                    | NO DEBE                                                                           |
|-----------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| <b>DEBE</b> recordar que un arreglo de $n$ elementos se numera desde cero hasta $n-1$ . | <b>NO DEBE</b> escribir o leer más allá del final de un arreglo.                  |
| <b>DEBE</b> utilizar indexación de arreglos en apuntadores que apunten a arreglos.      | <b>NO DEBE</b> confundir un arreglo de apuntadores con un apuntador a un arreglo. |

## Qué son los arreglos de tipo char

Una cadena es una serie de caracteres. Las únicas cadenas que ha visto hasta ahora son cadenas constantes sin nombre utilizadas en instrucciones cout, como

```
cout << "¡Hola, mundo!\n";
```

En C++, una cadena es un arreglo de valores de tipo char que termina con un carácter nulo. Puede declarar e inicializar una cadena de la misma forma en que lo haría con cualquier otro arreglo. Por ejemplo:

```
char Saludo[] = { 'i', 'H', 'o', 'l', 'a', ',', ' ', 'm', 'u', 'n', 'd', 'o', '!', '\0' };
```

El último carácter, '\0', es el carácter nulo que muchas funciones de C++ reconocen como terminador para una cadena. Aunque este método de carácter por carácter funciona, es difícil de escribir y al utilizarlo se pueden cometer errores. C++ le permite utilizar una forma abreviada para escribir la anterior línea de código. Ésta es

```
char Saludo[] = "¡Hola, mundo!";
```

Debe tener en cuenta dos cosas acerca de esta sintaxis:

- En vez de tener caracteres entre comillas simples separados por comas y encerrados entre llaves, tiene una cadena con doble comilla, sin comas y sin llaves.
- No necesita agregar el carácter nulo, ya que el compilador lo agrega por usted.

La cadena ¡Hola, mundo! es de 14 bytes. (!) ocupa 1 byte, Hola ocupa 4 bytes, la coma 1 byte, el espacio 1, mundo 5, (!) ocupa 1 y el carácter nulo 1.

También puede crear arreglos de caracteres que no estén inicializados. Al igual que con todos los arreglos, es importante asegurarse de no sobrepasar el espacio disponible en el búfer.

El listado 12.8 muestra el uso de un búfer sin inicializar.

**ENTRADA** **LISTADO 12.8** Cómo llenar un arreglo

```

1: //Listado 12.8 búferes de arreglos de tipo char
2:
3: #include <iostream.h>
4:
5: int main()
6: {
7:     char bufer[ 80 ];
8:
9:     cout << "Escriba la cadena: ";
10:    cin >> bufer;
11:    cout << "Aquí está el búfer: " << bufer << endl;
12:    return 0;
13: }

```

**SALIDA**      Escriba la cadena: ¡Hola, mundo!  
                   Aquí está el búfer: ¡Hola, mundo!

**ANÁLISIS** En la línea 7 se declara un búfer para guardar 80 caracteres. Este búfer es lo suficientemente grande para guardar una cadena de 79 caracteres y un carácter nulo terminador.

En la línea 9 se pide al usuario que escriba una cadena, la cual se escribe en el búfer en la línea 10. La sintaxis de `cin` es escribir un terminador nulo en el búfer después de escribir la cadena.

Hay dos problemas con el programa del listado 12.8. En primer lugar, si el usuario escribe más de 79 caracteres, `cin` escribe más allá del final del búfer. En segundo lugar, si el usuario escribe un espacio, `cin` piensa que es el final de la cadena y deja de escribir en el búfer.

Para resolver estos problemas, debe llamar a un método especial en `cin`: `get()`. `cin.get()` toma tres parámetros:

- El búfer que se va a llenar
- El número máximo de caracteres que se van a obtener
- El delimitador que termina la entrada

El delimitador predeterminado es `newline`. El listado 12.9 muestra su uso.

**ENTRADA** **LISTADO 12.9** Cómo llenar un arreglo

```

1: //Listado 12.9 uso de cin.get()
2:
3: #include <iostream.h>
4:
5: int main()
6: {
7:     char bufer[ 80 ];
8:

```

```

9:     cout << "Escriba la cadena: ";
10:    cin.get(bufer, 79 ); // llegar hasta 79 o a newline
11:    cout << "Aqui está el búfer: " << bufer << endl;
12:    return 0;
13: }

```

**SALIDA**      Escriba la cadena: ¡Hola, mundo!  
Aqui está el búfer: ¡Hola, mundo!

**ANÁLISIS** La línea 10 llama al método `get()` de `cin`. El búfer que se declara en la línea 7 se pasa como el primer argumento. El segundo argumento es el número máximo de caracteres a obtener. En este caso, debe ser 79 para dejar un espacio para el terminador nulo. No hay necesidad de proporcionar un carácter de terminación, ya que el valor pre-determinado de `newline` es suficiente.

## Uso de funciones para cadenas

C++ hereda de C una biblioteca de funciones para manejo de cadenas. De todas esas funciones, las más utilizadas son: `strcpy()`, `strncpy()`, `strcat()` y `strlen()`. `strcpy()` copia en un búfer designado todo el contenido de una cadena; `strncpy()` copia el número especificado de caracteres de una cadena a otra; `strlen()` nos indica el tamaño de la cadena (que puede ser distinto del tamaño del arreglo); y `strcat()` concatena dos cadenas. El listado 12.10 muestra el uso de estas funciones.

**ENTRADA** **LISTADO 12.10** Uso de funciones comunes para cadenas

```

1: // listado 12.10 - Uso de strcpy, strncpy, strlen y strcat
2:
3: #include <iostream.h>
4: #include <string.h>
5:
6: int main()
7: {
8:     char Cadena1[] = "¡Así, siempre así he de verte!";
9:     char Cadena2[ 80 ] = "";
10:    char Cadena3[ 80 ] = "";
11:
12:    cout << "Cadena1: " << Cadena1 << endl;
13:    strcpy(Cadena2, Cadena1);
14:    cout << "Cadena2: " << Cadena2 << endl;
15:    strncpy (Cadena3, Cadena1, 5); // no es toda la cadena
16:    Cadena3[ 5 ] = '\0'; // necesita un terminador nulo
17:    cout << "Cadena3 después de strncpy: ";
18:    cout << Cadena3 << endl;
19:    cout << "Cadena1 mide " << strlen (Cadena1)
20:        << " bytes de largo, \n";
21:    cout << "Cadena2 mide " << strlen (Cadena2)
22:        << " bytes de largo, \n";
23:    cout << "y Cadena3 mide " << strlen (Cadena3)

```

12

*continua*

**LISTADO 12.10** CONTINUACIÓN

---

```

24:           << " bytes de largo" << endl;
25:   strcat(Cadena3, Cadena1);
26:   cout << "Cadena3 después de strcat: " << Cadena3 << endl;
27:   cout << "Cadena1 aún mide " << strlen (Cadena1)
28:           << " bytes de largo, \n";
29:   cout << "Cadena2 aún mide " << strlen (Cadena2)
30:           << " bytes de largo, \n";
31:   cout << "y Cadena3 ahora mide " << strlen (Cadena3)
32:           << " bytes de largo" << endl;
33:   return 0;
34: }
```

---

**SALIDA**

Cadena1: ¡Así, siempre así he de verte!  
 Cadena2: ¡Así, siempre así he de verte!  
 Cadena3 después de strncpy: ¡Así,  
 Cadena1 mide 30 bytes de largo,  
 Cadena2 mide 30 bytes de largo,  
 y Cadena3 mide 5 bytes de largo  
 Cadena3 después de strcat: ¡Así,¡Así, siempre así he de verte!  
 Cadena1 aún mide 30 bytes de largo,  
 Cadena2 aún mide 30 bytes de largo,  
 y Cadena3 ahora mide 35 bytes de largo

**ANÁLISIS**

En la línea 4 se incluye el archivo de encabezado `string.h`. Este archivo contiene los prototipos de las funciones para cadenas. Las líneas 8, 9 y 10 asignan tres arreglos de caracteres; el primero se inicializa con “¡Así, siempre así he de verte！”, y los otros dos están vacíos.

En la línea 13, la función `strcpy()` toma dos arreglos de caracteres (el de destino seguido del de origen). Si el origen fuera más largo que el destino, `strcpy()` sobreescribiría más allá del final del búfer.

Para protegerse contra esto, la biblioteca estándar también incluye a `strncpy()` como se muestra en la línea 15. Esta variación toma un número máximo de caracteres a copiar. `strncpy()` copia hasta el primer carácter nulo del número máximo de caracteres especificados en el búfer de destino. Si este máximo es menor que la longitud de la cadena, no se incluye un terminador nulo en la cadena de destino. Así que usted tiene que poner uno, como se muestra en la línea 16.

`strlen()` toma un arreglo de caracteres como argumento y regresa la longitud del arreglo hasta, pero sin incluir, el terminador nulo. El valor que regresa tendrá como límite el tamaño del arreglo que contiene la cadena, menos uno.

En la línea 25, la función `strcat()` toma dos arreglos de caracteres (el de destino seguido del de origen). El arreglo de origen será concatenado al final de la cadena que se guarda en el arreglo de destino. Si el tamaño de las cadenas que se van a concatenar excede el tamaño de la cadena de destino, `strcat()` sobreescribiría más allá del final del búfer (para prevenir esto, se puede usar `strncat()`).

Desde luego que hay disponibles muchas otras funciones para cadenas.

## Uso de cadenas y apuntadores

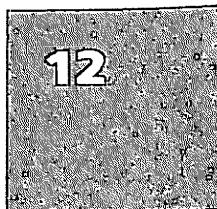
Como aprendió hoy, cuando utiliza el nombre del arreglo solo (sin un subíndice encerrado entre corchetes), está haciendo referencia a la dirección del arreglo. Esta variable se comporta como un apuntador. Es un apuntador especial (un apuntador constante). Es constante porque usted no puede cambiar la dirección a la que apunta. La razón de esto es la siguiente: los arreglos no se mueven; si cambiara el apuntador, no podría hacer referencia a la memoria asignada para ese arreglo.

Observe que, en el listado 12.10, el nombre del arreglo se utiliza como argumento en las funciones para cadenas. También puede utilizar apuntadores a cadenas en lugar de arreglos. La única diferencia es que no puede cambiar una cadena ahí mismo (no puede concatenarle otra cadena, como lo hizo con el arreglo Cadena3). El estándar del lenguaje no prohíbe esto, pero el compilador o el sistema operativo tal vez no lo permitan (tratan a las cadenas como si fueran de sólo lectura). Aunque no se prohíbe hacer esto, no debe cambiar una cadena así. Confíe en mí con respecto a esto (puede ocasionar todo tipo de efectos secundarios, y no es portable).

El listado 12.11 muestra el uso de cadenas y apuntadores.

### ENTRADA LISTADO 12.11 Uso de cadenas y apuntadores

```
1: // listado 12.11 - Cadenas y apuntadores
2:
3: #include <iostream.h>
4: #include <string.h>
5:
6: int main()
7: {
8:     char * Cadena1 = "¡Así, siempre así he de verte!";
9:     char Cadena2[ 80 ] = "";
10:    char Cadena3[ 80 ] = "";
11:
12:    cout << "Cadena1: " << Cadena1 << endl;
13:    strcpy(Cadena2, Cadena1 );
14:    cout << "Cadena2: " << Cadena2 << endl;
15:    strncpy(Cadena3, Cadena1, 5 ); // no es toda la cadena
16:    Cadena3[ 5 ] = '\0'; // necesita un terminador nulo
17:    cout << "Cadena3 después de strncpy: ";
18:    cout << Cadena3 << endl;
19:    cout << "Cadena1 mide " << strlen (Cadena1 )
20:        << " bytes de largo, \n";
21:    cout << "Cadena2 mide " << strlen (Cadena2 )
22:        << " bytes de largo, \n";
23:    cout << "y Cadena3 mide " << strlen (Cadena3 )
24:        << " bytes de largo" << endl;
25:    strcat(Cadena3, Cadena1 );
26:    cout << "Cadena3 después de strcat: " << Cadena3 << endl;
27:    cout << "Cadena1 aún mide " << strlen (Cadena1 )
```



**LISTADO 12.11** CONTINUACIÓN

```

28:           << " bytes de largo, \n";
29:   cout << "Cadena2 aún mide " << strlen (Cadena2 )
30:           << " bytes de largo, \n";
31:   cout << "y Cadena3 ahora mide " << strlen (Cadena3 )
32:           << " bytes de largo" << endl;
33:   Cadena1 = "Allí estás hoy, junto a la tienda de Ayax...";
34:   cout << "Cadena1: " << Cadena1 << endl;
35:   strcat(Cadena3, Cadena1 );
36:   cout << "Cadena3 después de strcat2: " << Cadena3 << endl;
37:   cout << "Cadena1 mide ahora " << strlen (Cadena1 )
38:           << " bytes de largo, \n";
39:   cout << "Cadena2 aún mide " << strlen (Cadena2 )
40:           << " bytes de largo, \n";
41:   cout << "y Cadena3 mide ahora " << strlen (Cadena3 )
42:           << " bytes de largo" << endl;
43:   return 0;
44: }
```

**SALIDA**

```

Cadena1: ¡Así, siempre así he de verte!
Cadena2: ¡Así, siempre así he de verte!
Cadena3 después de strncpy: ¡Así,
Cadena1 mide 30 bytes de largo,
Cadena2 mide 30 bytes de largo,
y Cadena3 mide 5 bytes de largo
Cadena3 después de strcat: ¡Así,¡Así, siempre así he de verte!
Cadena1 aún mide 30 bytes de largo,
Cadena2 aún mide 30 bytes de largo,
y Cadena3 ahora mide 35 bytes de largo
Cadena1: Allí estás hoy, junto a la tienda de Ayax...
Cadena3 después de strcat2: ¡Así,¡Así, siempre así he de verte!Allí
estás hoy, junto a la tienda de Ayax...
Cadena1 mide ahora 44 bytes de largo,
Cadena2 aún mide 30 bytes de largo,
y Cadena3 mide ahora 79 bytes de largo
```

**ANÁLISIS**

En la línea 4 se incluye el archivo de encabezado `string.h`. Este archivo contiene los prototipos de las funciones para cadenas. La línea 8 asigna un apuntador a una cadena de caracteres ("¡Así, siempre así he de verte!"), y las líneas 9 y 10 asignan dos arreglos de caracteres.

Hasta la línea 32, el listado 12.11 es exactamente igual al listado 12.10 (vea ese listado si necesita una explicación de esas líneas).

La línea 33 cambia a `Cadena1`. No modifica el contenido original ("¡Así, siempre así he de verte!"); ocasiona que `Cadena1` apunte a una cadena diferente en memoria (" Allí estás hoy, junto a la tienda de Ayax..."). Ésta es una distinción sutil, pero muy importante, ya que no debe utilizar un apuntador a cadena (como `Cadena1`) como destino en `strcpy()`, `strncpy()`, `strcat()` o en funciones similares.

Tampoco puede codificar algo como lo siguiente:

```
Cadena2 = "esto no compilará";
```

Esto no funciona debido a que la variable `Cadena2` fue asignada como un arreglo, y no se permite cambiar la constante de la dirección que marca el principio del arreglo.

La línea 35 es igual que la línea 25. La salida de la línea 36 será distinta de la de la línea 26, ya que `Cadena1` es diferente y `Cadena3` retiene su contenido en la línea 25.

La versión final de `Cadena3` no es muy legible debido a que no existen separadores entre las cadenas concatenadas en ella. Una buena técnica es utilizar `strcat()` en otro carácter en la cadena de destino (como un espacio en blanco) de la siguiente forma:

```
strcpy(result_cadena, cadena_original);
strcat(result_cadena, " ");
strcat(result_cadena, cadena_siguiente);
```

Aparecerá un espacio entre el contenido de `cadena_original` y `cadena_siguiente` en la variable `result_cadena`. La función `strcat()` requiere una cadena, lo que explica por qué en la primera llamada el espacio se encierra entre comillas dobles (para convertirlo en una cadena de longitud uno).

Puede utilizar apuntadores a cadenas con cualquiera de las funciones para cadenas.

## Clases de cadenas

La mayoría de los compiladores de C++ viene con una biblioteca de clases que incluye un gran conjunto de clases para manipulación de datos. Un componente estándar de una biblioteca de clases es la clase `String`.

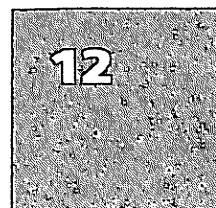
C++ heredó de C la cadena con terminación nula y la biblioteca de funciones que incluye `strcpy()`, pero estas funciones no están integradas en una estructura orientada a objetos. Una clase `String` proporciona un conjunto encapsulado de datos, funciones para manipular esos datos, así como funciones de acceso para que los mismos datos estén ocultos para los clientes de la clase `String`.

Si su compilador no le proporciona una clase `String` (y tal vez aunque lo haga), quizás usted quiera escribir su propia clase. GNU proporciona una clase `String`. El resto de esta lección hablará sobre el diseño y la implementación parcial de clases `String`.

Como mínimo, una clase `String` debe sobrepasar las limitaciones básicas de los arreglos de caracteres. Como todos los arreglos, los arreglos de caracteres son estáticos. Usted define su tamaño. Siempre ocupan el mismo espacio en memoria, incluso si no se necesita todo. Es desastroso escribir más allá del final del arreglo.

Una buena clase `String` asigna sólo la memoria que necesite, y siempre es la suficiente para guardar lo que reciba. Si no puede asignar suficiente memoria, debe fallar con elegancia.

El listado 12.12 proporciona una primera aproximación de una clase `String`.



**ENTRADA****LISTADO 12.12 Uso de una clase String**

```
1: //Listado 12.12 Uso de la clase String
2:
3: #include <iostream.h>
4: #include <string.h>
5:
6: // Clase String rudimentaria
7: class String
8: {
9: public:
10:    // constructores
11:    String();
12:    String(const char * const );
13:    String(const String & );
14:    -String();
15:    // operadores sobrecargados
16:    char & operator[] (unsigned short offset );
17:    char operator[] (unsigned short offset ) const;
18:    String operator+ (const String & );
19:    void operator+= (const String & );
20:    String & operator= (const String & );
21:    // Métodos generales de acceso
22:    unsigned short GetLen()const
23:        { return itsLen; }
24:    const char * GetString() const
25:        { return itsString; }
26: private:
27:    String (unsigned short ); // constructor privado
28:    char * itsString;
29:    unsigned short itsLen;
30: };
31:
32: // constructor predeterminado crea una cadena de 0 bytes
33: String::String()
34: {
35:     itsString = new char[ 1 ];
36:     itsString[ 0 ] = '\0';
37:     itsLen = 0;
38: }
39:
40: // constructor privado (ayudante), sólo lo utilizan
41: // los métodos de la clase para crear una cadena nueva del
42: // tamaño requerido. Llena de caracteres nulos.
43: String::String(unsigned short len )
44: {
45:     itsString = new char[ len + 1 ];
46:
47:     for (unsigned short i = 0; i <= len; i++ )
48:         itsString[ i ] = '\0';
49:     itsLen = len;
50: }
```

```
51: // Convierte un arreglo de caracteres en una Cadena
52: String::String(const char * const cString )
53: {
54:     itsLen = strlen(cString );
55:     itsString = new char[ itsLen + 1 ];
56:     for (unsigned short i = 0; i < itsLen; i++ )
57:         itsString[ i ] = cString[ i ];
58:     itsString[ itsLen ] = '\0';
59: }
60:
61:
62:
63: // constructor de copia
64: String::String(const String & rhs )
65: {
66:     itsLen = rhs.GetLen();
67:     itsString = new char[ itsLen + 1 ];
68:
69:     for (unsigned short i = 0; i < itsLen; i++ )
70:         itsString[ i ] = rhs[ i ];
71:     itsString[ itsLen ] = '\0';
72: }
73:
74: // destructor, libera la memoria asignada
75: String::~String ()
76: {
77:     delete [] itsString;
78:     itsLen = 0;
79: }
80:
81: // operador igual a, libera la memoria existente
82: // luego copia la cadena y el tamaño
83: String & String::operator= (const String & rhs )
84: {
85:     if (this == &rhs )
86:         return *this;
87:     delete [] itsString;
88:     itsLen = rhs.GetLen();
89:     itsString = new char[ itsLen + 1 ];
90:     for (unsigned short i = 0; i < itsLen; i++ )
91:         itsString[ i ] = rhs[ i ];
92:     itsString[ itsLen ] = '\0';
93:     return *this;
94: }
95:
96: //operador de desplazamiento no constante, i regresa
97: // referencia a carácter para que se pueda
98: // cambiar!
99: char & String::operator[] (unsigned short offset )
100: {
101:     if (offset > itsLen )
102:         return itsString[ itsLen - 1 ];
```

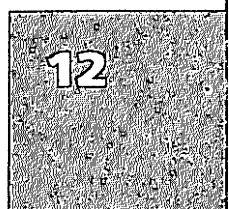
**LISTADO 12.12 CONTINUACIÓN**

```
103:     else
104:         return itsString[ offset ];
105:     }
106:
107: // operador de desplazamiento constante para utilizar
108: // en objetos const (ivea el constructor de copia!)
109: char String::operator[] (unsigned short offset ) const
110: {
111:     if (offset > itsLen )
112:         return itsString[ itsLen - 1 ];
113:     else
114:         return itsString[ offset ];
115: }
116:
117: // crea una cadena nueva al agregar la cadena
118: // actual a rhs
119: String String::operator+ (const String & rhs )
120: {
121:     unsigned short totalLen = itsLen + rhs.GetLen();
122:     String temp(totalLen );
123:     unsigned short i;
124:
125:     for (i = 0; i < itsLen; i++ )
126:         temp[ i ] = itsString[ i ];
127:     for (unsigned short j = 0; j < rhs.GetLen(); j++, i++ )
128:         temp[ i ] = rhs[ j ];
129:     temp[ totalLen ] = '\0';
130:     return temp;
131: }
132:
133: // cambia cadena actual, no regresa nada
134: void String::operator+= (const String & rhs )
135: {
136:     unsigned short rhsLen = rhs.GetLen();
137:     unsigned short totalLen = itsLen + rhsLen;
138:     String temp(totalLen );
139:     unsigned short i;
140:
141:     for (i = 0; i < itsLen; i++ )
142:         temp[ i ] = itsString[ i ];
143:     for (unsigned short j = 0; j < rhs.GetLen(); j++, i++ )
144:         temp[ i ] = rhs[ i - itsLen ];
145:     temp[ totalLen ] = '\0';
146:     *this = temp;
147: }
148:
149: int main()
150: {
151:     String s1("Prueba inicial" );
152:     cout << "S1:\t" << s1.GetString() << endl;
153:
```

```
154:     char * temp = "¡Hola, mundo!";
155:     s1 = temp;
156:     cout << "S1:\t" << s1.GetString() << endl;
157:
158:     char tempDos[ 26 ];
159:     strcpy(tempDos,"; ies grandioso estar aquí! ");
160:     s1 += tempDos;
161:     cout << "tempDos:\t" << tempDos << endl;
162:     cout << "S1:\t" << s1.GetString() << endl;
163:
164:     cout << "S1[3]:\t" << s1[ 3 ] << endl;
165:     s1[ 3 ]='x';
166:     cout << "S1:\t" << s1.GetString() << endl;
167:
168:     cout << "S1[999]:\t" << s1[ 999 ] << endl;
169:
170:     String s2(" Otra cadena" );
171:     String s3;
172:     s3 = s1 + s2;
173:     cout << "S3:\t" << s3.GetString() << endl;
174:
175:     String s4;
176:     s4 = "¿Por qué trabaja esta función?";
177:     cout << "S4:\t" << s4.GetString() << endl;
178:     return 0;
179: }
```

**SALIDA**

```
S1:      Prueba inicial
S1:      ¡Hola, mundo!
tempDos:          ; ies grandioso estar aquí!
S1:      ¡Hola, mundo!; ies grandioso estar aquí!
S1[3]:   1
S1:      ¡Hoxa, mundo!; ies grandioso estar aquí!
S1[999]:   !
S3:      ¡Hoxa, mundo!; ies grandioso estar aquí! Otra cadena
S4:      ¿Por qué trabaja esta función?
```

**ANÁLISIS**

Las líneas 7 a 30 son la declaración de una clase **String** simple. Las líneas 11 a 13 contienen tres constructores: el constructor predeterminado, el constructor de copia y un constructor que toma una cadena existente que termina con un carácter nulo (estilo C).

La clase **String** sobrecarga los operadores de desplazamiento ([ ]) de suma (+) y más igual a (+=). El operador de desplazamiento se sobrecarga dos veces: una vez como función constante que regresa un **char** y otra vez como función no constante que regresa una referencia a **char**.

La versión no constante se utiliza en instrucciones como la siguiente:

```
UnaCadena[ 3 ] = 'x';
```

como se ve en la línea 165. Esto permite un acceso directo a cada uno de los caracteres de la cadena. Se regresa una referencia al carácter para que la función que hace la llamada pueda manipular dicha referencia.

La versión constante se utiliza cuando se accede a un objeto `String` constante, como en la implementación del constructor de copia (línea 64). Observe que se accede a `rhs[i]`, aunque `rhs` esté declarado como `const String &`. No es legal tener acceso a este objeto mediante el uso de un método no constante. Por lo tanto, el operador de desplazamiento se debe sobrecargar con un método de acceso constante.

Si el objeto regresado fuera grande, tal vez sería mejor declarar el valor de retorno como una referencia constante. Sin embargo, como un `char` sólo mide 1 byte, no tendría caso hacer eso.

En las líneas 33 a 38 se implementa el constructor predeterminado. Éste crea una cadena de longitud 0. La convención de esta clase `String` es reportar su longitud sin contar el terminador nulo. Esta cadena predeterminada contiene sólo un terminador nulo.

En las líneas 64 a 72 se implementa el constructor de copia. Éste establece la longitud de la cadena nueva igual a la de la cadena existente (más 1 para el terminador nulo). Copia en la cadena nueva cada carácter de la cadena existente, y agrega un terminador nulo a la cadena nueva.

En las líneas 53 a 61 se implementa el constructor que toma una cadena existente estilo C. Este constructor es similar al constructor de copia. La longitud de la cadena existente se establece mediante una llamada a la función `strlen()` de la biblioteca estándar `String`.

En la línea 27 se declara otro constructor, `String(unsigned short)`, como función miembro privada. La intención del diseñador de esta clase es que ninguna clase vaya a crear un `String` de longitud arbitraria. Este constructor existe sólo para ayudar en la creación interna de `Strings` según se requiera; por ejemplo, según lo requiera el operador `+=`, en la línea 134. Esto se describirá con detalle más adelante, cuando hablaremos sobre `operator+=`.

El constructor `String(unsigned short)` llena cada elemento de su arreglo con `NULL`. Por lo tanto, el ciclo `for` evalúa `i<=len` en lugar de `i<len`.

El destructor, que se implementa en las líneas 75 a 79, elimina la cadena de caracteres mantenida por la clase. Asegúrese de incluir los corchetes en la llamada al operador `delete` para que se eliminan todos los miembros del arreglo, en lugar de que se elimine sólo el primero.

El operador de asignación primero comprueba si el lado derecho de la asignación es igual que el lado izquierdo. Si no lo es, se elimina la cadena actual, y se crea y se copia la cadena nueva. Se regresa una referencia para facilitar asignaciones como la siguiente:

```
String1 = String2 = String3;
```

El operador de desplazamiento se sobrecarga dos veces. En ambas ocasiones se realiza un chequeo rudimentario de los límites. Si el usuario intenta tener acceso a un carácter que se encuentre en una ubicación más allá del final del arreglo, se regresa el último carácter (es decir, `len - 1`).

En las líneas 119 a 131 se implementa el operador de suma (+) como operador de concatenación. Es conveniente poder escribir

```
String3 = String1 + String2;
```

y que `String3` sea la concatenación de las otras dos cadenas. Para lograr esto, la función del operador de suma (+) calcula la longitud combinada de las dos cadenas y crea una cadena temporal llamada `temp`. Esto invoca al constructor privado, el cual toma un entero y crea una cadena llena de caracteres nulos. Estos caracteres se reemplazan a continuación por el contenido de las dos cadenas. Primero se copia la cadena del lado izquierdo (`*this`), y luego la del lado derecho (`rhs`).

El primer ciclo `for` avanza por la cadena del lado izquierdo y agrega cada carácter a la cadena nueva. El segundo ciclo `for` avanza a través del lado derecho. Observe que `i` sigue contando la ubicación para la nueva cadena, incluso cuando `j` cuenta en la cadena `rhs`.

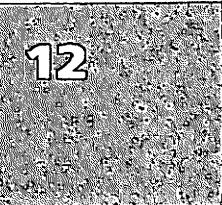
El operador de suma regresa la cadena `temp` por valor, la cual se asigna a la cadena del lado izquierdo de la asignación (`String3`). El operador `+=` actúa sobre la cadena existente (es decir, el lado izquierdo de la instrucción `string1 += string2`). Este operador funciona igual que el operador de suma, excepto que el valor `temp` se asigna a la cadena actual (`*this = temp`) en la línea 146.

La función `main()` (líneas de la 149 a la 179) actúa como un programa controlador de prueba para esta clase. La línea 151 crea un objeto `String` mediante el uso del constructor que toma como parámetro una cadena estilo C que termina con un carácter nulo. La línea 152 imprime su contenido mediante el uso de la función de acceso `GetString()`. La línea 154 crea otra cadena estilo C. La línea 155 prueba el operador de asignación, y la línea 156 imprime los resultados.

La línea 158 crea una tercera cadena estilo C llamada `tempDos`. La línea 159 invoca a `strcpy` para llenar el búfer con los caracteres ; ies grandioso estar aquí! La línea 160 invoca al operador `+=` y concatena a `tempDos` con la cadena existente `s1`. La línea 162 imprime los resultados.

En la línea 164 se accede al cuarto carácter de `s1`, y también se imprime. En la línea 165 se le asigna un nuevo valor. Esto invoca al operador de desplazamiento (`[ ]`) que no es constante. La línea 166 imprime el resultado, el cual muestra que, en efecto, se ha cambiado el valor actual.

La línea 168 intenta tener acceso a un carácter que se encuentra más allá del final del arreglo. Se regresa el último carácter del arreglo, como se tenía designado.



Las líneas 170 y 171 crean dos objetos `String` adicionales, y la línea 172 llama al operador de suma. La línea 173 imprime los resultados.

La línea 175 crea un nuevo objeto `String` llamado `s4`. La línea 176 invoca al operador de asignación. La línea 177 imprime los resultados. Usted podría estar pensando: “El operador de asignación está definido para tomar una referencia constante a `String` en la línea 20, pero aquí el programa pasa una cadena estilo C. ¿Por qué es válido esto?”

La respuesta es que el compilador espera un `String`, pero se le proporciona un arreglo de caracteres. Por lo tanto, comprueba si puede crear un `String` a partir de lo que se le proporciona. En la línea 12 se declaró un constructor que crea `Strings` a partir de arreglos de caracteres. El compilador crea un `String` temporal a partir del arreglo de caracteres y lo pasa al operador de asignación. Esto se conoce como conversión explícita, o promoción. Si no se hubiera declarado (y definido) el constructor que toma como parámetro un arreglo de caracteres, la asignación habría generado un error de compilación.

## Listas enlazadas y otras estructuras

Los arreglos son muy parecidos a los artículos de Tupperware. Son excelentes contenedores, pero tienen un tamaño fijo. Si usted escoge un contenedor demasiado grande, desperdicia espacio en su área de almacenamiento. Si escoge uno muy pequeño, su contenido se derrama y se hace todo un desastre.

Una manera de solucionar este problema es con una lista enlazada. Ésta es una estructura de datos que consiste en contenedores pequeños diseñados para ajustarse al tamaño y que se enlazan entre sí según se necesite. El objetivo es escribir una clase que guarde un objeto de sus datos (como un `GATO` o un `Rectángulo`) y que pueda apuntar al siguiente contenedor. Usted crea un contenedor para cada objeto que necesite guardar, y encadena esos contenedores entre sí según lo necesite.

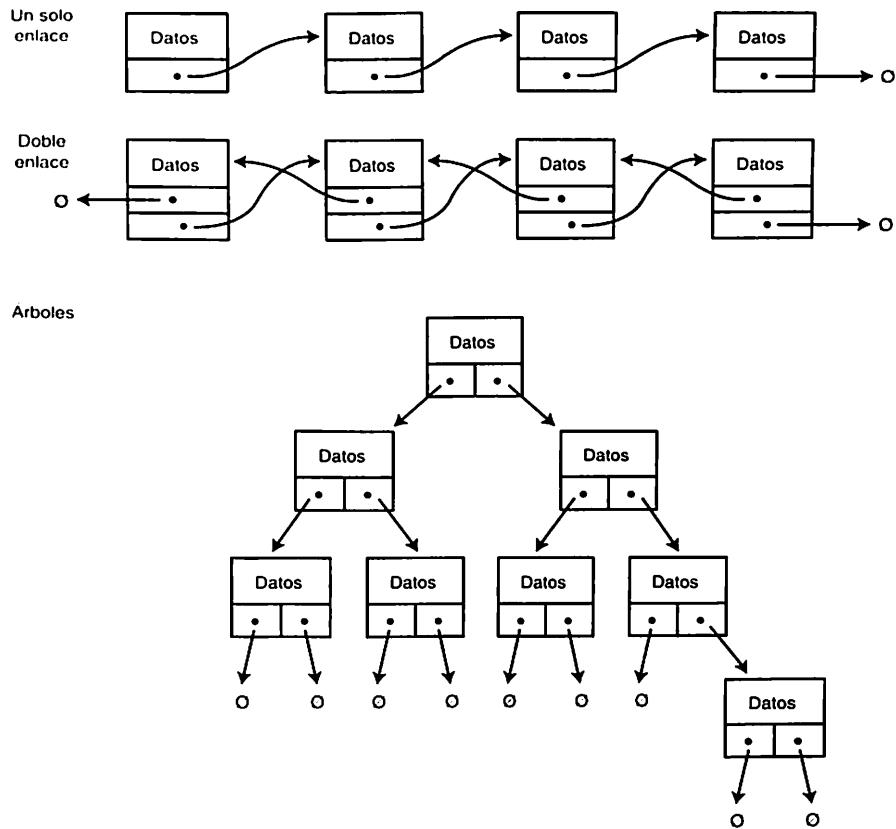
Los contenedores se llaman *nodos*. El primer nodo de la lista se llama *cabeza*, y el último nodo se llama *cola*.

Las listas tienen tres formas básicas. En orden de la más simple a la más compleja, son

- Listas con un solo enlace
- Listas con doble enlace
- Árboles

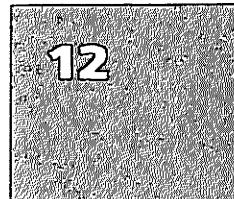
En una lista con un solo enlace, cada nodo apunta hacia el siguiente, pero no hacia el anterior. Para encontrar un nodo específico, se debe empezar en el inicio de la lista y de ahí avanzar de nodo en nodo, como en la búsqueda de un tesoro (“El siguiente nodo está debajo del sofá”). Una lista con doble enlace le permite moverse hacia atrás y hacia adelante en la lista. Un árbol es una estructura compleja creada a partir de nodos, cada uno de los cuales puede apuntar hacia dos o más direcciones. La figura 12.5 muestra estas tres estructuras fundamentales.

**FIGURA 12.5**  
*Listas enlazadas.*



## Análisis de un caso de prueba de listas enlazadas

En esta sección examinaremos detalladamente una lista enlazada como caso de estudio sobre la forma en que se crean las estructuras complejas, y especialmente sobre la forma de utilizar la herencia, el poliformismo y el encapsulamiento para manejar proyectos grandes.



### Delegación de responsabilidad

Una premisa fundamental de la programación orientada a objetos es que cada objeto hace algo muy bien y delega a otros objetos cualquier cosa que no sea su misión central.

Un automóvil es un ejemplo perfecto de esta idea sobre el hardware: el trabajo del motor es producir la energía. La distribución de esa energía no es el trabajo del motor; eso es responsabilidad de la transmisión. Girar no es el trabajo del motor ni de la transmisión; eso se delega a la dirección.

Una máquina bien diseñada tiene muchas piezas pequeñas y bien definidas, que hacen su propio trabajo y funcionan entre sí para lograr un mayor beneficio. Un programa bien diseñado es muy parecido: cada clase se dedica a su propio tejido, pero en conjunto crean una estupenda colcha de punto.

## Los componentes de una lista enlazada

La lista enlazada constará de nodos. La clase de nodos en sí será abstracta; utilizaremos tres subtipos para lograr nuestro objetivo. Habrá un nodo cabeza cuyo trabajo será manejar la cabeza de la lista, un nodo cola (¡adivine cuál será su trabajo!), y cero o más nodos internos. Estos nodos se encargarán de que los datos actuales estén guardados en la lista.

Observe que los datos y la lista son bastante distintos. En teoría, usted puede guardar en una lista cualquier tipo de datos. No son los datos los que están enlazados entre sí; es el nodo el que *guarda* los datos.

El programa controlador no sabe nada acerca de los nodos; trabaja con la lista. Sin embargo, la lista hace muy poco trabajo; simplemente lo delega a los nodos.

El listado 12.3 muestra el código de una lista enlazada; lo examinaremos con mucho detalle.

---

**ENTRADA LISTADO 12.13** Lista enlazada

```
1: // Listado 12.13 Muestra un método orientado a objetos para
2: // listas enlazadas. La lista delega responsabilidad al nodo.
3:
4: #include <iostream.h>
5:
6: enum { kEsMasChico, kEsMasGrande, kEsIgual};
7:
8: // Clase de datos que se va a colocar en la lista enlazada
9: // Cualquier clase de esta lista enlazada debe soportar dos métodos:
10: // Mostrar (despliega el valor) y Comparar (regresa la posición relativa)
11: class Datos
12: {
13: public:
14:     Datos(int val) : miValor(val){}
15:     ~Datos(){}
16:     int Comparar(const Datos & );
17:     void Mostrar()
18:     { cout << miValor << endl; }
19: private:
20:     int miValor;
21: };
22:
```

```
23: // Comparar se utiliza para decidir a qué lugar de la lista
24: // pertenece un objeto específico.
25: int Datos::Comparar(const Datos & losOtrosDatos )
26: {
27:     if (miValor < losOtrosDatos.miValor )
28:         return kEsMasChico;
29:     else if (miValor > losOtrosDatos.miValor )
30:         return kEsMasGrande;
31:     else
32:         return kEsIgual;
33: }
34:
35: // ADT que representa al objeto nodo de la lista
36: // Cada clase derivada debe redefinir a Insertar y a Mostrar
37: class Nodo
38: {
39: public:
40:     Nodo(){}
41:     virtual ~Nodo(){}
42:     virtual Nodo * Insertar(Datos * losDatos ) = 0;
43:     virtual void Mostrar() = 0;
44: };
45:
46: // Éste es el nodo que guarda el objeto actual
47: // En este caso el objeto es de tipo Datos
48: // Veremos como generalizar más esto cuando
49: // hablemos sobre las plantillas
50: class NodoInterno: public Nodo
51: {
52: public:
53:     NodoInterno(Datos * losDatos, Nodo * siguiente );
54:     ~NodoInterno()
55:     { delete miSiguiente; delete misDatos; }
56:     virtual Nodo * Insertar(Datos * losDatos );
57:     virtual void Mostrar()
58:     {
59:         // idelegar!
60:         misDatos->Mostrar();
61:         miSiguiente->Mostrar();
62:     }
63:
64: private:
65:     // los datos en si
66:     Datos * misDatos;
67:     // apunta al siguiente nodo de la lista enlazada
68:     Nodo * miSiguiente;
```

**LISTADO 12.13 CONTINUACIÓN**

```
69:     };
70:
71:     // Todo lo que hace el constructor es inicializar
72:     NodoInterno::NodoInterno(Datos * losDatos, Nodo * siguiente ):
73:         misDatos(losDatos ),
74:         miSiguiente(siguiente )
75:     {}
76:
77:     // la parte principal de la lista
78:     // Cuando se coloca un nuevo objeto en la lista
79:     // éste se pasa al nodo que averigua
80:     // en dónde debe ir y lo inserta en la lista
81:     Nodo * NodoInterno::Insertar(Datos * losDatos )
82:     {
83:         // ¿es el nuevo objeto más grande o más pequeño que yo?
84:         int resultado = misDatos->Comparar(*losDatos );
85:         switch(resultado )
86:         {
87:             // por convención si es igual que yo debe ir primero
88:             case kEsIgual:
89:                 // avanzar al siguiente case sin hacer nada
90:             case kEsMasGrande:
91:                 // los datos nuevos van antes de mí
92:                 {
93:                     NodoInterno * nodoDatos = new NodoInterno
94:                         (losDatos, this );
95:                     return nodoDatos;
96:                 }
97:                 // es mayor que yo así que lo paso al siguiente
98:                 // nodo para que se encargue de él.
99:             case kEsMasChico:
100:                 miSiguiente = miSiguiente->Insertar(losDatos );
101:                 return this;
102:             }
103:             return this; // apaciguar al compilador
104:         }
105:
106:
107:        // El nodo cola sólo es un centinela
108:        class NodoCola : public Nodo
109:        {
110:            public:
111:                NodoCola(){}
112:                ~NodoCola(){}}
```

```
113:     virtual Nodo * Insertar(Datos * losDatos );
114:     virtual void Mostrar() {}
115: };
116:
117: // Si los datos llegan a mí, se deben insertar antes de mí
118: // ya que soy la cola y no hay NADA después de mí
119: Nodo * NodoCola::Insertar(Datos * losDatos )
120: {
121:     NodoInterno * nodoDatos = new NodoInterno(losDatos, this );
122:     return nodoDatos;
123: }
124:
125: // El nodo cabeza no tiene datos, sólo apunta
126: // al inicio de la lista
127: class NodoCabeza : public Nodo
128: {
129: public:
130:     NodoCabeza();
131:     ~NodoCabeza()
132:     { delete miSiguiente; }
133:     virtual Nodo * Insertar(Datos * losDatos );
134:     virtual void Mostrar()
135:         { miSiguiente->Mostrar(); }
136: private:
137:     Nodo * miSiguiente;
138: };
139:
140: // Tan pronto como se crea la cabeza
141: // se crea la cola
142: NodoCabeza::NodoCabeza()
143: {
144:     miSiguiente = new NodoCola;
145: }
146:
147: // No hay nada antes de la cabeza así que sólo
148: // se pasan los datos al siguiente nodo
149: Nodo * NodoCabeza::Insertar(Datos * losDatos )
150: {
151:     miSiguiente = miSiguiente->Insertar(losDatos );
152:     return this;
153: }
154:
155: // Yo obtengo todos los méritos y no hago nada del trabajo
156: class ListaEnlazada
157: {
```

12

**LISTADO 12.13** CONTINUACIÓN

```
158:     public:
159:         ListaEnlazada();
160:         ~ListaEnlazada()
161:             { delete miCabeza; }
162:         void Insertar(Datos * losDatos );
163:         void MostrarTodo()
164:             { miCabeza->Mostrar(); }
165:     private:
166:         NodoCabeza * miCabeza;
167:     };
168:
169: // Al nacer, se crea el nodo cabeza
170: // Éste crea el nodo cola
171: // Así que una lista vacía apunta a la cabeza que
172: // apunta a la cola y no tiene nada en medio
173: ListaEnlazada::ListaEnlazada()
174: {
175:     miCabeza = new NodoCabeza;
176: }
177:
178: // Delegar, delegar, delegar
179: void ListaEnlazada::Insertar(Datos * apDatos )
180: {
181:     miCabeza->Insertar(apDatos );
182: }
183:
184: // programa controlador de prueba
185: int main()
186: {
187:     Datos * apDatos;
188:     int val;
189:     ListaEnlazada le;
190:
191:     // pedir al usuario que produzca algunos valores
192:     // colocarlos en la lista
193:     for (;;)
194:     {
195:         cout << "¿Cuál valor? (0 para detener): ";
196:         cin >> val;
197:         if (!val )
198:             break;
199:         apDatos = new Datos(val );
200:         le.Insertar(apDatos );
201:     }
```

---

```

202:     // ahora avanzar por la lista y mostrar los datos
203:     le.MostrarTodo();
204:     return 0; // ile queda fuera de alcance y se destruye!
205: }
```

---

**SALIDA**

¿Cuál valor? (0 para detener): 5  
 ¿Cuál valor? (0 para detener): 8  
 ¿Cuál valor? (0 para detener): 3  
 ¿Cuál valor? (0 para detener): 9  
 ¿Cuál valor? (0 para detener): 2  
 ¿Cuál valor? (0 para detener): 10  
 ¿Cuál valor? (0 para detener): 0  
 2  
 3  
 5  
 8  
 9  
 10

**ANÁLISIS**

Lo primero que hay que observar es la constante enumerada que proporciona tres valores constantes: kEsMasChico, kEsMasGrande y kEsIgual. Cada objeto que se pueda guardar en esta lista enlazada debe soportar un método Comparar(). Estas constantes serán el valor del resultado regresado por el método Comparar().

Para fines ilustrativos, la clase Datos se crea en las líneas 11 a 21, y el método Comparar() se implementa en las líneas 25 a 33. Un objeto Datos guarda un valor y puede compararse a sí mismo con otros objetos Datos. También soporta un método Mostrar() para desplegar su valor (el del objeto Datos).

La manera más sencilla de entender el funcionamiento de la lista enlazada es ejemplificar paso a paso el uso de una. En la línea 185 se define un programa controlador; en la línea 187 se declara un apuntador a un objeto Datos, y en la línea 189 se declara una lista enlazada local.

Cuando se crea la lista enlazada, se llama al constructor en la línea 173. El único trabajo que se hace en el constructor es asignar un objeto NodoCabeza y asignar la dirección de ese objeto al apuntador que se guarda en la lista enlazada de la línea 166.

Esta asignación de un NodoCabeza invoca al constructor NodoCabeza que se muestra en las líneas 142 a 145. Éste a su vez asigna un NodoCola y asigna su dirección al apuntador miSiguiente del nodo cabeza. La creación del NodoCola llama al constructor NodoCola que se muestra en la línea 111, el cual está en línea y no hace nada.

Por lo tanto, con el simple hecho de asignar una lista enlazada en la pila, se crea la lista, se crean un nodo cabeza y un nodo cola, y se establece su relación, como se muestra en la figura 12.6.



**FIGURA 12.6**

*La lista enlazada después de su creación.*



La línea 193 empieza un ciclo infinito. Se pedirán valores al usuario para agregarlos a la lista enlazada. El usuario puede agregar tantos valores como desee, y debe escribir un `0` cuando haya terminado. El código de la línea 197 evalúa el valor escrito; si es igual a `0`, se sale del ciclo.

Si el valor no es `0`, se crea un nuevo objeto `Datos` en la línea 199, y se inserta en la lista en la línea 200. Para fines ilustrativos, suponga que el usuario escribe el valor 15. Esto invoca al método `Insertar` en la línea 179.

La lista enlazada delega inmediatamente la responsabilidad de insertar el objeto a su nodo cabeza. Esto invoca al método `Insertar` en la línea 149. El nodo cabeza pasa de inmediato la responsabilidad a cualquier nodo al que apunte `miSiguiente`. En este caso (cuando se agrega el primer elemento a la lista), está apuntando al nodo cola (recuerde, cuando el nodo cabeza nació, creó un enlace a un nodo cola). Esto, por consecuencia, invoca al método `Insertar` en la línea 119.

`NodoCola::Insertar()` sabe que el objeto que ha recibido debe ser insertado inmediatamente antes que él, es decir, el nuevo objeto estará en la lista justo antes del nodo cola. Por lo tanto, crea un nuevo objeto `NodoInterno` en la línea 121, pasando los datos y un apuntador a él mismo. Esto invoca al constructor para el objeto `NodoInterno`, que se muestra en la línea 72.

El constructor de `NodoInterno` no hace nada más que inicializar su apuntador a `Datos` con la dirección del objeto `Datos` que recibió y a su apuntador `miSiguiente` con la dirección del nodo que recibió. En este caso, el nodo al que apuntará es el nodo cola (recuerde, el nodo cola pasó su propio apuntador `this`).

Ahora que se ha creado `NodoInterno`, se asigna la dirección de ese nodo interno al apuntador `nodoDatos` en la línea 121, y esa dirección se regresa a su vez del método `NodoCola::Insertar()`. Esto nos regresa a `NodoCabeza::Insertar()`, en donde se asigna la dirección de `NodoInterno` al apuntador `miSiguiente` de `NodoCabeza` (en la línea 151). Por último, se regresa la dirección de `NodoCabeza` a la lista enlazada donde, en la línea 181, se descarta (no se hace nada con ella porque la lista enlazada ya conoce la dirección del nodo cabeza).

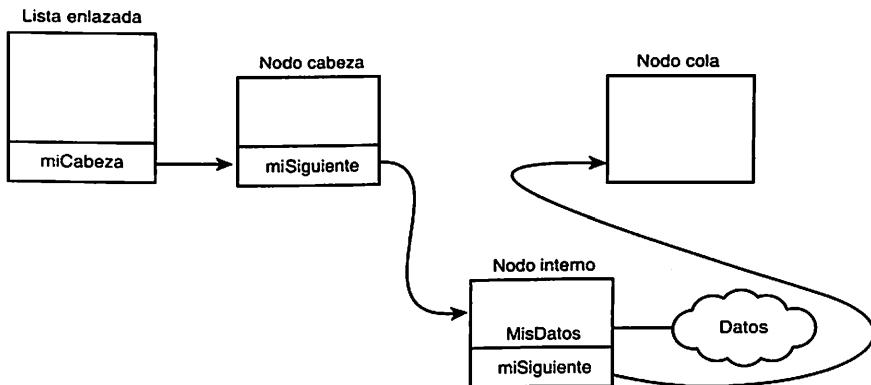
¿Por qué tomarse la molestia de regresar la dirección si no se utiliza? `Insertar` se declara en la clase base `Nodo`. El valor de retorno se necesita para las otras implementaciones. Si se cambia el valor de retorno de `NodoCabeza::Insertar()`, se generará un

error de compilación; es más sencillo regresar el `NodoCabeza` y dejar que la lista enlaza-  
da descarte su dirección.

¿Qué fue lo que pasó? Los datos se insertaron en la lista. La lista los pasó a la cabeza. La  
cabeza, ciegamente, pasó los datos al nodo al que estaba apuntando en ese momento. En  
este caso (cuando se insertó el primer elemento en la lista), la cabeza estaba apuntando a  
la cola. La cola creó inmediatamente un nuevo nodo interno, e inicializó el nuevo nodo para  
que apuntara a ella. Luego la cola regresó la dirección del nuevo nodo a la cabeza, la cual  
reasignó su apuntador `miSiguiente` para que apuntara al nuevo nodo. ¡Listo! Los datos  
están en la lista en el lugar adecuado, como se muestra en la figura 12.7.

**FIGURA 12.7**

*La lista enlazada  
después de haber  
insertado el primer  
nodo.*



Después de insertar el primer nodo, el control del programa continúa en la línea 195. Una vez más se evalúa el valor. Para fines ilustrativos, suponga que se escribe el valor 3. Esto ocasiona que se cree un nuevo objeto `Datos` en la línea 199 y que se inserte a la lista en la línea 200.

Una vez más, en la línea 181, la lista pasa los datos a su `NodoCabeza`. A su vez, el método `NodoCabeza::Insertar()` pasa el nuevo valor al nodo al que `miSiguiente` esté apun-  
tando en ese momento. Como sabe, ahora está apuntando al nodo que contiene el objeto `Datos` cuyo valor es 15. Esto invoca al método `NodoInterno::Insertar()` en la línea 81.

En la línea 84, `NodoInterno` utiliza su apuntador `misDatos` para indicar a su objeto `Datos` (el que tiene el valor 15) que llame a su método `Comparar()`, y le pasa el nuevo objeto `Datos` (cuyo valor es 3). Esto invoca al método `Comparar()` definido en la línea 25.

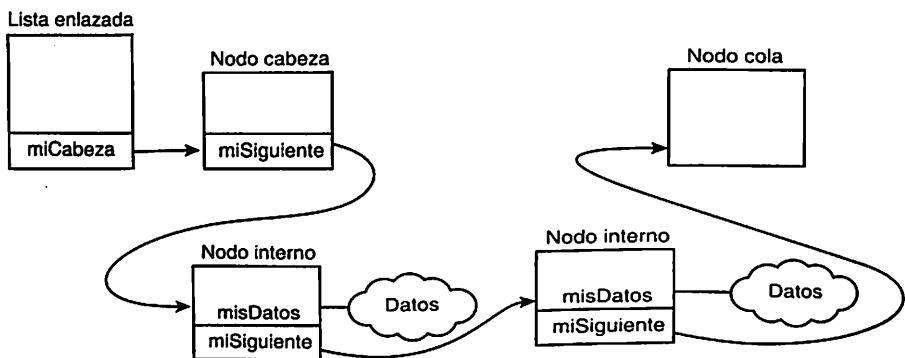
Los dos valores se comparan y, como `miValor` será 15 y `losOtrosDatos.miValor` será 3, el valor regresado será `kEsMasGrande`. Esto ocasionará que el flujo del programa salte a la línea 93.

Se crea un nuevo `NodoInterno` para el nuevo objeto `Datos`. El nuevo nodo apuntará al objeto `NodoInterno` actual, y la dirección del nuevo `NodoInterno` se regresa del método

`NodoInterno::Insertar()` al `NodoCabeza`. Por lo tanto, el nuevo nodo, cuyo objeto tiene un valor menor que el del objeto del nodo actual, se inserta en la lista, y la lista ahora se ve como la figura 12.8.

**FIGURA 12.8**

*La lista enlazada después de haber insertado el segundo nodo.*



En la tercera invocación del ciclo, el cliente agrega el valor 8. Éste es mayor que 3 pero menor que 15, y por lo tanto se debe insertar entre los dos nodos existentes. El progreso será como en el ejemplo anterior, excepto que cuando el nodo cuyo objeto tiene el valor 3 haga la comparación, en lugar de regresar `kEsMasGrande`, regresará `kEsMasChico` (lo cual indica que el objeto cuyo valor es 3 es más chico que el objeto nuevo, cuyo valor es 8).

Esto ocasionará que el método `NodoInterno::Insertar()` se ramifique hacia la línea 100. En lugar de crear un nuevo nodo e insertarlo, `NodoInterno` sólo pasará los datos nuevos al método `Insertar` del nodo al que esté apuntando en ese momento el apuntador `miSiguiente`. En este caso, invocará a `Insertar` en el `NodoInterno` cuyo objeto `Datos` es 15.

Se hará otra vez la comparación y se creará un nuevo `NodoInterno`. Este nuevo `NodoInterno` apuntará al `NodoInterno` cuyo objeto `Datos` tiene el valor 15, y su dirección se pasará de regreso al `NodoInterno` cuyo objeto `Datos` tiene el valor 3, como se muestra en la línea 100.

El efecto obtenido es que el nuevo nodo se insertará en la lista en la ubicación correcta.

De ser posible, trate de seguir paso a paso la inserción de varios nodos en su depurador. Debe ver a estos métodos invocarse entre sí y a los apuntadores ajustarse apropiadamente.

## ¿Qué ha aprendido?

Dorothy dijo: "Si alguna vez vuelvo a tratar de seguir los sentimientos de mi corazón, no iré más allá de mi propio jardín". Aunque es cierto que no hay ningún lugar como el hogar, también es cierto que esto no es nada parecido a la programación procedural. En este tipo de programación, una función controladora examinaría los datos e invocaría a las funciones correspondientes.

En la programación orientada a objetos, a cada objeto individual se le otorga un conjunto ajustado y bien definido de responsabilidades. La lista enlazada es responsable de mantener el nodo cabeza. El nodo cabeza pasa inmediatamente los datos nuevos al nodo al que esté apuntando en ese momento, sin importar qué nodo sea.

El nodo cola crea un nodo nuevo y lo inserta siempre que reciba datos. Sólo sabe una cosa: si llegó a mí, se inserta antes de mí.

Los nodos internos son un poco más complicados; piden a su objeto existente que se compare con el nuevo objeto. Dependiendo del resultado, lo insertan o sólo lo pasan al siguiente nodo.

Observe que `NodoInterno` no tiene idea de cómo hacer la comparación; eso se deja al objeto en sí. Todo lo que `NodoInterno` sabe es pedir a los objetos que se comparan entre sí y que esperen una de tres posibles respuestas. Dada una respuesta, hace la inserción; de no ser así, sólo pasa el objeto nuevo al siguiente nodo, sin saber ni importarle en dónde acabará.

Así que, ¿cuál objeto está a cargo? En un programa orientado a objetos bien diseñado, no hay ni un objeto a cargo. Cada objeto hace su propio trabajo, y el efecto obtenido es una máquina que funciona perfectamente.

12

## Uso de clases de arreglos en lugar de arreglos integrados

Escribir su propia clase que manipule arreglos tiene muchas ventajas en comparación con el uso de los arreglos integrados. Para empezar, puede prevenir el desbordamiento de arreglos. También podría considerar crear su clase de arreglos con tamaño dinámico: Al momento de la creación podría tener un solo miembro, y crecer según sea necesario durante el funcionamiento del programa.

Tal vez también quiera ordenar los miembros del arreglo. Podría considerar una cantidad de variantes de arreglos poderosas. Entre las más populares están

- Colección ordenada: Cada miembro está colocado en orden.
- Conjunto: Ningún miembro aparece más de una vez.
- Diccionario: Utiliza pares relacionados en los que un valor actúa como clave para recuperar el otro valor.

- **Arreglo disperso:** Se permiten índices con valores no secuenciales, y sólo aquellos valores que se agregan al arreglo consumen memoria. Por lo tanto, puede pedir el valor de `ArregloDisperso[5]` o de `ArregloDisperso[200]`, pero es posible que la memoria esté asignada sólo para un número pequeño de entradas.
- **Bolsa:** Una colección desordenada a la que se agregan y quitan elementos en orden aleatorio.

Por medio de la sobrecarga del operador de índice (`[ ]`), puede convertir una lista enlazada en una colección ordenada. Por medio de la eliminación de duplicados, puede convertir una colección en un conjunto. Si cada objeto de la lista tiene un par de valores relacionados, puede utilizar una lista enlazada para crear un diccionario o un arreglo disperso.

## Resumen

Hoy aprendió cómo crear arreglos en C++. Un arreglo es una colección de tamaño fijo de objetos del mismo tipo.

Los arreglos no hacen comprobación de sus límites. Por lo tanto es válido, aunque desastroso, leer o escribir más allá del final del arreglo. Los arreglos empiezan a contar desde 0. Un error común es escribir al desplazamiento  $n$  de un arreglo con  $n$  elementos.

Los arreglos pueden ser unidimensionales o multidimensionales. En cualquier caso, los elementos del arreglo se pueden inicializar, siempre y cuando el arreglo contenga ya sea tipos integrados, como `int`, u objetos de una clase que tenga un constructor predeterminado.

Los arreglos y su contenido pueden estar en el heap o en la pila. Si elimina un arreglo en el heap, recuerde utilizar los corchetes (`[]`) en la llamada a `delete`.

Los nombres de arreglos son apunadores constantes al primer elemento del arreglo. Los apunadores y arreglos utilizan aritmética de apunadores para encontrar el siguiente elemento de un arreglo.

Puede crear listas enlazadas para manejar colecciones con tamaños que no conozca en tiempo de compilación. A partir de las listas enlazadas, puede crear cualquier número de estructuras de datos más complejas.

Las cadenas son arreglos de caracteres, o `chars`. C++ proporciona características especiales para manejar cadenas de valores tipo `char`, incluyendo la capacidad de inicializarlas con cadenas encerradas entre comillas.

## Preguntas y respuestas

**P** ¿Qué pasa si escribo en el elemento 25 de un arreglo de 24 miembros?

**R** Escribirá en otra área de memoria, con efectos potencialmente desastrosos en su programa.

**P ¿Qué hay en un elemento no inicializado de un arreglo?**

**R** Lo que haya en la memoria en cualquier momento dado. Los resultados de utilizar este miembro sin asignarle un valor son impredecibles.

**P ¿Puedo combinar arreglos?**

**R** Sí. Con arreglos simples puede utilizar apuntadores para combinarlos en un arreglo nuevo y más grande. Con cadenas, puede utilizar algunas de las funciones integradas, como `strcat`, para combinar cadenas.

**P ¿Por qué debo crear una lista enlazada si un arreglo puede funcionar?**

**R** Un arreglo debe tener un tamaño fijo, mientras que una lista enlazada puede cambiar su tamaño dinámicamente en tiempo de ejecución.

**P ¿Por qué usar arreglos integrados si puedo crear una clase de arreglos mejor?**

**R** Los arreglos integrados son rápidos y fáciles de utilizar.

**P ¿Debe una clase de cadenas utilizar un `char *` para guardar el contenido de la cadena?**

**R** No. Puede utilizar el almacenamiento de memoria que el diseñador considere más conveniente.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

12

### Cuestionario

1. ¿Cuáles son el primero y último elementos en `UnArreglo[25]`?
2. ¿Cómo se declara un arreglo multidimensional?
3. Inicialice los miembros del arreglo de la pregunta 2.
4. ¿Cuántos elementos hay en el arreglo `UnArreglo[10][5][20]`?
5. ¿Cuál es el número máximo de elementos que se pueden agregar a una lista enlazada?
6. ¿Puede utilizar notación de subíndice en una lista enlazada?
7. ¿Cuál es el último carácter de la cadena “Brad es una buena persona”?

## Ejercicios

1. Declare un arreglo de dos dimensiones que represente un tablero del juego tic-tac-toe.
2. Escriba el código que inicialice con 0 todos los elementos del arreglo que creó en el ejercicio 1.
3. Escriba la declaración de una clase Nodo que guarde enteros.
4. **CAZA ERRORES:** ¿Qué está mal en este fragmento de código?

```
unsigned short UnArreglo[5][4];
for (int i = 0; i<4; i++)
    for (int j = 0; j<5; j++)
        UnArreglo[i][j] = i+j;
```

5. **CAZA ERRORES:** ¿Qué está mal en este fragmento de código?

```
unsigned short UnArreglo[5][4];
for (int i = 0; i<=5; i++)
    for (int j = 0; j<=4; j++)
        UnArreglo[i][j] = 0;
```

# SEMANA 2

DÍA 13

## Polimorfismo

Ayer aprendió cómo escribir funciones virtuales en clases derivadas. Éste es el bloque de construcción fundamental del polimorfismo: la capacidad de enlazar objetos específicos de clases derivadas con apuntadores a clases base en tiempo de ejecución. Hoy aprenderá lo siguiente:

- Qué es la herencia múltiple y cómo utilizarla
- Qué es la herencia virtual
- Qué son los tipos de datos abstractos
- Qué son las funciones virtuales puras

## Problemas con herencia simple

Suponga que está trabajando con sus clases de animales por un tiempo, y divide la jerarquía de clases en `Aves` y `Mamíferos`. La clase `Ave` incluye el método (o función miembro) `Volar()`. La clase `Mamífero` se ha dividido en varios tipos de `Mamíferos`, incluyendo `Caballo`. Esta clase incluye los métodos `Relinchar()` y `Galopar()`.

De repente, se da cuenta de que necesita un objeto `Pegaso`: una cruce entre un `Caballo` y un `Ave`. Un `Pegaso` puede `Volar()`, puede `Relinchar()` y puede `Galopar()`. Con la herencia simple, se encuentra en un gran aprieto.

Puede hacer que Pegaso sea un Ave, pero entonces no podrá Relinchar() ni Galopar(). Puede hacerlo un Caballo, pero entonces no podrá Volar().

Su primera solución puede ser copiar el método Volar() en la clase Pegaso y derivar a Pegaso de Caballo. Esto funciona bien, lo único malo es que tiene el método Volar() en dos lugares (Ave y Pegaso). Si cambia uno, debe recordar cambiar el otro. Claro que un desarrollador que le dé mantenimiento a su código meses o años después, debe saber también que tiene que arreglar ambos lugares.

Sin embargo, pronto tendrá un nuevo problema. Suponga que quiere crear una lista de objetos Caballo y una lista de objetos Ave. Le gustaría poder agregar sus objetos Pegaso a las dos listas, pero si Pegaso es un Caballo, no puede agregarlo a una lista de Aves.

Hay un par de soluciones potenciales. Puede cambiar el nombre del método Galopar() de Caballo a Mover(), y luego puede redefinir Mover() en su objeto Pegaso para que haga el trabajo de Volar(). Podría entonces redefinir el método Mover() de sus otros caballos para que hagan el trabajo de Galopar(). Pegaso podría ser lo suficientemente listo como para galopar distancias cortas y volar distancias más largas.

```
Pegaso::Mover(long distancia)
{
    if (distancia > muyLejos)
        volar(distancia);
    else
        galopar(distancia);
}
```

Esto es un poco limitante. Tal vez un día Pegaso quiera volar una distancia corta o galopar una distancia larga. Otra solución podría ser definir el método Volar() en la clase Caballo, como se muestra en el listado 13.1. El problema es que los caballos no pueden volar, por lo que tendría que hacer que este método no hiciera nada a menos que fuera un Pegaso.

**ENTRADA****LISTADO 13.1 Si los caballos pudieran volar...**

```
1: // Listado 13.1. Si los caballos pudieran volar...
2: // Infiltando a Volar() en Caballo
3:
4: #include <iostream.h>
5:
6: class Caballo
7: {
8: public:
9:     void Galopar(){ cout << "Galopando...\n"; }
10:    virtual void Volar() { cout << "Los caballos no pueden volar.\n" ; }
11: private:
12:     int suEdad;
13: };
14:
```

```
15: class Pegaso : public Caballo
16: {
17:     public:
18:         virtual void Volar() { cout <<
19:             "¡Puedo volar! ¡Puedo volar! ¡Puedo volar!\n"; }
20:     };
21:     const int NumeroCaballos = 5;
22:     int main()
23:     {
24:         Caballo* Rancho[NumeroCaballos];
25:         Caballo* apCaballo;
26:         int opcion,i;
27:         for (i=0; i<NumeroCaballos; i++)
28:         {
29:             cout << "(1)Caballo (2)Pegaso: ";
30:             cin >> opcion;
31:             if (opcion == 2)
32:                 apCaballo = new Pegaso;
33:             else
34:                 apCaballo = new Caballo;
35:             Rancho[i] = apCaballo;
36:         }
37:         cout << "\n";
38:         for (i=0; i<NumeroCaballos; i++)
39:         {
40:             Rancho[i]->Volar();
41:             delete Rancho[i];
42:         }
43:         return 0;
44:     }
```

**SALIDA**

```
(1)Caballo (2)Pegaso: 1
(1)Caballo (2)Pegaso: 2
(1)Caballo (2)Pegaso: 1
(1)Caballo (2)Pegaso: 2
(1)Caballo (2)Pegaso: 1

Los caballos no pueden volar.
¡Puedo volar! ¡Puedo volar! ¡Puedo volar!
Los caballos no pueden volar.
¡Puedo volar! ¡Puedo volar! ¡Puedo volar!
Los caballos no pueden volar.
```

**13****ANÁLISIS**

Este programa evidentemente funciona, aunque con la desventaja de que la clase Caballo tiene un método `Volar()`. En la línea 10 se le proporciona a Caballo el método `Volar()`. En una clase real, esta solución podría hacer que el compilador emitiera un error, o que el programa fallara silenciosamente. En la línea 18, la clase Pegaso redefine el método `Volar()` para “que haga lo correcto”, que se representa aquí con la impresión de un feliz mensaje.

El arreglo de apuntadores a `Caballo` de la línea 24 se utiliza para demostrar que se llama al método `Volar()` apropiado, con base en el enlace en tiempo de ejecución del objeto `Caballo` o del objeto `Pegaso`.



### Nota

Estos ejemplos se han simplificado hasta lo más esencial para exemplificar los puntos bajo consideración. Los constructores, destructores virtuales y demás se han eliminado para mantener sencillo el código.

## Filtración ascendente

Colocar la función requerida en los niveles superiores de la jerarquía de clases es una solución común para este problema y el resultado es que muchas funciones “se filtran” dentro de la clase base. La clase base se encuentra entonces en grave peligro de convertirse en un espacio de nombres global para todas las funciones que podrían ser utilizadas por alguna de las clases derivadas. Esto puede minar seriamente la tipificación de clases de C++, y puede crear una clase base grande y difícil.

En general, usted necesita filtrar la funcionalidad compartida que se define en las capas superiores de la jerarquía, sin migrar la interfaz de cada clase. Esto significa que si dos clases comparten una clase base común (por ejemplo, `Caballo` y `Ave` comparten `Animal`) y tienen una función en común (por ejemplo, tanto las aves como los caballos comen), usted necesitaría llevar esa funcionalidad hacia la clase base y crear una función virtual.

No obstante, lo que necesita evitar es filtrar una interfaz definida en capas superiores (digamos, el método `Volar()`), para que pueda llamar a ese método sólo en algunas clases derivadas. Si usted tiene que realizar este tipo de filtros, entonces su interfaz no pertenece a esa capa en la jerarquía de clases.

## Conversión descendente

Una alternativa para el método anterior, que aún se encuentra dentro de la herencia simple, es mantener el método `Volar()` dentro de `Pegaso` y sólo llamarlo si el apuntador está realmente apuntando a un objeto `Pegaso`. Para hacer esto, necesitará poder preguntar a su apuntador a qué tipo apunta en realidad. Esto se conoce como Identificación de Tipo en Tiempo de Ejecución (RTTI). RTTI se convirtió hasta hace poco en un componente oficial de C++.

Si su compilador no soporta RTTI, puede imitarlo colocando un método que regrese un tipo enumerado en cada una de las clases. Entonces puede probar ese tipo en tiempo de ejecución y llamar a `Volar()` si regresa `Pegaso`.

Las primeras versiones de los compiladores GNU (2.7.2 y anteriores) no soportan RTTI en todas las plataformas. La versión que se incluye en el CD-ROM (2.9.5) sí lo soporta (de manera predeterminada).

**Nota**

Evite el uso de RTTI en sus programas. Esto puede ser el indicador de un mal diseño. En vez de eso, considere el uso de las funciones virtuales, plantillas o herencia múltiple.

Para llamar a `Volar()`, debe convertir el apuntador, y debe indicarle que el objeto al que está apuntando es un objeto `Pegaso`, no un `Caballo`. Esto se conoce como conversión descendente, ya que se está convirtiendo el objeto `Caballo` en un tipo más derivado.

Ahora, C++ soporta oficialmente, aunque tal vez en forma renuente, la conversión descendente por medio del nuevo operador `dynamic_cast`. Este operador funciona así:

Si tiene un apuntador a una clase base, como `Caballo`, y lo asigna a un apuntador a una clase derivada, como `Pegaso`, puede utilizar el apuntador a `Caballo` de varias formas. Si luego necesita tener acceso al objeto `Pegaso`, puede crear un apuntador a `Pegaso` y utilizar el operador `dynamic_cast` para hacer la conversión.

El apuntador principal será examinado en tiempo de ejecución. Si la conversión es correcta, su nuevo apuntador a `Pegaso` estará bien definido. Si la conversión es incorrecta, quiere decir que realmente no tenía un objeto `Pegaso`, y su nuevo apuntador será nulo. El listado 13.2 ejemplifica este punto.

**ENTRADA LISTADO 13.2 Conversión descendente**

```
1: // Listado 13.2 Uso de dynamic_cast.  
2: // Uso de rtti  
3:  
4: #include <iostream.h>  
5: enum TIPO { CABALLO, PEGASO };  
6:  
7: class Caballo  
8: {  
9: public:  
10:     virtual void Galopar(){ cout << "Galopando...\n"; }  
11:  
12: private:  
13:     int suEdad;  
14: };  
15:  
16: class Pegaso : public Caballo  
17: {  
18: public:  
19:  
20:     virtual void Volar() { cout <<  
21:         "¡Puedo volar! ¡Puedo volar! ¡Puedo volar!\n"; }  
21: };
```

**LISTADO 13.2** CONTINUACIÓN

---

```
22:  
23:     const int NumeroCaballos = 5;  
24:     int main()  
25:     {  
26:         Caballo* Rancho[NumeroCaballos];  
27:         Caballo* apCaballo;  
28:         int opcion,i;  
29:         for (i=0; i<NumeroCaballos; i++)  
30:         {  
31:             cout << "(1)Caballo (2)Pegaso: ";  
32:             cin >> opcion;  
33:             if (opcion == 2)  
34:                 apCaballo = new Pegaso;  
35:             else  
36:                 apCaballo = new Caballo;  
37:             Rancho[i] = apCaballo;  
38:         }  
39:         cout << "\n";  
40:         for (i=0; i<NumeroCaballos; i++)  
41:         {  
42:             Pegaso *apPeg = dynamic_cast< Pegaso *> (Rancho[i]);  
43:             if (apPeg)  
44:                 apPeg->Volar();  
45:             else  
46:                 cout << "Sólo es un caballo\n";  
47:  
48:             delete Rancho[i];  
49:         }  
50:     return 0;  
51: }
```

---

**SALIDA**

```
(1)Caballo (2)Pegaso: 1  
(1)Caballo (2)Pegaso: 2  
(1)Caballo (2)Pegaso: 1  
(1)Caballo (2)Pegaso: 2  
(1)Caballo (2)Pegaso: 1  
  
Sólo es un caballo  
¡Puedo volar! ¡Puedo volar! ¡Puedo volar!  
Sólo es un caballo  
¡Puedo volar! ¡Puedo volar! ¡Puedo volar!  
Sólo es un caballo
```

**Preguntas frecuentes**

**FAQ:** Al compilar, obtengo un error de mi compilador g++ de GNU (versión 2.7.2 o anterior):

```
1st13-02.cxx: In function 'int main()':  
1st13-02.cxx:42: cannot take typeid of object when -fRTTI is not specified  
1st13-02.cxx:42: invalid type argument  
1st13-02.cxx:42: failed to build type descriptor node of 'Pegaso',  
maybe typeinfo.h not included
```

**Respuesta:** Éste es uno de los mensajes de error más confusos del compilador GNU.

Desafortunadamente, la versión que usted utiliza tal vez no tiene soporte para RTTI, aún cuando sugiere formas de solucionar esto.

Puede probar volviendo a compilar con la opción -fRTTI, pero si recibe la advertencia de que no se reconoce la opción, entonces su plataforma y versión específicas no tienen soporte para esta capacidad.

La versión 2.9.5 soporta RTTI de manera predeterminada.

**ANÁLISIS**

Esta solución también funciona. `Volar()` se deja fuera de `Caballo`, y no se llama en objetos `Caballo`. Sin embargo, al llamar a este método en objetos `Pegaso`, éstos se deben convertir en forma explícita; los objetos `Caballo` no tienen el método `Volar()`, por esta razón, el apuntador nos debe indicar que está apuntando a un objeto `Pegaso` antes de usarlo.

La necesidad de convertir el objeto `Pegaso` es una advertencia de que algo puede estar mal en su diseño. Este programa, en efecto, mina el polimorfismo de funciones virtuales, ya que depende de la conversión del objeto al tipo indicado en tiempo de ejecución.

## Cómo agregar objetos a dos listas

El otro problema con estas soluciones es que como declaró `Pegaso` como un tipo de `Caballo`, no puede agregar un objeto `Pegaso` a una lista de `Aves`. Ya pagó el precio de pasar el método `Volar()` a la clase `Caballo` o de realizar una conversión descendente del apuntador, y aún así no obtiene toda la funcionalidad que necesita.

Una última solución con herencia simple se presenta a sí misma. Puede enviar a `Volar()`, `Relinchar()` y `Galopar()` hacia una clase base común para `Ave` y `Caballo`: `Animal`. Ahora, en lugar de tener una lista de `Aves` y una lista de `Caballos`, puede tener una lista unificada de `Animales`. Esto funciona, pero filtra más funcionalidad dentro de la clase base.

Como una alternativa, puede dejar los métodos en donde están y realizar la conversión descendente de los objetos Caballo, Ave y Pegaso, ¡pero eso es peor!

| DEBE                                                                                                                                                                                                                                          | NO DEBE                                                                                                                                                                                                      |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>DEBE</b> mover la funcionalidad hacia las capas superiores de la jerarquía de clases.</p> <p><b>DEBE</b> evitar el cambio del tipo de un objeto en tiempo de ejecución (utilice métodos virtuales, plantillas y herencia múltiple).</p> | <p><b>NO DEBE</b> realizar la conversión descendente de apunadores entre objetos base y objetos derivados.</p> <p><b>NO DEBE</b> mover la interfaz hacia las capas superiores de la jerarquía de clases.</p> |

## Herencia múltiple

Es posible衍生 una nueva clase de más de una clase base. Esto se conoce como herencia múltiple. Para衍生 de más clases aparte de la clase base, se separa cada clase base con comas en la designación de la clase. El listado 13.3 muestra cómo declarar una clase Pegaso para que se derive tanto de Caballo como de Ave. El programa luego agrega objetos Pegaso en ambos tipos de listas.

### ENTRADA

### LISTADO 13.3 Herencia múltiple

```

1: // Listado 13.3. Herencia múltiple.
2: // Herencia múltiple
3:
4: #include <iostream.h>
5:
6: class Caballo
7: {
8: public:
9:     Caballo() { cout << "Constructor de Caballo... "; }
10:    virtual ~Caballo() { cout << "Destructor de Caballo... "; }
11:    virtual void Relinchar() const { cout << "¡Yihii!... "; }
12: private:
13:     int suEdad;
14: };
15:
16: class Ave
17: {
18: public:
19:     Ave() { cout << "Constructor de Ave... "; }
20:     virtual ~Ave() { cout << "Destructor de Ave... "; }
21:     virtual void Gorjeear() const { cout << "Griiii... "; }
22:     virtual void Volar() const
23:     {
24:         cout << "¡Puedo volar! ¡Puedo volar! ¡Puedo volar! ";
25:     }

```

```
26:     private:
27:         int suPeso;
28:     };
29:
30: class Pegaso : public Caballo, public Ave
31: {
32: public:
33:     void Gorjear() const { Relinchar(); }
34:     Pegaso() { cout << "Constructor de Pegaso... " ; }
35:     ~Pegaso() { cout << "Destructor de Pegaso... " ; }
36: };
37:
38: const int NumeroMagico = 2;
39: int main()
40: {
41:     Caballo* Rancho[NumeroMagico];
42:     Ave* Pajarera[NumeroMagico];
43:     Caballo * apCaballo;
44:     Ave * apAve;
45:     int opcion,i;
46:     for (i=0; i<NumeroMagico; i++)
47:     {
48:         cout << "\n(1)Caballo (2)Pegaso: ";
49:         cin >> opcion;
50:         if (opcion == 2)
51:             apCaballo = new Pegaso;
52:         else
53:             apCaballo = new Caballo;
54:         Rancho[i] = apCaballo;
55:     }
56:     for (i=0; i<NumeroMagico; i++)
57:     {
58:         cout << "\n(1)Ave (2)Pegaso: ";
59:         cin >> opcion;
60:         if (opcion == 2)
61:             apAve = new Pegaso;
62:         else
63:             apAve = new Ave;
64:         Pajarera[i] = apAve;
65:     }
66:
67:     cout << "\n";
68:     for (i=0; i<NumeroMagico; i++)
69:     {
70:         cout << "\nRancho[" << i << "]: " ;
71:         Rancho[i]->Relinchar();
72:         delete Rancho[i];
73:     }
74:
75:     for (i=0; i<NumeroMagico; i++)
76:     {
77:         cout << "\nPajarera[" << i << "]: " ;
78:         Pajarera[i]->Gorjear();
79:         Pajarera[i]->Volar();
```

13

**LISTADO 13.3** CONTINUACIÓN

```

80:           delete Pajarera[i];
81:       }
82:   return 0;
83: }
```

**SALIDA**

```

(1)Caballo (2)Pegaso: 1
Constructor de Caballo...
(1)Caballo (2)Pegaso: 2
Constructor de Caballo... Constructor de Ave... Constructor de
Pegaso...
(1)Ave (2)Pegaso: 1
Constructor de Ave...
(1)Ave (2)Pegaso: 2
Constructor de Caballo... Constructor de Ave... Constructor de
Pegaso...

Rancho[0]: ¡Yihii!... Destructor de Caballo...
Rancho[1]: ¡Yihii!... Destructor de Pegaso... Destructor de Ave...
Destructor de Caballo...
Pajarera[0]: Griii... ¡Puedo volar! ¡Puedo volar! ¡Puedo volar!
Destructor de Ave...
Pajarera[1]: ¡Yihii!... ¡Puedo volar! ¡Puedo volar! ¡Puedo volar!
Destructor de Pegaso... Destructor de Ave... Destructor de Caballo...
```

**ANÁLISIS**

En las líneas 6 a 14 se declara la clase **Caballo**. El constructor y el destructor imprimen un mensaje, y el método **Relinchar()** imprime la palabra (mejor dicho, la voz onomatopéyica) ¡**Yihii!**!

En las líneas 16 a 28 se declara la clase **Ave**. Además de su constructor y su destructor, esta clase tiene dos métodos: **Gorjeear()** y **Volar()**, los cuales imprimen mensajes de identificación. En un programa real, éstos podrían, por ejemplo, activar la bocina o generar imágenes animadas.

Finalmente, en las líneas 30 a 36 se declara la clase **Pegaso**. Se deriva de **Caballo** y de **Ave**. La clase **Pegaso** redefine el método **Gorjeear()** para llamar al método **Relinchar()**, el cual hereda de **Caballo**.

Se crean dos listas: una llamada **Rancho** con apunadores a **Caballo** en la línea 41, y una llamada **Pajarera** con apunadores a **Ave** en la línea 42. En las líneas 46 a 55 se agregan objetos **Caballo** y **Pegaso** a la lista **Rancho**. En las líneas 56 a 65 se agregan objetos **Ave** y **Pegaso** a la lista **Pajarera**.

Las llamadas a los métodos virtuales tanto en los apunadores a **Ave** como en los apunadores a **Caballo** hacen lo correcto para los objetos **Pegaso**. Por ejemplo, en la línea 78 los métodos del arreglo **Pajarera** se utilizan para llamar a **Gorjeear()** en los objetos a los que apuntan. La clase **Ave** declara este método como virtual, por lo que se llama a la función apropiada para cada objeto.

Observe que cada vez que se crea un objeto Pegaso, la salida refleja que también se crean tanto la parte Ave como la parte Caballo de dicho objeto. Cuando se destruye un objeto Pegaso, también se destruyen la parte Ave y la parte Caballo, gracias a que los destructores se hicieron virtuales.

#### Declaración de la herencia múltiple

Un objeto que se herede de más de una clase se declara enlistando las clases base después del signo de dos puntos (:) que va después del nombre de la clase. Las clases base se separan por medio de comas.

##### Ejemplo 1

```
class Pegaso : public Caballo, public Ave
```

##### Ejemplo 2

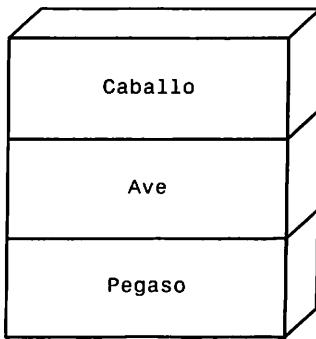
```
class Schnoodle : public Schnauzer, public Poodle
```

## Las partes de un objeto con herencia múltiple

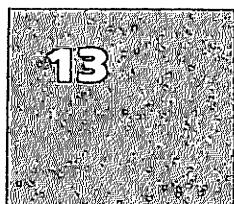
Cuando se crea el objeto Pegaso en memoria, ambas clases base forman parte del objeto Pegaso, como se muestra en la figura 13.1.

FIGURA 13.1

Objetos con herencia múltiple.



Surgen varias cuestiones con los objetos que tienen clases base múltiples. Por ejemplo, ¿qué pasa si dos clases base que por casualidad tienen el mismo nombre, tienen funciones o datos virtuales? ¿Cómo se inicializan los constructores de clases base múltiples? ¿Qué ocurre si ambas clases base múltiples se derivan de la misma clase? Las siguientes secciones responderán estas preguntas y explicarán la forma en que se puede utilizar la herencia múltiple.



## Constructores en objetos con herencia múltiple

Si Pegaso se deriva tanto de Caballo como de Ave, y cada una de las clases base tiene constructores que llevan parámetros, la clase Pegaso inicializa estos constructores uno por uno. El listado 13.4 muestra cómo se hace esto.

**ENTRADA****LISTADO 13.4** Cómo llamar varios constructores

```
1: // Listado 13.4
2: // Cómo llamar varios constructores
3: #include <iostream.h>
4: typedef int CUARTAS;
5: enum COLOR { Rojo, Verde, Azul, Amarillo, Blanco, Negro, Cafe } ;
6:
7: class Caballo
8: {
9: public:
10:    Caballo(COLOR color, CUARTAS altura);
11:    virtual ~Caballo() { cout << "Destructor de Caballo...\n"; }
12:    virtual void Relinchar()const { cout << "¡Yihii!... "; }
13:    virtual CUARTAS ObtenerAltura() const { return suAltura; }
14:    virtual COLOR ObtenerColor() const { return suColor; }
15: private:
16:    CUARTAS suAltura;
17:    COLOR suColor;
18: };
19:
20: Caballo::Caballo(COLOR color, CUARTAS altura):
21:    suColor(color),suAltura(altura)
22: {
23:    cout << "Constructor de Caballo...\n";
24: }
25:
26: class Ave
27: {
28: public:
29:    Ave(COLOR color, bool emigra);
30:    virtual ~Ave() {cout << "Destructor de Ave...\n"; }
31:    virtual void Gorjeear()const { cout << "Griiii... "; }
32:    virtual void Volar()const
33:    {
34:        cout << "¡Puedo volar! ¡Puedo volar! ¡Puedo volar! ";
35:    }
36:    virtual COLOR ObtenerColor()const { return suColor; }
37:    virtual bool ObtenerMigracion() const { return suMigracion; }
38:
39: private:
40:    COLOR suColor;
41:    bool suMigracion;
42: };
43:
```

```
44:     Ave::Ave(COLOR color, bool emigra):
45:         suColor(color), suMigracion(emigra)
46:     {
47:         cout << "Constructor de Ave...\n";
48:     }
49:
50: class Pegaso : public Caballo, public Ave
51: {
52: public:
53:     void Gorjear()const { Relinchar(); }
54:     Pegaso(COLOR, CUARTAS, bool, long);
55:     ~Pegaso() {cout << "Destructor de Pegaso...\n";}
56:     virtual long ObtenerNumeroCreyentes() const
57:     {
58:         return suNumeroCreyentes;
59:     }
60:
61: private:
62:     long suNumeroCreyentes;
63: };
64:
65: Pegaso::Pegaso(
66:     COLOR aColor,
67:     CUARTAS altura,
68:     bool emigra,
69:     long NumCreyen):
70: Caballo(aColor, altura),
71: Ave(aColor, emigra),
72: suNumeroCreyentes(NumCreyen)
73: {
74:     cout << "Constructor de Pegaso...\n";
75: }
76:
77: int main()
78: {
79:     Pegaso *apPeg = new Pegaso(Rojo, 5, true, 10);
80:     apPeg->Volar();
81:     apPeg->Relinchar();
82:     cout << "\nSu Pegaso mide " << apPeg->ObtenerAltura();
83:     cout << " cuartas de altura y ";
84:     if (apPeg->ObtenerMigracion())
85:         cout << "si emigra.";
86:     else
87:         cout << "no emigra.";
88:     cout << "\nUn total de " << apPeg->ObtenerNumeroCreyentes();
89:     cout << " personas creen que si existe.\n";
90:     delete apPeg;
91:     return 0;
92: }
```

**SALIDA**

```

Constructor de Caballo...
Constructor de Ave...
Constructor de Pegaso...
¡Puedo volar! ¡Puedo volar! ¡Puedo volar! ¡Yihii!...
Su Pegaso mide 5 cuartas de altura y si emigra.
Un total de 10 personas creen que si existe.

Destructor de Pegaso...
Destructor de Ave...
Destructor de Caballo...

```

**ANÁLISIS**

En las líneas 7 a 18 se declara la clase `Caballo`. El constructor lleva dos parámetros: uno es una enumeración que se declara en la línea 5, y el otro es un `typedef` (definición de tipo) que se declara en la línea 4. La implementación del constructor en las líneas 20 a 24 simplemente inicializa las variables miembro e imprime un mensaje.

En las líneas 26 a 42 se declara la clase `Ave`, y la implementación de su constructor se encuentra en las líneas 44 a 48. De nuevo, la clase `Ave` lleva dos parámetros. Curiosamente, el constructor de `Caballo` toma colores (para que usted pueda detectar caballos de diferentes colores), y el constructor de `Ave` toma el color de las plumas (para que los que tienen las mismas plumas puedan mantenerse juntos). Esto conduce a un problema cuando quiere preguntar al `Pegaso` cuál es su color, lo que verá en el siguiente ejemplo.

La clase `Pegaso` en sí se declara en las líneas 50 a 63, y su constructor se encuentra en las líneas 65 a 75. La inicialización del objeto `Pegaso` incluye tres instrucciones. Primero, el constructor de `Caballo` se inicializa con color y altura. Luego, el constructor de `Ave` se inicializa con color y un valor booleano. Finalmente, se inicializa la variable miembro `suNumeroCreyentes` del objeto `Pegaso`. Después de hacer todo eso, se llama al cuerpo del constructor de `Pegaso`.

En la función `main()` se crea un apuntador a `Pegaso` y se utiliza para tener acceso a los métodos de los objetos base.

## Resolución de ambigüedad

En el listado 13.4, tanto la clase `Caballo` como la clase `Ave` tienen un método llamado `ObtenerColor()`. Tal vez necesite pedir al objeto `Pegaso` que regrese su color, pero hay un problema: La clase `Pegaso` hereda tanto de `Ave` como de `Caballo`. Ambos tienen un color, y sus métodos para obtener ese color tienen los mismos nombres y las mismas firmas. Esto crea una ambigüedad para el compilador, que usted debe resolver.

Si simplemente escribe

```
COLOR colorActual = apPeg->ObtenerColor();
```

g++ producirá este error de compilación:

```
request for method 'ObtenerColor' is ambiguous
```

Puede resolver la ambigüedad con una llamada explícita al método que quiere invocar:

```
COLOR colorActual = apPeg->Caballo::ObtenerColor();
```

Siempre que necesite resolver de qué clase heredar una función miembro o datos miembro, puede identificar plenamente la llamada anteponiendo el nombre de la clase a los datos o a la función de la clase base.

Observe que si Pegaso fuera a redefinir esta función, el problema se debe resolver en el método de Pegaso:

```
virtual COLOR ObtenerColor()const { return Caballo::ObtenerColor(); }
```

Esto oculta el problema de los clientes de la clase Pegaso y encapsula dentro de Pegaso el conocimiento de cuál va a ser la clase base de la que quiere heredar su color. El cliente aún tiene la libertad de forzar esta cuestión escribiendo

```
COLOR colorActual = apPeg->Ave::ObtenerColor();
```

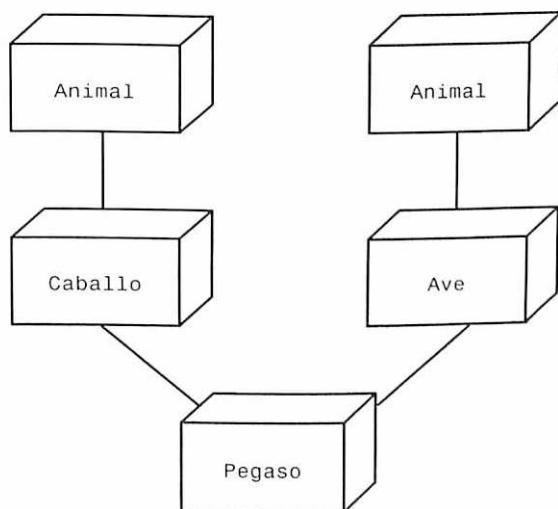
## Herencia de una clase base compartida

¿Qué pasa si tanto Ave como Caballo heredan de una clase base común, como Animal?

La figura 13.2 muestra cómo se ve esto.

**FIGURA 13.2**

*Clases base comunes.*

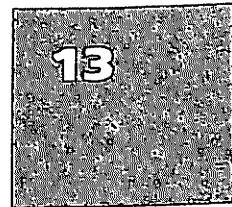


Como puede ver en la figura 13.2, existen dos objetos de la clase base. Cuando se llama a una función o a un dato miembro en la clase base compartida, existe otra ambigüedad. Por ejemplo, si Animal declara a suEdad como variable miembro y a ObtenerEdad() como un método, y usted llama a apPeg->ObtenerEdad(), ¿quiso llamar a la función ObtenerEdad() que heredó de Animal por medio de Caballo, o por medio de Ave? Debe resolver también esta ambigüedad, como se muestra en el listado 13.5.

**ENTRADA****LISTADO 13.5 Clases base comunes**

```
1:  // Listado 13.5
2:  // Clases base comunes
3:  #include <iostream.h>
4:
5:  typedef int CUARTAS;
6:  enum COLOR { Rojo, Verde, Azul, Amarillo, Blanco, Negro, Cafe } ;
7:
8:  class Animal           // base común para caballo y ave
9: {
10: public:
11:     Animal(int);
12:     virtual ~Animal() { cout << "Destructor de Animal...\n"; }
13:     virtual int ObtenerEdad() const { return suEdad; }
14:     virtual void AsignarEdad(int edad) { suEdad = edad; }
15: private:
16:     int suEdad;
17: };
18:
19: Animal::Animal(int edad):
20: suEdad(edad)
21: {
22:     cout << "Constructor de Animal...\n";
23: }
24:
25: class Caballo : public Animal
26: {
27: public:
28:     Caballo(COLOR color, CUARTAS altura, int edad);
29:     virtual ~Caballo() { cout << "Destructor de Caballo...\n"; }
30:     virtual void Relinchar()const { cout << "¡Yihii!... "; }
31:     virtual CUARTAS ObtenerAltura() const { return suAltura; }
32:     virtual COLOR ObtenerColor() const { return suColor; }
33: protected:
34:     CUARTAS suAltura;
35:     COLOR suColor;
36: };
37:
38: Caballo::Caballo(COLOR color, CUARTAS altura, int edad):
39:     Animal(edad),
```

```
40:     suColor(color),suAltura(altura)
41: {
42:     cout << "Constructor de Caballo...\n";
43: }
44:
45: class Ave : public Animal
46: {
47: public:
48:     Ave(COLOR color, bool migra, int edad);
49:     virtual ~Ave() {cout << "Destructor de Ave...\n"; }
50:     virtual void Gorjear()const { cout << "Griii... "; }
51:     virtual void Volar()const
52:         { cout << "¡Puedo volar! ¡Puedo volar! ¡Puedo volar! "; }
53:     virtual COLOR ObtenerColor()const { return suColor; }
54:     virtual bool ObtenerMigracion() const { return suMigracion; }
55: protected:
56:     COLOR suColor;
57:     bool suMigracion;
58: };
59:
60: Ave::Ave(COLOR color, bool emigra, int edad):
61:     Animal(edad),
62:     suColor(color), suMigracion(emigra)
63: {
64:     cout << "Constructor de Ave...\n";
65: }
66:
67: class Pegaso : public Caballo, public Ave
68: {
69: public:
70:     void Gorjear()const { Relinchar(); }
71:     Pegaso(COLOR, CUARTAS, bool, long, int);
72:     virtual ~Pegaso() {cout << "Destructor de Pegaso...\n";}
73:     virtual long ObtenerNumeroCreyentes() const
74:         { return suNumeroCreyentes; }
75:     virtual COLOR ObtenerColor()const { return Caballo::suColor; }
76:     virtual int ObtenerEdad() const { return Caballo::ObtenerEdad(); }
77: private:
78:     long suNumeroCreyentes;
79: };
80:
81: Pegaso::Pegaso(
82:     COLOR aColor,
83:     CUARTAS altura,
84:     bool emigra,
85:     long NumCreyen,
86:     int edad):
87:     Caballo(aColor, altura,edad),
88:     Ave(aColor, emigra,edad),
89:     suNumeroCreyentes(NumCreyen)
90: {
```



**LISTADO 13.5** CONTINUACIÓN

```
91:     cout << "Constructor de Pegaso...\n";
92: }
93:
94: int main()
95: {
96:     Pegaso *apPeg = new Pegaso(Rojo, 5, true, 10, 2);
97:     int edad = apPeg->ObtenerEdad();
98:     cout << "Este Pegaso tiene " << edad << " años de edad.\n";
99:     delete apPeg;
100:    return 0;
101: }
```

**SALIDA**

```
Constructor de Animal...
Constructor de Caballo...
Constructor de Animal...
Constructor de Ave...
Constructor de Pegaso...
Este Pegaso tiene 2 años de edad.
Destructor de Pegaso...
Destructor de Ave...
Destructor de Animal...
Destructor de Caballo...
Destructor de Animal...
```

**ANÁLISIS**

Hay varias características interesantes en este listado. La clase `Animal` se declara en las líneas 8 a 17. `Animal` agrega una variable miembro llamada `suEdad` y dos métodos de acceso: `ObtenerEdad()` y `AsignarEdad()`.

En la línea 25 se declara la clase `Caballo` derivada de `Animal`. El constructor de `Caballo` ahora tiene un tercer parámetro, `edad`, mismo que pasa a su clase base, `Animal`. Observe que la clase `Caballo` no redefine a `ObtenerEdad()`; simplemente la hereda.

En la línea 45 se declara la clase `Ave` derivada de `Animal`. Su constructor también toma una edad y la utiliza para inicializar la clase base, `Animal`. También hereda `ObtenerEdad()` sin redefinirla.

`Pegaso` hereda tanto de `Ave` como de `Animal`, por lo que ahora tiene dos clases `Animal` en su cadena hereditaria. Si se fuera a llamar a `ObtenerEdad()` en un objeto `Pegaso`, habría que resolver la ambigüedad, o identificar completamente el método que se quiere si `Pegaso` no redefinió el método.

Esto se resuelve en la línea 76 cuando el objeto `Pegaso` redefine a `ObtenerEdad()` para que no haga nada más que una *cadena ascendente*, es decir, llamar al mismo método de una clase base.

La cadena ascendente se hace por dos razones: ya sea para resolver la ambigüedad sobre cuál clase base llamar, como en este caso, o para hacer algo y luego dejar que la función de la clase base haga algo más. A veces, tal vez quiera trabajar y luego hacer una cadena ascendente, o hacer la cadena y luego hacer el trabajo cuando regrese la función de la clase base.

El constructor de **Pegaso** toma cinco parámetros: el color de la criatura, su altura (se mide en cuartas), si emigra o no, cuántas personas creen que existe, y su edad. En la línea 87, el constructor inicializa la parte **Caballo** del objeto **Pegaso** con el color, la altura y la edad. En la línea 88 inicializa la parte **Ave** con el color, si emigra o no, y la edad. Por último, en la línea 89 inicializa la variable **suNúmeroCreyentes**.

En la línea 87, la llamada al constructor de **Caballo** invoca a la implementación que se muestra en la línea 38. El constructor de **Caballo** utiliza el parámetro **edad** para inicializar la parte **Animal** de la parte **Caballo** del objeto **Pegaso**. Luego inicializa las dos variables miembro de **Caballo**: **suColor** y **suAltura**.

En la línea 88, la llamada al constructor de **Ave** invoca la implementación que se muestra en la línea 60. Aquí también se utiliza el parámetro **edad** para inicializar la parte **Animal** de **Ave**.

Observe que el parámetro **aColor** de **Pegaso** se utiliza para inicializar las propiedades tanto de **Ave** como de **Caballo**. Observe también que **edad** se utiliza para inicializar la variable **suEdad** de la clase base **Animal** de **Caballo** y de la clase base **Animal** de **Ave**.

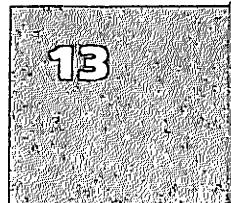
## Herencia virtual

En el listado 13.5, la clase **Pegaso** hizo un esfuerzo considerable para resolver la ambigüedad acerca de cuál de sus clases base **Animal** quería invocar. La mayoría de las veces, la decisión sobre cuál clase base utilizar es arbitraria (después de todo, la clase **Caballo** y la clase **Ave** tienen la misma clase base).

Usted puede decirle a C++ que no quiere dos copias de la clase base compartida, como se muestra en la figura 13.2, sino que mejor quisiera tener una sola clase base compartida, como se muestra en la figura 13.3.

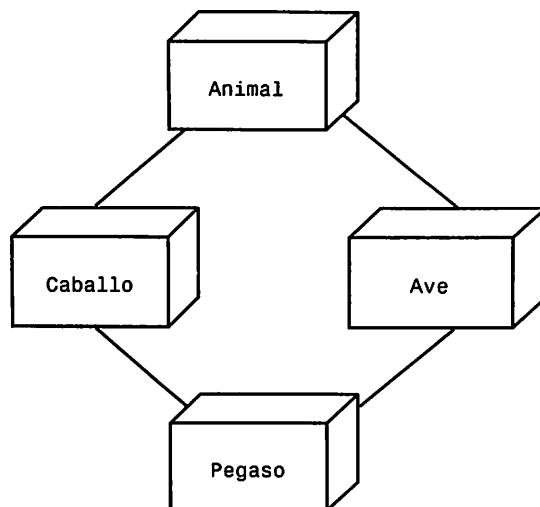
Puede lograr esto haciendo que **Animal** sea una clase base virtual tanto de **Caballo** como de **Ave**. La clase **Animal** no cambia en nada. Las clases **Caballo** y **Ave** cambian sólo en el uso del término **virtual** en sus declaraciones. Sin embargo, **Pegaso** cambia considerablemente.

Por lo general, el constructor de una clase inicializa sólo sus propias variables y su clase base. No obstante, las clases base heredadas en forma virtual son una excepción. Las inicializa su clase más derivada. Por lo tanto, a **Animal** no la inicializa **Caballo** ni **Ave**, sino **Pegaso**. **Caballo** y **Ave** tienen que inicializar a **Animal** en sus constructores, pero estas inicializaciones serán ignoradas cuando se cree un objeto **Pegaso**.



**FIGURA 13.3**

*Una herencia de diamante.*



El listado 13.6 modifica el listado 13.5 para aprovechar la herencia virtual.

**ENTRADA****LISTADO 13.6** Muestra del uso de la herencia virtual

```
1:  // Listado 13.6
2:  // Herencia virtual
3:  #include <iostream.h>
4:
5:  typedef int CUARTAS;
6:  enum COLOR { Rojo, Verde, Azul, Amarillo, Blanco, Negro, Cafe } ;
7:
8:  class Animal      // base común para caballo y ave
9:  {
10:  public:
11:    Animal(int);
12:    virtual ~Animal() { cout << "Destructor de Animal...\n"; }
13:    virtual int ObtenerEdad() const { return suEdad; }
14:    virtual void AsignarEdad(int edad) { suEdad = edad; }
15:  private:
16:    int suEdad;
17:  };
18:
19:  Animal::Animal(int edad):
20:    suEdad(edad)
21:  {
22:    cout << "Constructor de Animal...\n";
23:  }
24:
25:  class Caballo : virtual public Animal
26:  {
```

```
27:     public:
28:         Caballo(COLOR color, CUARTAS altura, int edad);
29:         virtual ~Caballo() { cout << "Destructor de Caballo...\n"; }
30:         virtual void Relinchar()const { cout << "¡Yihii!... "; }
31:         virtual CUARTAS ObtenerAltura() const { return suAltura; }
32:         virtual COLOR ObtenerColor() const { return suColor; }
33:     protected:
34:         CUARTAS suAltura;
35:         COLOR suColor;
36:     };
37:
38:     Caballo::Caballo(COLOR color, CUARTAS altura, int edad):
39:         Animal(edad),
40:         suColor(color),suAltura(altura)
41:     {
42:         cout << "Constructor de Caballo...\n";
43:     }
44:
45:     class Ave : virtual public Animal
46:     {
47:     public:
48:         Ave(COLOR color, bool emigra, int edad);
49:         virtual ~Ave() {cout << "Destructor de Ave...\n"; }
50:         virtual void Gorjear()const { cout << "Griiii... "; }
51:         virtual void Volar()const
52:             { cout << "¡Puedo volar! ¡Puedo volar! ¡Puedo volar! "; }
53:         virtual COLOR ObtenerColor()const { return suColor; }
54:         virtual bool ObtenerMigracion() const { return suMigracion; }
55:     protected:
56:         COLOR suColor;
57:         bool suMigracion;
58:     };
59:
60:     Ave::Ave(COLOR color, bool emigra, int edad):
61:         Animal(edad),
62:         suColor(color), suMigracion(emigra)
63:     {
64:         cout << "Constructor de Ave...\n";
65:     }
66:
67:     class Pegaso : public Caballo, public Ave
68:     {
69:     public:
70:         void Gorjear()const { Relinchar(); }
71:         Pegaso(COLOR, CUARTAS, bool, long, int);
72:         virtual ~Pegaso() {cout << "Destructor de Pegaso...\n"; }
73:         virtual long ObtenerNumeroCreyentes() const
74:             { return suNumeroCreyentes; }
75:         virtual COLOR ObtenerColor()const { return Caballo::suColor; }
76:     private:
77:         long suNumeroCreyentes;
```

13

**LISTADO 13.6** CONTINUACIÓN

```
78:     };
79:
80:     Pegaso::Pegaso(
81:         COLOR aColor,
82:         CUARTAS altura,
83:         bool emigra,
84:         long NumCreyen,
85:         int edad):
86:     Caballo(aColor, altura, edad),
87:     Ave(aColor, emigra, edad),
88:     Animal(edad*2),
89:     suNumeroCreyentes(NumCreyen)
90: {
91:     cout << "Constructor de Pegaso...\n";
92: }
93:
94: int main()
95: {
96:     Pegaso *apPeg = new Pegaso(Rojo, 5, true, 10, 2);
97:     int edad = apPeg->ObtenerEdad();
98:     cout << "Este Pegaso tiene " << edad << " años de edad.\n";
99:     delete apPeg;
100:    return 0;
101: }
```

**SALIDA**

```
Constructor de Animal...
Constructor de Caballo...
Constructor de Ave...
Constructor de Pegaso...
Este Pegaso tiene 4 años de edad.
Destructor de Pegaso...
Destructor de Ave...
Destructor de Caballo...
Destructor de Animal...
```

**ANÁLISIS**

En la línea 25, Caballo declara que tiene herencia virtual de Animal, y en la línea 45 Ave hace la misma declaración. Observe que los constructores tanto de Ave como de Animal aún inicializan el objeto Animal.

Pegaso hereda tanto de Ave como de Animal, y siendo el objeto más derivado de Animal, también inicializa a Animal. Sin embargo, la inicialización de Pegaso es la que se llama, y las llamadas al constructor de Animal de Ave y de Caballo se ignoran. Puede ver esto debido a que se pasa el valor 2, y Caballo y Ave se lo pasan a Animal, pero Pegaso lo duplica. El resultado, 4, se refleja en la impresión de la línea 98 y como se muestra en la salida.

Pegaso ya no tiene que resolver la ambigüedad en la llamada a `ObtenerEdad()`, por lo que tiene la libertad de heredar simplemente esta función de `Animal`. Observe que Pegaso aún debe resolver la ambigüedad en la llamada a `ObtenerColor()` debido a que esta función se encuentra en sus dos clases base y no en `Animal`.

#### Declaración de clases para herencia virtual

Para asegurar que las clases derivadas tengan sólo una instancia de clases base comunes, declare las clases intermedias de forma que hereden virtualmente de la clase base.

##### Ejemplo 1

```
class Caballo : virtual public Animal  
class Ave : virtual public Animal  
class Pegaso : public Caballo, public Ave
```

##### Ejemplo 2

```
class Schnauzer : virtual public Perro  
class Poodle : virtual public Perro  
class Schnoodle : public Schnauzer, public Poodle
```

## Problemas con la herencia múltiple

Aunque la herencia múltiple ofrece varias ventajas sobre la herencia simple, muchos programadores de C++ se muestran renuentes a usarla. Los problemas que citan son que muchos compiladores aún no la soportan, que dificulta la depuración, y que casi todo lo que se puede hacer con la herencia múltiple se puede hacer sin ella.

Estos son puntos válidos, y usted debe estar en contra de implementar programas complejos si no es necesario. Algunos depuradores tienen muchas dificultades con la herencia múltiple, y algunos diseños se hacen complejos al utilizar herencia múltiple cuando no se necesita.

| DEBE                                                                                                                                 | NO DEBE                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| <b>DEBE</b> utilizar herencia múltiple cuando una clase nueva necesite funciones y características de más de una clase base.         | <b>NO DEBE</b> utilizar herencia múltiple cuando baste con la herencia simple. |
| <b>DEBE</b> utilizar la herencia virtual cuando las clases más derivadas deban tener sólo una instancia de la clase base compartida. |                                                                                |
| <b>DEBE</b> inicializar la clase base compartida desde la clase más derivada cuando utilice clases base virtuales.                   |                                                                                |

## Mezclas y clases de capacidad

Una forma de llegar a un término medio entre herencia múltiple y herencia simple es utilizar lo que se conoce como mezclas. Por ejemplo, podría hacer que su clase `Caballo` se derive de `Animal` y de `Desplegable`. `Desplegable` sólo agregaría unos cuantos métodos para desplegar cualquier objeto en pantalla.

Una *mezcla*, o clase *de capacidad*, es una clase que agrega funcionalidad sin agregar muchas o ninguna propiedad.

Las clases de capacidad se mezclan en una clase derivada de la misma forma en que se podría mezclar cualquier otra clase: declarando a la clase derivada para que herede de las clases de capacidad en forma pública. La única diferencia entre una clase de capacidad y cualquier otra clase es que la primera tiene pocas o ninguna propiedad. Ésta es una distinción arbitraria, y es sólo una manera abreviada de indicar que a veces todo lo que se quiere hacer es mezclar algunas capacidades adicionales sin complicar la clase derivada.

Esto hará que para algunos depuradores sea más sencillo trabajar con mezclas que con objetos con herencia múltiple más complejos. Además, existe una menor probabilidad de ambigüedad al tener acceso a las propiedades en la otra clase base principal.

Por ejemplo, si `Caballo` se derivara de `Animal` y de `Desplegable`, `Desplegable` no tendría propiedades. `Animal` no tendría cambio, por lo que todas las propiedades de `Caballo` se derivarían de `Animal`, pero sus funciones se derivarían tanto de `Animal` como de `Desplegable`.

El término mezcla viene de una tienda de helados de Somerville, Massachusetts, en donde se mezclaban dulces y pasteles en los sabores básicos de la nieve. Esto pareció una buena metáfora para algunos de los programadores que hacían programas orientados a objetos, quienes solían veranear ahí, especialmente cuando trabajaban con el lenguaje de programación orientada a objetos llamado SCOOPS (en inglés, este término se utiliza para la cuchara con la que se sirve la nieve).

## Tipos de datos abstractos

Con frecuencia, usted creará una jerarquía de clases en conjunto. Por ejemplo, podría crear una clase `Figura` y derivar de ésta las clases `Rectangulo` y `Circulo`. De `Rectangulo`, podría derivar a `Cuadrado` como un caso especial de `Rectangulo`.

Cada una de las clases derivadas redefinirá el método `Dibujar()`, el método `ObtenerArea()`, y así sucesivamente. El listado 13.7 muestra una implementación simple de la clase `Figura` y de sus clases derivadas `Circulo` y `Rectangulo`.

**ENTRADA****LISTADO 13.7 Clases de Figura**

```
1: //Listado 13.7. Clases de Figura.
2:
3: #include <iostream.h>
4:
5:
6: class Figura
7: {
8: public:
9:     Figura(){}
10:    virtual ~Figura(){}
11:    virtual long ObtenerArea() { return -1; }
12:    virtual long ObtenerPerim() { return -1; }
13:    virtual void Dibujar() {}
14: private:
15: };
16:
17: class Circulo : public Figura
18: {
19: public:
20:     Circulo(int radio):suRadio(radio){}
21:     ~Circulo(){}
22:     long ObtenerArea() { return 3 * suRadio * suRadio; }
23:     long ObtenerPerim() { return 6 * suRadio; }
24:     void Dibujar();
25: private:
26:     int suRadio;
27:     int suCircunferencia;
28: };
29:
30: void Circulo::Dibujar()
31: {
32:     cout << "¡Aqui va la rutina para dibujar un Circulo!\n";
33: }
34:
35:
36: class Rectangulo : public Figura
37: {
38: public:
39:     Rectangulo(int longitud, int ancho):
40:         suLongitud(longitud), suAncho(ancho){}
41:     virtual ~Rectangulo(){}
42:     virtual long ObtenerArea() { return suLongitud * suAncho; }
43:     virtual long ObtenerPerim() { return 2*suLongitud + 2*suAncho; }
44:     virtual int ObtenerLongitud() { return suLongitud; }
45:     virtual int ObtenerAncho() { return suAncho; }
46:     virtual void Dibujar();
47: private:
48:     int suAncho;
```

13

**LISTADO 13.7** CONTINUACIÓN

```
49:         int suLongitud;
50:     };
51:
52: void Rectangulo::Dibujar()
53: {
54:     for (int i = 0; i<suLongitud; i++)
55:     {
56:         for (int j = 0; j<suAncho; j++)
57:             cout << "x ";
58:
59:         cout << "\n";
60:     }
61: }
62:
63: class Cuadrado : public Rectangulo
64: {
65: public:
66:     Cuadrado(int longitud);
67:     Cuadrado(int longitud, int ancho);
68:     ~Cuadrado(){}
69:     long ObtenerPerim() {return 4 * ObtenerLongitud();}
70: };
71:
72: Cuadrado::Cuadrado(int longitud):
73:     Rectangulo(longitud,longitud)
74: {}
75:
76: Cuadrado::Cuadrado(int longitud, int ancho):
77:     Rectangulo(longitud,ancho)
78:
79: {
80:     if (ObtenerLongitud() != ObtenerAncho())
81:         cout << "Error, no es un Cuadrado... ¿un Rectangulo??\n";
82: }
83:
84: int main()
85: {
86:     int opcion;
87:     bool fSalir = false;
88:     Figura * sp;
89:
90:     while (! fSalir)
91:     {
92:         cout << "(1)Circulo (2)Rectangulo (3)Cuadrado (0)Salir: ";
93:         cin >> opcion;
94:
95:         switch (opcion)
96:         {
```

```

97:         case 0: fSalir = true;
98:         break;
99:         case 1: sp = new Circulo(5);
100:        break;
101:        case 2: sp = new Rectangulo(4,6);
102:        break;
103:        case 3: sp = new Cuadrado(5);
104:        break;
105:        default: cout << "Escriba un número entre 0 y 3"
106:             << endl;
107:             continue;
108:             break;
109:         }
110:         if(fSalir) break;
111:         sp->Dibujar();
112:         delete sp;
113:         cout << "\n";
114:     }
115: }
```

**SALIDA**

```

(1)Circulo (2)Rectangulo (3)Cuadrado (0)Salir: 2
X X X X X X
X X X X X X
X X X X X X
X X X X X X

(1)Circulo (2)Rectangulo (3)Cuadrado (0)Salir:3
X X X X X
X X X X X
X X X X X
X X X X X
X X X X X

(1)Circulo (2)Rectangulo (3)Cuadrado (0)Salir:0
```

**ANÁLISIS**

En las líneas 6 a 15 se declara la clase `Figura`. Los métodos `ObtenerArea()` y `ObtenerPerim()` regresan un valor de error, y `Dibujar()` no realiza ninguna acción.

Después de todo, ¿qué significa dibujar una `Figura`? Sólo se pueden dibujar tipos de figuras (círculos, rectángulos, etc.); Como es una abstracción, no se puede dibujar una `Figura`.

13

`Circulo` se deriva de `Figura` y redefine los tres métodos virtuales. Observe que no hay motivo para agregar la palabra “virtual”, ya que es parte de su herencia. Pero no hay nada malo en hacerlo, como se muestra en la clase `Rectangulo` en las líneas 42, 43, y 46. Es una buena idea incluir el término `virtual` como recordatorio, o una forma de documentación.

`Cuadrado` se deriva de `Rectangulo`, y también redefine el método `ObtenerPerim()`, heredando el resto de los métodos definidos en `Rectangulo`.

Es preocupante que un cliente trate de instanciar un objeto `Figura`, y lo mejor sería hacer eso imposible. La clase `Figura` existe sólo para proporcionar una interfaz para las clases que se derivan de ella; como tal, es un tipo de datos abstracto, o ADT (Abstract Data Types).

Un tipo de datos abstracto representa un concepto (como el de figura) en vez de un objeto (como un círculo). En C++, un ADT siempre es la clase base para otras clases, y no es válido crear una instancia de un ADT.

## Funciones virtuales puras

C++ soporta la creación de tipos de datos abstractos con funciones virtuales puras. Una función virtual se convierte en pura inicializándola con cero, como en

```
virtual void Dibujar() = 0;
```

Cualquier clase que tenga una o más funciones virtuales puras es un ADT, y es ilegal instanciar un objeto de una clase que sea ADT. Si se trata de hacer esto se producirá un error en tiempo de compilación. Al colocar una función virtual pura en su clase, les está indicando dos cosas a los clientes de su clase:

- No crear un objeto de esta clase; hacer derivaciones de ella.
- Asegurarse de redefinir la función virtual pura.

Cualquier clase que se derive de un ADT hereda la función virtual pura como pura, por lo que debe redefinir cada función virtual pura si quiere crear instancias de objetos. Por ejemplo, si `Rectangulo` hereda de `Figura`, y `Figura` tiene tres funciones virtuales puras, `Rectangulo` debe redefinir esas tres funciones; si no lo hace, será también un ADT. El listado 13.8 vuelve a utilizar la clase `Figura` y la modifica para convertirla en un tipo de datos abstracto. Para ahorrar espacio, aquí no se reproduce el resto del listado 13.7. Reemplace la declaración de `Figura` del listado 13.7, líneas 7 a 16, con la declaración de `Figura` del listado 13.8 y ejecute el programa de nuevo.

### ENTRADA LISTADO 13.8 Tipos de datos abstractos

```

1: class Figura
2: {
3: public:
4:     Figura(){}
5:     ~Figura(){}
6:     virtual long ObtenerArea() = 0; // error si se trata de instanciar
7:     virtual long ObtenerPerim()= 0; // la clase Figura
8:     virtual void Dibujar() = 0;
9: private:
10: };

```

**SALIDA**

```
(1)Circulo (2)Rectangulo (3)Cuadrado (0)Salir: 2
x x x x x x
x x x x x x
x x x x x x
x x x x x x

(1)Circulo (2)Rectangulo (3)Cuadrado (0)Salir: 3
x x x x x
x x x x x
x x x x x
x x x x x
x x x x x

(1)Circulo (2)Rectangulo (3)Cuadrado (0)Salir: 0
```

**ANÁLISIS**

Como puede ver, el funcionamiento del programa no sufre cambios. La única diferencia es que ahora sería imposible crear un objeto de la clase Figura.

**Tipos de datos abstractos**

Para declarar una clase como tipo de datos abstractos se incluyen una o más funciones virtuales puras en la declaración de la clase. Un función virtual pura se declara escribiendo = 0 después de la declaración de la función.

He aquí un ejemplo:

```
class Figura
{
    virtual void Dibujar() = 0;      // virtual pura
};
```

**Implementación de funciones virtuales puras**

Por lo general, las funciones virtuales puras de una clase base abstracta no se implementan. Como no se crean objetos de ese tipo, no hay razón para proporcionar implementaciones, y el ADT trabaja simplemente como la definición de una interfaz para los objetos derivados a partir de él.

Sin embargo, es posible proporcionar una implementación para una función virtual pura. La función puede entonces ser llamada por objetos que se deriven del ADT, tal vez para proporcionar una funcionalidad común para todas las funciones redefinidas. El listado 13.9 reproduce el listado 13.7, esta vez con Figura como un ADT y con una implementación para la función virtual pura Dibujar(). La clase Circulo redefine a Dibujar(), como se debe, pero luego hace una cadena ascendente para la función de la clase base para obtener una funcionalidad adicional.

En este ejemplo, la funcionalidad adicional es simplemente un mensaje adicional impreso, pero podemos imaginar que la clase base proporciona un mecanismo de dibujo compartido, tal vez preparar una ventana que utilizarán todas las clases derivadas.

**ENTRADA****LISTADO 13.9** Implementación de funciones virtuales puras

```
1:  //Implementación de funciones virtuales puras
2:
3:  #include <iostream.h>
4:
5:  class Figura
6:
7:  public:
8:      Figura(){}
9:  virtual ~Figura(){}
10:     virtual long ObtenerArea() = 0;
11:     virtual long ObtenerPerim()= 0;
12:     virtual void Dibujar() = 0;
13: private:
14: };
15:
16: void Figura::Dibujar()
17: {
18:     cout << "¡Mecanismo abstracto de dibujo!\n";
19: }
20:
21: class Circulo : public Figura
22: {
23: public:
24:     Circulo(int radio):suRadio(radio){}
25: virtual ~Circulo(){}
26:     long ObtenerArea() { return 3 * suRadio * suRadio; }
27:     long ObtenerPerim() { return 9 * suRadio; }
28:     void Dibujar();
29: private:
30:     int suRadio;
31:     int suCircunferencia;
32: };
33:
34: void Circulo::Dibujar()
35: {
36:     cout << "¡Aqui va una rutina para dibujar un Circulo!\n";
37:     Figura::Dibujar();
38: }
39:
40:
41: class Rectangulo : public Figura
42: {
43: public:
```

```
44:         Rectangulo(int longitud, int ancho):
45:             suLongitud(longitud), suAncho(ancho){}
46:     virtual ~Rectangulo(){}
47:         long ObtenerArea() { return suLongitud * suAncho; }
48:         long ObtenerPerim() {return 2*suLongitud + 2*suAncho; }
49:         virtual int ObtenerLongitud() { return suLongitud; }
50:         virtual int ObtenerAncho() { return suAncho; }
51:         void Dibujar();
52:     private:
53:         int suAncho;
54:         int suLongitud;
55:     };
56:
57:     void Rectangulo::Dibujar()
58:     {
59:         for (int i = 0; i<suLongitud; i++)
60:         {
61:             for (int j = 0; j<suAncho; j++)
62:                 cout << "x ";
63:
64:             cout << "\n";
65:         }
66:         Figura::Dibujar();
67:     }
68:
69:
70:     class Cuadrado : public Rectangulo
71:     {
72:     public:
73:         Cuadrado(int longitud);
74:         Cuadrado(int longitud, int ancho);
75:     virtual ~Cuadrado(){}
76:         long ObtenerPerim() {return 4 * ObtenerLongitud();}
77:     };
78:
79:     Cuadrado::Cuadrado(int longitud):
80:         Rectangulo(longitud,longitud)
81:     {}
82:
83:     Cuadrado::Cuadrado(int longitud, int ancho):
84:         Rectangulo(longitud,ancho)
85:
86:     {
87:         if (ObtenerLongitud() != ObtenerAncho())
88:             cout << "Error, no es un cuadrado... ¿un Rectangulo??\n";
89:     }
90:
91:     int main()
92:     {
93:         int opcion;
94:         bool fSalir = false;
```

13

**LISTADO 13.9** CONTINUACIÓN

```
95:     Figura * sp;
96:
97:     while (1)
98:     {
99:         cout << "(1)Circulo (2)Rectangulo (3)Cuadrado (0)Salir: ";
100:        cin >> opcion;
101:
102:        switch (opcion)
103:        {
104:            case 1: sp = new Circulo(5);
105:            break;
106:            case 2: sp = new Rectangulo(4,6);
107:            break;
108:            case 3: sp = new Cuadrado (5);
109:            break;
110:            default: fSalir = true;
111:            break;
112:        }
113:        if (fSalir)
114:            break;
115:
116:        sp->Dibujar();
117:        delete sp;
118:        cout << "\n";
119:    }
120:    return 0;
121: }
```

**ENTRADA**

(1)Circulo (2)Rectangulo (3)Cuadrado (0)Salir: 2

x x x x x  
x x x x x  
x x x x x  
x x x x x

iMecanismo abstracto de dibujo!

(1)Circulo (2)Rectangulo (3)Cuadrado (0)Salir: 3

x x x x x  
x x x x x  
x x x x x  
x x x x x  
x x x x x

iMecanismo abstracto de dibujo!

(1)Circulo (2)Rectangulo (3)Cuadrado (0)Salir: 0

**ANÁLISIS**

En las líneas 5 a 14 se declara el tipo de datos abstracto **Figura**, y sus tres métodos de acceso se declaran como virtuales puros. Hay que tener en cuenta que esto no es necesario. Si cualquiera fuera declarado como virtual puro, la clase hubiera sido un ADT.

Los métodos **ObtenerArea()** y **ObtenerPerim()** no se implementan, pero **Dibujar()** sí. **Círculo** y **Rectángulo** redefinen a **Dibujar()**, y ambos hacen una cadena ascendente para el método base, aprovechando la funcionalidad compartida de la clase base.

## **Jerarquías de abstracción complejas**

Algunas veces tendrá la necesidad de derivar ADTs de otros ADTs. Tal vez lo hará porque necesitará que algunas de las funciones virtuales puras dejen de ser puras, y que otras se queden como puras.

Si crea la clase **Animal**, puede hacer que **Comer()**, **Dormir()**, **Mover()** y **Reproducir()** sean funciones virtuales puras. Tal vez quiera derivar a **Mamífero** y a **Pez** de **Animal**.

Tal vez al examinarlos decida que cada **Mamífero** se reproduzca de la misma forma, por lo que la función **Mamífero::Reproducir()** no será pura, pero dejará a **Comer()**, **Dormir()** y **Mover()** como funciones virtuales puras.

De **Mamífero** derivará a **Perro**, y **Perro** deberá redefinir e implementar las tres funciones virtuales puras restantes, para que pueda crear objetos de tipo **Perro**.

Lo que está usted diciendo como diseñador de la clase, es que ningún **Animal** o ningún **Mamífero** puede ser instanciado, pero que todos los **Mamíferos** pueden heredar el método **Reproducir()** proporcionado sin redefinirlo.

El listado 13.10 muestra esta técnica con una implementación simplificada de estas clases.

**ENTRADA****LISTADO 13.10** Derivación de ADTs de otros ADTs

```

1: // Listado 13.10
2: // Derivación de ADTs de otros ADTs
3: #include <iostream.h>
4:
5: enum COLOR { Rojo, Verde, Azul, Amarillo, Blanco, Negro, Cafe } ;
6:
7: class Animal           // base común para Mamífero y Pez
8: {
9: public:
10:    Animal(int);
11:    virtual ~Animal() { cout << "Destructor de Animal...\n"; }
12:    virtual int ObtenerEdad() const { return suEdad; }
13:    virtual void AsignarEdad(int edad) { suEdad = edad; }
14:    virtual void Dormir() const = 0;

```

**LISTADO 13.10** CONTINUACIÓN

```
15:     virtual void Comer() const = 0;
16:     virtual void Reproducir() const = 0;
17:     virtual void Mover() const = 0;
18:     virtual void Hablar() const = 0;
19: private:
20:     int suEdad;
21: };
22:
23: Animal::Animal(int edad):
24: suEdad(edad)
25: {
26:     cout << "Constructor de Animal...\n";
27: }
28:
29: class Mamifero : public Animal
30: {
31: public:
32:     Mamifero(int edad):Animal(edad)
33:     { cout << "Constructor de Mamifero...\n"; }
34:     virtual ~Mamifero() { cout << "Destructor de Mamifero...\n"; }
35:     virtual void Reproducir() const
36:     { cout << "Reproducción de Mamifero representada...\n"; }
37: };
38:
39: class Pez : public Animal
40: {
41: public:
42:     Pez(int edad):Animal(edad)
43:     { cout << "Constructor de Pez...\n"; }
44:     virtual ~Pez() {cout << "Destructor de Pez...\n"; }
45:     virtual void Dormir() const { cout << "Pez roncando...\n"; }
46:     virtual void Comer() const { cout << "Pez comiendo...\n"; }
47:     virtual void Reproducir() const
48:     { cout << "Pez poniendo huevos...\n"; }
49:     virtual void Mover() const
50:     { cout << "Pez nadando...\n"; }
51:     virtual void Hablar() const { }
52: };
53:
54: class Caballo : public Mamifero
55: {
56: public:
57:     Caballo(int edad, COLOR color):
58:     Mamifero(edad), suColor(color)
59:     { cout << "Constructor de Caballo...\n"; }
60:     virtual ~Caballo() { cout << "Destructor de Caballo...\n"; }
61:     virtual void Hablar()const { cout << "iYihii!... \n"; }
62:     virtual COLOR ObtenerSuColor() const { return suColor; }
```

```
63:     virtual void Dormir() const
64:         { cout << "Caballo roncando...\n"; }
65:     virtual void Comer() const { cout << "Caballo comiendo...\n"; }
66:     virtual void Mover() const { cout << "Caballo corriendo...\n"; }
67:
68: protected:
69:     COLOR suColor;
70: };
71:
72: class Perro : public Mamifero
73: {
74: public:
75:     Perro(int edad, COLOR color):
76:         Mamifero(edad), suColor(color)
77:             { cout << "Constructor de Perro...\n"; }
78:     virtual ~Perro() { cout << "Destructor de Perro...\n"; }
79:     virtual void Hablar()const { cout << "Guau!... \n"; }
80:     virtual void Dormir() const { cout << "Perro roncando...\n"; }
81:     virtual void Comer() const { cout << "Perro comiendo...\n"; }
82:     virtual void Mover() const { cout << "Perro corriendo...\n"; }
83:     virtual void Reproducir() const
84:         { cout << "Perro reproduciéndose...\n"; }
85:
86: protected:
87:     COLOR suColor;
88: };
89:
90: int main()
91: {
92:     Animal *apAnimal=NULL;
93:     int opcion;
94:     bool fSalir = false;
95:
96:     while (1)
97:     {
98:         cout << "(1)Perro (2)Caballo (3)Pez (0)Salir: ";
99:         cin >> opcion;
100:
101:         switch (opcion)
102:         {
103:             case 1: apAnimal = new Perro(5,Cafe);
104:                 break;
105:             case 2: apAnimal = new Caballo(4,Negro);
106:                 break;
107:             case 3: apAnimal = new Pez (5);
108:                 break;
109:             default: fSalir = true;
110:                 break;
111:         }
112:         if (fSalir)
113:             break;
```

**LISTADO 13.10** CONTINUACIÓN

```

114:
115:     apAnimal->Hablar();
116:     apAnimal->Comer();
117:     apAnimal->Reproducir();
118:     apAnimal->Mover();
119:     apAnimal->Dormir();
120:     delete apAnimal;
121:     cout << "\n";
122: }
123: return 0;
124: }
```

**SALIDA**

```

(1)Perro (2)Caballo (3)Ave (0)Salir: 1
Constructor de Animal...
Constructor de Mamifero...
Constructor de Perro...
¡Guau!...
Perro comiendo...
Perro reproduciéndose....
Perro corriendo...
Perro roncando...
Destructor de Perro...
Destructor de Mamifero...
Destructor de Animal...

(1)Perro (2)Caballo (3)Ave (0)Salir: 0
```

**ANÁLISIS**

En las líneas 7 a 21 se declara el tipo de datos abstracto **Animal**. **Animal** tiene métodos de acceso virtuales no puros para **suEdad**, los cuales son compartidos por todos los objetos de tipo **Animal**. **Animal** tiene cinco funciones virtuales puras, **Dormir()**, **Comer()**, **Reproducir()**, **Mover()** y **Hablar()**.

**Mamifero** se deriva de **Animal**, se declara en las líneas 29 a 37 y no agrega datos. No obstante, redefine la función **Reproducir()**, proporcionando una forma común de reproducción para todos los mamíferos. **Pez** debe redefinir a **Reproducir()** ya que **Pez** se deriva directamente de **Animal** y no puede aprovechar la reproducción de los mamíferos (¡y eso es bueno!).

Las clases derivadas de **Mamifero** ya no tienen que redefinir la función **Reproducir()**, pero pueden hacerlo si quieren, como lo hace **Perro** en la línea 83. **Pez**, **Caballo** y **Perro** redefinen las funciones virtuales puras restantes, para que se puedan crear instancias de objetos de su tipo.

En el cuerpo del programa se utiliza un apuntador a **Animal** para apuntar a los diversos objetos derivados en turno. Se invocan los métodos virtuales, y con base en el enlace en tiempo de ejecución del apuntador, se llama al método correcto en la clase derivada.

Se generaría un error en tiempo de compilación si se tratara de instanciar un **Animal** o un **Mamifero**, ya que ambos son tipos de datos abstractos.

## ¿Qué tipos son abstractos?

En algunos programas, la clase `Animal` es abstracta, en otros no lo es. ¿Qué es lo que determina si una clase se hace abstracta o no?

La respuesta a esta pregunta no se decide por un factor intrínseco del mundo real, sino por lo que tenga sentido en el programa. Si está escribiendo un programa que represente una granja o un zoológico, tal vez quiera que `Animal` sea un tipo de datos abstracto, pero que `Perro` sea una clase de la que pueda instanciar objetos.

Por otro lado, si está haciendo una perrera animada, tal vez quiera dejar a `Perro` como un tipo de datos abstracto y sólo instanciar tipos de perros: Retrievers, Terriers, y así sucesivamente. El nivel de abstracción depende de la nitidez con la que necesite diferenciar sus tipos.

| DEBE                                                                                                                      | NO DEBE                                                                      |
|---------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------|
| <b>DEBE</b> utilizar tipos de datos abstractos para proporcionar una funcionalidad común para varias clases relacionadas. | <b>NO DEBE</b> tratar de instanciar un objeto de un tipo de datos abstracto. |
| <b>DEBE</b> redefinir todas las funciones virtuales puras.                                                                |                                                                              |
| <b>DEBE</b> hacer que cualquier función que se deba redefinir sea virtual pura.                                           |                                                                              |

## El patrón observador

Una tendencia muy popular en C++ es la creación y diseminación de patrones de diseño. Éstas son soluciones bien documentadas para problemas comunes encontrados por los programadores de C++. Como ejemplo, el patrón observador resuelve un problema común de herencia.

Imagine que desarrolla una clase cronómetro que sabe cómo contar los segundos transcurridos. Una clase así podría tener una variable miembro llamada `susSegundos`, que sería un entero, y tendría métodos para asignar valores a la variable miembro `susSegundos`, obtener valores de ella, e incrementarla.

Ahora suponga que su programa quiere que se le informe cada vez que se incremente la variable miembro `susSegundos` del cronómetro. Una solución obvia sería colocar un método de notificación en el cronómetro. Sin embargo, la notificación no es una parte intrínseca de la medición del tiempo, y el código complejo necesario para registrar esas clases que necesitan que se les informe cada vez que el reloj se incremente, realmente no pertenece a la clase cronómetro.

Lo que es más importante, después de descubrir la lógica de registrar aquellas clases que estén interesadas en estos cambios y luego notificarlos, le gustaría aislar esto dentro de una clase propia y poder volver a utilizarlo con otras clases que quieran ser “observadas” de esta manera.

Por lo tanto, una mejor solución es crear una clase observadora. Haga de este observador un tipo de datos abstracto con una función virtual pura llamada `Actualizar()`.

Ahora cree un segundo tipo de datos abstracto llamado `Sujeto`. `Sujeto` mantiene un arreglo de objetos `Observador` y también proporciona dos métodos: `registrar()` (el cual agrega observadores a su lista) y `Notificar()`, el cual se llama cuando hay algo que reportar.

Las clases que quieren ser notificadas de los cambios en su cronómetro heredan de `Observador`. El cronómetro mismo hereda de `Sujeto`. La clase `Observador` se registra a sí misma con la clase `Sujeto`. La clase `Sujeto` llama a `Notificar` cada vez que cambia (en este caso, cuando el cronómetro se actualiza).

Por último, hay que tener en cuenta que no todos los clientes de cronómetro quieren ser observables, por lo que crearemos una nueva clase llamada `CronometroObservado`, la cual hereda tanto de cronómetro como de `Sujeto`. Esto da a `CronometroObservado` las características de cronómetro, así como su capacidad para ser observado.

## **Unas palabras sobre la herencia múltiple, los tipos de datos abstractos y Java**

Muchos programadores de C++ saben que Java se basó, en gran parte, en C++, y aún así los creadores de Java optaron por omitir la herencia múltiple. Su opinión fue que la herencia múltiple presentaba complejidad que interfería con la facilidad de uso de Java. Ellos sintieron que podían cubrir el 90% de la funcionalidad de la herencia múltiple utilizando lo que se conoce como interfaces.

Una *interfaz* es muy similar a un tipo de datos abstracto en cuanto a que define un conjunto de funciones que sólo se pueden implementar en una clase derivada. Sin embargo, con las interfaces no se deriva directamente de la interfaz; se deriva de otra clase y se implementa la interfaz, algo muy parecido a la herencia múltiple. Por consecuencia, este matrimonio entre un tipo de datos abstracto y la herencia múltiple proporciona algo semejante a una clase de capacidades, pero sin la complejidad o sobrecarga producida por la herencia múltiple. Además, ya que las interfaces no necesitan tener ni implementaciones ni datos miembro, se elimina la necesidad de herencia virtual.

Que esto sea un error o una característica depende de la forma en que se vea. De cualquier manera, si comprende lo que son la herencia múltiple y los tipos de datos abstractos de C++, estará en una excelente posición para poder utilizar algunas de las características más avanzadas de Java, en caso de que decida aprender también ese lenguaje.

El patrón observador y la forma en que se implementa tanto en Java como en C++ se tratan detalladamente en el artículo de Robert Martin, titulado “C++ y Java: Una comparación crítica”, que aparece en la edición de enero de 1997 de *C++ Report*.

## Resumen

Hoy aprendió cómo vencer algunas de las limitaciones de la herencia simple. Conoció el peligro de la filtración ascendente de interfaces y los riesgos de la conversión descendente en la jerarquía de clases. También aprendió cómo utilizar la herencia múltiple, los problemas que puede crear y cómo resolverlos por medio de la herencia virtual.

También aprendió lo que son los tipos de datos abstractos y cómo crear clases abstractas por medio de funciones virtuales puras. Conoció la forma de implementar funciones virtuales puras y por qué y cuándo hacerlo. Por último, vio cómo implementar el patrón observador usando herencia múltiple y tipos de datos abstractos.

## Preguntas y respuestas

**P ¿Qué significa filtrar la funcionalidad de manera ascendente?**

**R** Esto se refiere a la idea de llevar la funcionalidad compartida hacia una clase base común. Si dos o más clases comparten una función, es recomendable encontrar una clase base común en la que se pueda guardar esa función.

**P ¿Es bueno utilizar siempre la filtración ascendente?**

**R** Sí, si filtra la funcionalidad compartida en las capas superiores de la jerarquía de clases. No, si sólo está moviendo la interfaz. Es decir, si sólo algunas clases derivadas pueden utilizar el método, sería un error moverlo hacia una clase base común. Si lo hace, tendrá que cambiar el tipo del objeto en tiempo de ejecución antes de decidir si puede invocar a la función.

**P ¿Por qué es malo cambiar el tipo de un objeto en tiempo de ejecución?**

**R** Con programas grandes, las instrucciones switch se vuelven enormes y difíciles de mantener. El objeto de las funciones virtuales es dejar que la tabla virtual determine el tipo del objeto en tiempo de ejecución, en lugar de que lo haga el programador.

**P ¿Por qué es mala la conversión?**

**R** La conversión no es mala si se hace en una manera que sea segura para el tipo de datos o la clase. Si se llama a una función que sabe que el objeto debe ser de un tipo específico, hacer la conversión a ese tipo está bien. La conversión puede minar la poderosa comprobación de tipos de C++, y eso es lo que se quiere evitar. Si está cambiando el tipo del objeto en tiempo de ejecución y luego convirtiendo un apuntador, eso puede ser una señal de advertencia de que algo está mal en su diseño.

**P ¿Por qué no hacer virtuales todas las funciones?**

**R** Las funciones virtuales son soportadas por una tabla de funciones virtuales, lo que provoca una sobrecarga en tiempo de ejecución, tanto en el tamaño del programa como en su rendimiento. El *aptrv*, o apuntador a función virtual, es un detalle de implementación de las funciones virtuales. Cada objeto de una clase que tenga funciones virtuales tiene un *aptrv*, el cual apunta a la tabla de funciones virtuales para esa clase. Si tiene clases muy pequeñas de las que no espera derivar otras, tal vez no quiera hacer ninguna función virtual.

**P ¿Cuándo se debe hacer virtual el destructor?**

**R** En cualquier momento que usted crea que la clase va a derivar a otras clases, y vaya a utilizar un apuntador a la clase base para tener acceso a un objeto de la subclase. Como regla general, si ha hecho virtual cualquiera de las funciones de su clase, asegúrese de que el destructor también sea virtual.

**P ¿Por qué tomarse la molestia de crear un tipo de datos abstracto?, ¿por qué no sólo hacerlo no abstracto y evitar crear objetos de ese tipo?**

**R** El propósito de muchas de las convenciones de C++ es ayudar al compilador a encontrar errores, para poder evitar errores en tiempo de ejecución en el código que proporcione a sus clientes. Hacer una clase abstracta (es decir, proporcionar funciones virtuales puras) ocasiona que el compilador marque como un error cualquier objeto creado de ese tipo abstracto.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

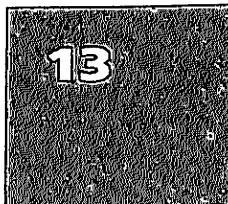
### Cuestionario

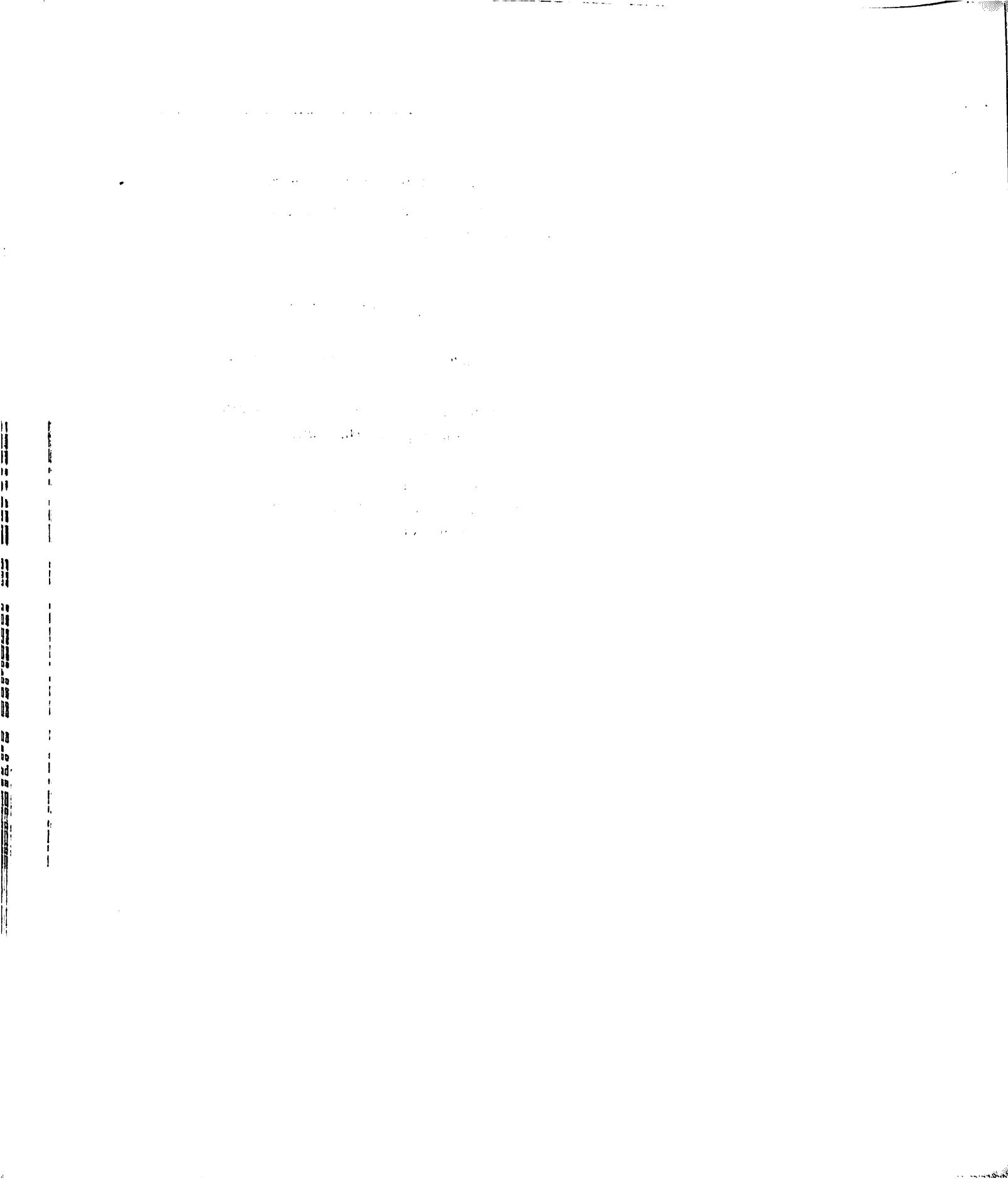
1. ¿Qué es una conversión descendente?
2. ¿Qué es el *aptrv*?
3. Si un rectángulo “redondo” tiene bordes rectos y esquinas redondeadas, y su clase *RectRedondo* hereda tanto de *Rectangulo* como de *Circulo*, y éstos a su vez heredan de *Figura*, ¿cuántas *Figuras* se crearán cuando cree un *RectRedondo*?
4. Si *Caballo* y *Ave* heredan de *Animal* usando herencia virtual pública, ¿inicializan sus constructores el constructor de *Animal*? Si *Pegaso* hereda tanto de *Caballo* como de *Ave*, ¿cómo inicializa el constructor de *Animal*?

5. Declare una clase llamada **vehiculo** y conviértala en un tipo de datos abstracto.
6. Si una clase base es un ADT y tiene tres funciones virtuales puras, ¿cuántas de estas funciones se deben redefinir en sus clases derivadas?

## Ejercicios

1. Muestre la declaración de una clase llamada **AvionJet**, que herede de **Cohete** y de **Avion**.
2. Muestre la declaración de una clase llamada **777**, que herede de la clase **AvionJet** descrita en el ejercicio 1.
3. Escriba un programa que derive a **Auto** y a **Camion** de la clase **Vehiculo**. Convierta a **Vehiculo** en un ADT que tenga dos funciones virtuales puras. Haga que **Auto** y **Camion** no sean ADTs.
4. Modifique el programa del ejercicio 3 para que **Auto** sea un ADT, y derive de **Auto** a **AutoDeportivo**, **Vagoneta** y **Sedan**. En la clase **Auto**, proporcione una implementación para una de las funciones virtuales puras de **Vehiculo** y hágala no pura.





# SEMANA 2

DÍA 14

## Clases y funciones especiales

C++ ofrece varias maneras de limitar el alcance y el impacto de variables y apuntadores. Hasta ahora ha visto cómo crear variables globales, variables locales de funciones, apuntadores a variables y variables miembro de clases. Hoy aprenderá lo siguiente:

- Qué son los datos miembro estáticos (variables) y las funciones miembro estáticas
- Cómo utilizar variables miembro estáticas y funciones miembro estáticas
- Cómo crear y manipular apuntadores a funciones y apuntadores a funciones miembro
- Cómo trabajar con arreglos de apuntadores a función

### Datos miembro estáticos

Hasta ahora probablemente ha pensado que los datos de cada objeto son sólo para ese objeto y no se comparten entre objetos de una clase. Por ejemplo, si tiene cinco objetos Gato, cada uno tiene su propia edad, peso y otros datos. La edad de uno no afecta la edad de otro.

Sin embargo, a veces necesitará mantener el registro de una reserva de datos. Por ejemplo, tal vez quiera saber cuántos objetos de una clase específica se han creado en su programa, y cuantos existen todavía. Las variables miembro estáticas se comparten entre todas las instancias de una clase. Son un pacto entre los datos globales, que están disponibles para todos los componentes de su programa, y los datos miembro, que por lo general están disponibles sólo para cada objeto.

Puede pensar que un miembro estático pertenece a la clase en lugar de al objeto. Los datos miembro regulares son uno por objeto, pero los miembros estáticos son uno por clase. El listado 14.1 declara un objeto Gato con un dato miembro estático, llamado CuantosGatos. Esta variable mantiene el registro de cuántos objetos Gato se han creado. Esto se hace incrementando la variable estática CuantosGatos con cada construcción y decrementándola con cada destrucción.

**ENTRADA LISTADO 14.1 Datos miembro estáticos**

---

```
1: //Listado 14.1 datos miembro estáticos
2:
3: #include <iostream.h>
4:
5: class Gato
6: {
7: public:
8:     Gato(int edad):suEdad(edad){CuantosGatos++; }
9:     virtual ~Gato() { CuantosGatos--; }
10:    virtual int ObtenerEdad() { return suEdad; }
11:    virtual void AsignarEdad(int edad) { suEdad = edad; }
12:    static int CuantosGatos;
13:
14: private:
15:     int suEdad;
16:
17: };
18:
19: int Gato::CuantosGatos = 0;
20:
21: int main()
22: {
23:     const int MaxGatos = 5; int i;
24:     Gato *CasaGatos[MaxGatos];
25:     for (i = 0; i<MaxGatos; i++)
26:         CasaGatos[i] = new Gato(i);
27:
28:     for (i = 0; i<MaxGatos; i++)
29:     {
30:         cout << "iQuedan ";
31:         cout << Gato::CuantosGatos;
32:         cout << " gatos!\n";
33:         cout << "Se va a eliminar el que tiene ";
```

```

34:         cout << CasaGatos[i]->ObtenerEdad();
35:         cout << " años de edad\n";
36:         delete CasaGatos[i];
37:         CasaGatos[i] = 0;
38:     }
39: return 0;
40: }
```

**SALIDA**

¡Quedan 5 gatos!  
 Se va a eliminar el que tiene 0 años de edad  
 ¡Quedan 4 gatos!  
 Se va a eliminar el que tiene 1 años de edad  
 ¡Quedan 3 gatos!  
 Se va a eliminar el que tiene 2 años de edad  
 ¡Quedan 2 gatos!  
 Se va a eliminar el que tiene 3 años de edad  
 ¡Quedan 1 gatos!  
 Se va a eliminar el que tiene 4 años de edad

**ANÁLISIS**

En las líneas 5 a 17 se declara la clase simplificada Gato. En la línea 12 se declara una variable miembro estática de tipo int llamada CuantosGatos.

La declaración de CuantosGatos no define un entero; no se reserva espacio de almacenamiento. A diferencia de las variables miembro que no son estáticas, no se reserva espacio de almacenamiento al instanciar un objeto Gato, debido a que la variable miembro CuantosGatos no se encuentra en el objeto. Por lo tanto, la variable se define y se inicializa en la línea 19.

Es un error común olvidar definir las variables miembro estáticas de las clases. ¡No permita que esto le pase a usted! Desde luego que si le pasa, el compilador GNU emitirá una variedad de mensajes de error como los que se muestran a continuación:

```

lst14-01.cxx: In method 'Gato::Gato(int)':
lst14-01.cxx:8: 'CuantosGatos' undeclared (first use this function)
lst14-01.cxx:8: (Each undeclared identifier is reported only once
lst14-01.cxx:8: for each function it appears in.)
lst14-01.cxx: In method 'Gato::~Gato()':
lst14-01.cxx:9: 'CuantosGatos' undeclared (first use this function)
lst14-01.cxx: At top level:
lst14-01.cxx:19: 'int Gato::CuantosGatos' is not a static member of 'class Gato'
lst14-01.cxx: In function 'int main()':
lst14-01.cxx:31: 'CuantosGatos' is not a member of type 'Gato'
```

Algunos compiladores no detectan el problema, y el enlazador emite un mensaje como:  
 undefined symbol Gato::CuantosGatos

No necesita hacer esto para suEdad debido a que no es una variable miembro estática y se define cada vez que usted crea un objeto Gato, lo cual hace aquí en la línea 26.

El constructor para Gato incrementa la variable miembro estática en la línea 8. El destructor la decrementa en la línea 9. Así que, en cualquier momento dado, CuantosGatos tiene una medición exacta de cuántos objetos Gato fueron creados y aún no son destruidos.

El programa controlador de las líneas 21 a 40 crea instancias de cinco Gatos y los coloca en un arreglo. Esto llama a cinco constructores de Gato, y por consecuencia CuantosGatos se incrementa cinco veces a partir de su valor inicial 0.

Luego, el programa avanza con un ciclo a través de cada una de las cinco posiciones del arreglo e imprime el valor de CuantosGatos antes de eliminar el apuntador a Gato actual. La impresión refleja que el valor de inicio es 5 (después de todo, se construyen 5), y que cada vez que se ejecuta el ciclo queda un Gato menos.

Observe que CuantosGatos es una variable pública y `main()` tiene acceso directo a ella. No existe motivo alguno para exponer esta variable miembro de esta manera. Es preferible hacerla privada junto con las otras variables miembro y proporcionar un método público de acceso, siempre y cuando se tenga acceso a los datos siempre a través de una instancia de Gato. Por otro lado, si quiere tener acceso a estos datos en forma directa, sin tener necesariamente un objeto Gato disponible, tiene dos opciones: mantener la variable pública, como se muestra en el listado 14.2, o proporcionar una función miembro estática, como se explicará más adelante en este día.

**ENTRADA****LISTADO 14.2 Acceso a los miembros estáticos sin un objeto**

---

```
1: //Listado 14.2 datos miembro estáticos
2:
3: #include <iostream.h>
4:
5: class Gato
6: {
7: public:
8:     Gato(int edad):suEdad(edad){CuantosGatos++; }
9:     virtual ~Gato() { CuantosGatos--; }
10:    virtual int ObtenerEdad() { return suEdad; }
11:    virtual void AsignarEdad(int edad) { suEdad = edad; }
12:    static int CuantosGatos;
13:
14: private:
15:     int suEdad;
16:
17: };
18:
19: int Gato::CuantosGatos = 0;
20:
21: void FuncionTelepatica();
22:
23: int main()
```

```
24:  {
25:      const int MaxGatos = 5; int i;
26:      Gato *CasaGatos[MaxGatos];
27:      for (i = 0; i<MaxGatos; i++)
28:      {
29:          CasaGatos[i] = new Gato(i);
30:          FuncionTelepatica();
31:      }
32:
33:      for (i = 0; i<MaxGatos; i++)
34:      {
35:          delete CasaGatos[i];
36:          FuncionTelepatica();
37:      }
38:      return 0;
39:  }
40:
41: void FuncionTelepatica()
42: {
43:     cout << "¡Hay ";
44:     cout << Gato::CuantosGatos << " gatos vivos!\n";
45: }
```

**SALIDA**

```
¡Hay 1 gatos vivos!
¡Hay 2 gatos vivos!
¡Hay 3 gatos vivos!
¡Hay 4 gatos vivos!
¡Hay 5 gatos vivos!
¡Hay 4 gatos vivos!
¡Hay 3 gatos vivos!
¡Hay 2 gatos vivos!
¡Hay 1 gatos vivos!
¡Hay 0 gatos vivos!
```

**ANÁLISIS**

El listado 14.2 es muy parecido al listado 14.1, excepto por la adición de una nueva función, llamada `FuncionTelepatica()`. Esta función no crea un objeto `Gato`, ni toma un objeto `Gato` como parámetro, pero puede tener acceso a la variable miembro `CuantosGatos`. De nuevo, vale la pena recalcar que esta variable miembro no se encuentra en ningún objeto específico; se encuentra en la clase como un todo y, de ser pública, cualquier función que se encuentre en el programa puede tener acceso a ella.

La alternativa de hacer esta variable pública es hacerla privada. Si lo hace, puede tener acceso a ella a través de una función miembro, pero entonces debe tener disponible un objeto de esa clase. El listado 14.3 muestra este método. La alternativa, funciones miembro estáticas, se discute inmediatamente después del análisis del listado 14.3.

**ENTRADA****LISTADO 14.3** Acceso a los miembros estáticos por medio de funciones miembro que no son estáticas

```
1: //Listado 14.3 datos miembro estáticos privados
2:
3: #include <iostream.h>
4:
5: class Gato
6: {
7: public:
8:     Gato(int edad):suEdad(edad){CuantosGatos++;}
9:     virtual ~Gato() { CuantosGatos--; }
10:    virtual int ObtenerEdad() { return suEdad; }
11:    virtual void AsignarEdad(int edad) { suEdad = edad; }
12:    virtual int ObtenerCuantos() { return CuantosGatos; }
13:
14:
15: private:
16:     int suEdad;
17:     static int CuantosGatos;
18: };
19:
20: int Gato::CuantosGatos = 0;
21:
22: int main()
23: {
24:     const int MaxGatos = 5; int i;
25:     Gato *CasaGatos[MaxGatos];
26:     for (i = 0; i<MaxGatos; i++)
27:         CasaGatos[i] = new Gato(i);
28:
29:     for (i = 0; i<MaxGatos; i++)
30:     {
31:         cout << "Quedan ";
32:         cout << CasaGatos[i]->ObtenerCuantos();
33:         cout << " gatos!\n";
34:         cout << "Se va a eliminar el que tiene ";
35:         cout << CasaGatos[i]->ObtenerEdad()+2;
36:         cout << " años de edad\n";
37:         delete CasaGatos[i];
38:         CasaGatos[i] = 0;
39:     }
40:     return 0;
41: }
```

**SALIDA**

```
iQuedan 5 gatos!
Se va a eliminar el que tiene 2 años de edad
iQuedan 4 gatos!
Se va a eliminar el que tiene 3 años de edad
iQuedan 3 gatos!
Se va a eliminar el que tiene 4 años de edad
```

```

¡Quedan 2 gatos!
Se va a eliminar el que tiene 5 años de edad
¡Quedan 1 gatos!
Se va a eliminar el que tiene 6 años de edad

```

**ANÁLISIS**

En la línea 17 se declara la variable miembro estática CuantosGatos con acceso privado. Ahora no puede tener acceso a esta variable desde funciones que no sean miembro, como FuncionTelepatica del listado anterior.

Aún cuando CuantosGatos es estática, sigue dentro del alcance de la clase. Cualquier función de la clase, como ObtenerCuantos(), puede tener acceso a ella, así como las funciones miembro pueden tener acceso a cualquier dato miembro. Sin embargo, para que una función pueda llamar a ObtenerCuantos(), debe tener un objeto desde el que pueda llamarla.

| <b>DEBE</b>                                                                                                                                                                                                                                   | <b>No DEBE</b>                                                                                                                                                              |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>DEBE</b> utilizar variables miembro estáticas para compartir datos entre todas las instancias de una clase.</p> <p><b>DEBE</b> hacer que las variables miembro estáticas sean protegidas o privadas si quiere restringir su acceso.</p> | <p><b>NO DEBE</b> utilizar variables miembro estáticas para guardar datos para un objeto. Los datos miembro estáticos se comparten entre todos los objetos de su clase.</p> |

## Funciones miembro estáticas

Las funciones miembro estáticas son como las variables miembro estáticas: no existen en un objeto sino en el alcance de la clase. Por ende, se pueden llamar sin necesidad de tener un objeto de esa clase, como se muestra en el listado 14.4.

**ENTRADA** **LISTADO 14.4** Funciones miembro estáticas

```

1: //Listado 14.4 Funciones miembro estáticas
2:
3: #include <iostream.h>
4:
5: class Gato
6: {
7: public:
8:     Gato(int edad):suEdad(edad){CuantosGatos++; }
9:     virtual ~Gato() { CuantosGatos--; }
10:    virtual int ObtenerEdad() { return suEdad; }
11:    virtual void AsignarEdad(int edad) { suEdad = edad; }
12:    static int ObtenerCuantos() { return CuantosGatos; }
13: private:
14:     int suEdad;

```

**LISTADO 14.4** CONTINUACIÓN

```
15:     static int CuantosGatos;
16: };
17:
18: int Gato::CuantosGatos = 0;
19:
20: void FuncionTelepatica();
21:
22: int main()
23: {
24:     const int MaxGatos = 5;
25:     Gato *CasaGatos[MaxGatos]; int i;
26:     for (i = 0; i<MaxGatos; i++)
27:     {
28:         CasaGatos[i] = new Gato(i);
29:         FuncionTelepatica();
30:     }
31:
32:     for (i = 0; i<MaxGatos; i++)
33:     {
34:         delete CasaGatos[i];
35:         FuncionTelepatica();
36:     }
37:     return 0;
38: }
39:
40: void FuncionTelepatica()
41: {
42:     cout << "¡Hay " << Gato::ObtenerCuantos() << " gatos vivos!\n";
43: }
```

**SALIDA**

```
¡Hay 1 gatos vivos!
¡Hay 2 gatos vivos!
¡Hay 3 gatos vivos!
¡Hay 4 gatos vivos!
¡Hay 5 gatos vivos!
¡Hay 4 gatos vivos!
¡Hay 3 gatos vivos!
¡Hay 2 gatos vivos!
¡Hay 1 gatos vivos!
¡Hay 0 gatos vivos!
```

**ANÁLISIS**

La variable miembro estática CuantosGatos se declara para tener acceso privado, en la línea 15 de la declaración de Gato. La función de acceso, ObtenerCuantos(), se declara como pública y estática en la línea 12.

Como `ObtenerCuantos()` es pública, cualquier función puede tener acceso a ella, y como es estática, no hay necesidad de tener un objeto de tipo `Gato` para llamarla. Por lo tanto, en la línea 42 la función `FuncionTelepatica()` puede tener acceso al método de acceso estático, aunque no tenga acceso a un objeto `Gato`. Claro que podría haber llamado a `ObtenerCuantos()` desde los objetos `Gato` disponibles en `main()`, de igual forma que con cualquier otro método de acceso.

### Nota

Las funciones miembro estáticas no tienen un apuntador `this`. Por lo tanto, no se pueden declarar como `const`. Además, como las variables de datos miembro se acceden en funciones miembro mediante el apuntador `this`, ¡las funciones miembro estáticas no pueden tener acceso a ninguna variable miembro que no sea estática!

### Funciones miembro estáticas

Puede tener acceso a las funciones miembro estáticas llamándolas desde un objeto de la clase, como lo hace con cualquier otra función miembro, o puede llamarlas sin un objeto si identifica completamente el nombre de la clase y del método.

He aquí un ejemplo:

```
class Gato
{
public:
    static int ObtenerCuantos() { return CuantosGatos; }
private:
    static int CuantosGatos;
};
int Gato::CuantosGatos = 0;
int main()
{
    int cuantos;
    Gato elGato; // definir un gato
    cuantos = elGato.ObtenerCuantos(); // acceso a través de un objeto
    cuantos = Gato::ObtenerCuantos(); // acceso sin un objeto
}
```

## Apuntadores a funciones

Así como un nombre de arreglo es un apuntador constante al primer elemento del arreglo, el nombre de una función es un apuntador constante a la función. Es posible declarar una variable de apuntador que apunte a una función y que invoque a la función mediante ese apuntador. Esto puede ser muy útil; le permite crear programas que deciden cuáles funciones invocar con base en las acciones del usuario.

La única parte difícil sobre los apuntadores a funciones es entender el tipo de objeto al que se está apuntando. Un apuntador a `int` apunta a una variable entera, y un apuntador a una función debe apuntar a una función del tipo de valor de retorno y firma apropiados.

En la siguiente declaración

```
long (* funcPtr) (int);
```

`funcPtr` se declara como apuntador (observe el `*` que está antes del nombre) que apunta a una función que toma un parámetro entero y regresa un tipo `long`. Los paréntesis alrededor de `* funcPtr` son necesarios, ya que los paréntesis alrededor de `int` vinculan de forma más estrecha, es decir, tienen una mayor precedencia que el operador de indirección (`*`). Sin los primeros paréntesis, esto declararía una función que toma un entero y regresa un apuntador a un tipo `long`. (Recuerde que los espacios no importan aquí.)

Examine estas dos declaraciones:

```
long * Funcion (int);  
long (* funcPtr) (int);
```

La primera, `Funcion ()`, es una función que toma un entero y regresa un apuntador a una variable de tipo `long`. La segunda, `funcPtr`, es un apuntador a una función que toma un entero y regresa una variable de tipo `long`.

La declaración de un apuntador a una función siempre incluirá el tipo de valor de retorno y los paréntesis indicando el tipo de los parámetros, en caso de haberlos. El listado 14.5 muestra la declaración y el uso de los apuntadores a funciones.

**ENTRADA****LISTADO 14.5 Apuntadores a funciones**

```
1: // Listado 14.5 Uso de apuntadores a funciones  
2:  
3: #include <iostream.h>  
4:  
5: void Cuadrado (int&,int&);  
6: void Cubo (int&, int&);  
7: void Intercambiar (int&, int &);  
8: void ObtenerValores(int&, int&);  
9: void ImprimirValores(int, int);  
10:  
11: int main()  
12: {  
13:     void (* apFunc) (int &, int &);  
14:     bool fSalir = false;  
15:  
16:     int valUno=1, valDos=2;
```

```
17:     int opcion;
18:     while (fSalir == false)
19:     {
20:         cout << "(0)Salir (1)Cambiar Valores (2)Cuadrado
21:             (3)Cubo (4)Intercambiar: ";
22:         cin >> opcion;
23:         switch (opcion)
24:         {
25:             case 1: apFunc = ObtenerValores; break;
26:             case 2: apFunc = Cuadrado; break;
27:             case 3: apFunc = Cubo; break;
28:             case 4: apFunc = Intercambiar; break;
29:             default : fSalir = true; break;
30:         }
31:         if (fSalir)
32:             break;
33:
34:         ImprimirValores(valUno, valDos);
35:         apFunc(valUno, valDos);
36:         ImprimirValores(valUno, valDos);
37:     }
38:     return 0;
39: }
40:
41: void ImprimirValores(int x, int y)
42: {
43:     cout << "x: " << x << " y: " << y << endl;
44: }
45:
46: void Cuadrado (int & rX, int & rY)
47: {
48:     rX *= rX;
49:     rY *= rY;
50: }
51:
52: void Cubo (int & rX, int & rY)
53: {
54:     int tmp;
55:
56:     tmp = rX;
57:     rX *= rX;
58:     rX = rX * tmp;
59:
60:     tmp = rY;
61:     rY *= rY;
62:     rY = rY * tmp;
63: }
64:
65: void Intercambiar(int & rX, int & rY)
66: {
```

**LISTADO 14.5** CONTINUACIÓN

```

67:     int temp;
68:     temp = rX;
69:     rX = rY;
70:     rY = temp;
71: }
72:
73: void ObtenerValores (int & rValUno, int & rValDos)
74: {
75:     cout << "Nuevo valor para valUno: ";
76:     cin >> rValUno;
77:     cout << "Nuevo valor para valDos: ";
78:     cin >> rValDos;
79: }
```

**SALIDA**

```

(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 1
x: 1 y: 2
Nuevo valor para valUno: 2
Nuevo valor para valDos: 3
x: 2 y: 3
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 3
x: 2 y: 3
x: 8 y: 27
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 2
x: 8 y: 27
x: 64 y: 729
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 4
x: 64 y: 729
x: 729 y: 64
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 0
```

**ANÁLISIS** En las líneas 5 a 8 se declaran cuatro funciones, cada una con el mismo tipo de valor de retorno y firma, que regresan void y toman como parámetros dos referencias a enteros.

En la línea 13 se declara apFunc como un apuntador a una función que regresa void y toma dos parámetros de referencia a enteros. apFunc puede apuntar a cualquiera de las funciones anteriores. Se ofrece repetidamente al usuario la opción de cuál función invocar, y apFunc se asigna de acuerdo con la respuesta. En las líneas 34 a 36 se imprime el valor actual de los dos enteros, se invoca a la función asignada actualmente, y luego se vuelven a imprimir los valores.

**Apuntador a función**

Un apuntador a una función se invoca de la misma manera que las funciones a las que apunta, excepto que se utiliza el nombre del apuntador a la función, en lugar del nombre de la función.

Para asignar un apuntador a una función específica, se asigna al nombre de la función sin los paréntesis. El nombre de la función es un apuntador constante a la función en sí. Utilice el apuntador a una función de la misma forma que utilizaría el nombre de la función. El apuntador a una función debe concordar con el valor de retorno y la firma de la función a la cual esté asignado.

He aquí un ejemplo:

```
long (*apFuncUno) (int, int);
long UnaFuncion (int, int);
apFuncUno = UnaFuncion;
apFuncUno(5,7);
```

## Por qué utilizar apuntadores a funciones

Evidentemente, usted podría escribir el programa del listado 14.5 sin los apuntadores a funciones, pero el uso de estos apuntadores hace que la intención y el uso del programa sean explícitos: escoja una función de una lista, y luego invóquela.

El listado 14.6 utiliza los prototipos y las definiciones de funciones del listado 14.5, pero el cuerpo del programa no utiliza un apuntador a una función. Analice las diferencias entre estos dos listados.

### ENTRADA

### **LISTADO 14.6** Modificación del listado 14.5, esta vez sin el apuntador a una función

```
1: // Listado 14.6 Sin apuntadores a funciones
2:
3: #include <iostream.h>
4:
5: void Cuadrado (int&,int&);
6: void Cubo (int&, int&);
7: void Intercambiar (int&, int &);
8: void ObtenerValores(int&, int&);
9: void ImprimirValores(int, int);
10:
11: int main()
12: {
13:     bool fSalir = false;
14:     int valUno=1, valDos=2;
15:     int opcion;
16:     while (fSalir == false)
17:     {
18:         cout << "(0)Salir (1)Cambiar Valores (2)Cuadrado
19:         (3)Cubo (4)Intercambiar: ";
20:         cin >> opcion;
21:         switch (opcion)
```

14

**LISTADO 14.6** CONTINUACIÓN

```
21:         {
22:             case 1:
23:                 ImprimirValores(valUno, valDos);
24:                 ObtenerValores(valUno, valDos);
25:                 ImprimirValores(valUno, valDos);
26:                 break;
27:
28:             case 2:
29:                 ImprimirValores(valUno, valDos);
30:                 Cuadrado(valUno, valDos);
31:                 ImprimirValores(valUno, valDos);
32:                 break;
33:
34:             case 3:
35:                 ImprimirValores(valUno, valDos);
36:                 Cubo(valUno, valDos);
37:                 ImprimirValores(valUno, valDos);
38:                 break;
39:
40:             case 4:
41:                 ImprimirValores(valUno, valDos);
42:                 Intercambiar(valUno, valDos);
43:                 ImprimirValores(valUno, valDos);
44:                 break;
45:
46:             default :
47:                 fSalir = true;
48:                 break;
49:             }
50:
51:             if (fSalir)
52:                 break;
53:             }
54:             return 0;
55:         }
56:
57:         void ImprimirValores(int x, int y)
58:     {
59:         cout << "x: " << x << " y: " << y << endl;
60:     }
61:
62:         void Cuadrado (int & rX, int & rY)
63:     {
64:         rX *= rX;
65:         rY *= rY;
66:     }
67:
68:         void Cubo (int & rX, int & rY)
```

```
69:     {
70:         int tmp;
71:
72:         tmp = rX;
73:         rX *= rX;
74:         rX = rX * tmp;
75:
76:         tmp = rY;
77:         rY *= rY;
78:         rY = rY * tmp;
79:     }
80:
81:     void Intercambiar(int & rX, int & rY)
82:     {
83:         int temp;
84:         temp = rX;
85:         rX = rY;
86:         rY = temp;
87:     }
88:
89:     void ObtenerValores (int & rValUno, int & rValDos)
90:     {
91:         cout << "Nuevo valor para valUno: ";
92:         cin >> rValUno;
93:         cout << "Nuevo valor para valDos: ";
94:         cin >> rValDos;
95:     }
```

**SALIDA**

```
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 1
x: 1 y: 2
Nuevo valor para valUno: 2
Nuevo valor para valDos: 3
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 3
x: 2 y: 3
x: 8 y: 27
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 2
x: 8 y: 27
x: 64 y: 729
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 4
x: 64 y: 729
x: 729 y: 64
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 0
```

**ANÁLISIS**

Se ha omitido la implementación de las funciones debido a que es idéntica a la que se proporciona en el listado 14.5. Como puede ver, la salida no cambia, pero el cuerpo del programa se ha expandido de 22 líneas a 46. Las llamadas a `ImprimirValores()` se deben repetir para cada caso.

Fue tentador colocar `ImprimirValores()` al inicio del ciclo `while` y de nuevo al final, en lugar de colocarlo en cada instrucción `case`. Sin embargo, esto habría llamado a `ImprimirValores()` incluso para el caso de salida, y esto no era parte de la especificación.

Dejando a un lado el aumento de tamaño del código y las llamadas repetidas para hacer lo mismo, la claridad en general está algo reducida. Sin embargo, éste es un caso artificial, creado para mostrar cómo funcionan los apuntadores a funciones. En condiciones reales, las ventajas son aún más claras: los apuntadores a funciones pueden eliminar código duplicado, clarificar su programa y permitirle crear tablas de funciones a llamar con base en las condiciones en tiempo de ejecución.

#### Invocación abreviada

No necesita desreferenciar el apuntador a función, aunque puede hacerlo. Por lo tanto, si `apFunc` es un apuntador a una función que toma un parámetro entero y regresa una variable de tipo `long`, y asigna `apFunc` a una función relacionada, puede invocar esa función ya sea con `apFunc(x);`

o con

`(*apFunc)(x);`

Las dos formas son idénticas. La primera es sólo una versión abreviada de la segunda.

## Uso de arreglos de apuntadores a funciones

Así como puede declarar un arreglo de apuntadores a enteros, también puede declarar un arreglo de apuntadores a funciones que regresen un tipo de valor específico y que tengan una firma específica. El listado 14.7 es una reproducción del listado 14.5, pero esta vez se utiliza un arreglo para invocar todas las opciones al mismo tiempo.

#### ENTRADA LISTADO 14.7 Muestra del uso de un arreglo de apuntadores a funciones

```
1: // Listado 14.7 Muestra del uso de un arreglo de
   apuntadores a funciones
2:
3: #include <iostream.h>
4:
5: void Cuadrado (int&,int&);
6: void Cubo (int&, int&);
7: void Intercambiar (int&, int &);
8: void ObtenerValores(int&, int&);
9: void ImprimirValores(int, int);
10:
11: int main()
```

```
12:      {
13:          int valUno=1, valDos=2;
14:          int opcion, i;
15:          const int MaxArreglo = 5;
16:          void (*apFuncArreglo[MaxArreglo])(int&, int&);
17:
18:          for (i=0;i<MaxArreglo;i++)
19:          {
20:              cout << "(1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar:
21: ";
22:              cin >> opcion;
23:              switch (opcion)
24:              {
25:                  case 1:apFuncArreglo[i] = ObtenerValores; break;
26:                  case 2:apFuncArreglo[i] = Cuadrado; break;
27:                  case 3:apFuncArreglo[i] = Cubo; break;
28:                  case 4:apFuncArreglo[i] = Intercambiar; break;
29:                  default:apFuncArreglo[i] = 0;
30:              }
31:          }
32:          for (i=0;i<MaxArreglo; i++)
33:          {
34:              if (apFuncArreglo[i] == 0)
35:                  continue;
36:              apFuncArreglo[i](valUno,valDos);
37:              ImprimirValores(valUno,valDos);
38:          }
39:      return 0;
40:  }
41:
42:  void ImprimirValores(int x, int y)
43:  {
44:      cout << "x: " << x << " y: " << y << endl;
45:  }
46:
47:  void Cuadrado (int & rX, int & rY)
48:  {
49:      rX *= rX;
50:      rY *= rY;
51:  }
52:
53:  void Cubo (int & rX, int & rY)
54:  {
55:      int tmp;
56:
57:      tmp = rX;
58:      rX *= rX;
59:      rX = rX * tmp;
60:
61:      tmp = rY;
62:      rY *= rY;
```



continúa

**LISTADO 14.7** CONTINUACIÓN

```

63:         rY = rY * tmp;
64:     }
65:
66:     void Intercambiar(int & rX, int & rY)
67:     {
68:         int temp;
69:         temp = rX;
70:         rX = rY;
71:         rY = temp;
72:     }
73:
74:     void ObtenerValores (int & rValUno, int & rValDos)
75:     {
76:         cout << "Nuevo valor para valUno: ";
77:         cin >> rValUno;
78:         cout << "Nuevo valor para valDos: ";
79:         cin >> rValDos;
80:     }

```

**SALIDA**

```

(1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 1
(1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 2
(1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 3
(1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 4
(1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 2
Nuevo valor para valUno: 2
Nuevo valor para valDos: 3
x: 2 y: 3
x: 4 y: 9
x: 64 y: 729
x: 729 y: 64
x: 531441 y:4096

```

**ANÁLISIS**

Una vez más se ha omitido la implementación de las funciones para ahorrar espacio, pero es la misma que la del listado 14.5. En la línea 16 se declara el arreglo `apFuncArreglo` como un arreglo de cinco apuntadores a funciones que regresan `void` y que toman dos referencias de tipo entero.

En las líneas 18 a 30 se pide al usuario que elija las funciones a invocar, y a cada miembro del arreglo se le asigna la dirección de la función apropiada. En las líneas 32 a 38 se invoca una por una cada función seleccionada por el usuario. El resultado se imprime después de cada invocación.

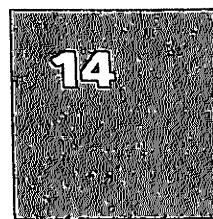
## Paso de apuntadores a funciones hacia otras funciones

Los apuntadores a funciones (y los arreglos de apuntadores a funciones) se pueden pasar a otras funciones, las cuales pueden realizar cierta acción y luego llamar a la función apropiada por medio del apuntador.

Por ejemplo, el listado 14.5 se podría mejorar pasando el apuntador a función elegido a otra función (fuera de `main()`), que imprima los valores, invoque a la función y luego imprima otra vez los valores. El listado 14.8 muestra esta variación.

**ENTRADA** **LISTADO 14.8** Paso de apuntadores a funciones como argumentos de una función

```
1: // Listado 14.8 Sin apuntadores a funciones
2:
3: #include <iostream.h>
4:
5: void Cuadrado (int&,int&);
6: void Cubo (int&, int&);
7: void Intercambiar (int&, int &);
8: void ObtenerValores(int&, int&);
9: void ImprimirValores(void (*)(int&, int&),int&, int&);
10:
11: int main()
12: {
13:     int valUno=1, valDos=2;
14:     int opcion;
15:     bool fSalir = false;
16:
17:     void (*apFunc)(int&, int&);
18:
19:     while (fSalir == false)
20:     {
21:         cout << "(0)Salir (1)Cambiar Valores (2)Cuadrado
22: =>(3)Cubo (4)Intercambiar: ";
23:         cin >> opcion;
24:         switch (opcion)
25:         {
26:             case 1:apFunc = ObtenerValores; break;
27:             case 2:apFunc = Cuadrado; break;
28:             case 3:apFunc = Cubo; break;
29:             case 4:apFunc = Intercambiar; break;
30:             default:fSalir = true; break;
31:         }
32:         if (fSalir == true)
33:             break;
34:         ImprimirValores (apFunc, valUno, valDos);
35:     }
36:
37:     return 0;
38:
39: void ImprimirValores(void (*apFunc)(int&, int&),int& x, int& y)
40: {
41:     cout << "x: " << x << " y: " << y << endl;
42:     apFunc(x,y);
```



**LISTADO 14.8** CONTINUACIÓN

```
43:         cout << "x: " << x << " y: " << y << endl;
44:     }
45:
46: void Cuadrado (int & rX, int & rY)
47: {
48:     rX *= rX;
49:     rY *= rY;
50: }
51:
52: void Cubo (int & rX, int & rY)
53: {
54:     int tmp;
55:
56:     tmp = rX;
57:     rX *= rX;
58:     rX = rX * tmp;
59:
60:     tmp = rY;
61:     rY *= rY;
62:     rY = rY * tmp;
63: }
64:
65: void Intercambiar(int & rX, int & rY)
66: {
67:     int temp;
68:     temp = rX;
69:     rX = rY;
70:     rY = temp;
71: }
72:
73: void ObtenerValores (int & rValUno, int & rValDos)
74: {
75:     cout << "Nuevo valor para valUno: ";
76:     cin >> rValUno;
77:     cout << "Nuevo valor para valDos: ";
78:     cin >> rValDos;
79: }
```

---

**SALIDA**

```
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 1
x: 1 y: 2
Nuevo valor para valUno: 2
Nuevo valor para valDos: 3
x: 2 y: 3
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 3
x: 2 y: 3
x: 8 y: 27
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 2
x: 8 y: 27
```

```

x: 64 y: 729
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 4
x: 64 y: 729
x: 729 y:64
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 0

```

**ANÁLISIS**

En la línea 17 se declara a `apFunc` como un apuntador a una función que regresa `void` y toma dos parámetros, ambos referencias a enteros. En la línea 9 se declara a `ImprimirValores` como una función que toma tres parámetros. El primero es un apuntador a una función que regresa `void` pero que toma como parámetros dos referencias a enteros, y el segundo y tercer argumentos para `ImprimirValores` son referencias a enteros. De nuevo se piden al usuario las funciones que se van a llamar, y luego se llama a `ImprimirValores` en la línea 33.

Una buena manera de probar el nivel de conocimiento de un programador de C++ es preguntarle lo que significa la siguiente declaración:

```
void ImprimirValores(void (*)(int&, int&),int&, int&);
```

Este tipo de declaración se utiliza con poca frecuencia y probablemente tenga que consultar este libro cada vez que la necesite, pero salvará su programa en esas raras ocasiones en que requiera esta construcción.

## Uso de `typedef` con apuntadores a funciones

La construcción `void (*)(int&, int&)` es, en el mejor de los casos, incómoda. Puede utilizar `typedef` para simplificar esto, al declarar un tipo (en este caso se llama `VAF`) como apuntador a una función que regresa `void` y que toma como parámetros dos referencias a enteros. El listado 14.9 reproduce el listado 14.8, pero utilizando esta instrucción `typedef`.

**ENTRADA****LISTADO 14.9** Uso de `typedef` para hacer más legibles los apuntadores a funciones

```

1: // Listado 14.9. Uso de typedef para hacer más legibles los apuntadores a
  ↵funciones
2:
3: #include <iostream.h>
4:
5: void Cuadrado (int&,int&);
6: void Cubo (int&, int&);
7: void Intercambiar (int&, int&);
8: void ObtenerValores(int&, int&);
9: typedef void (*VAF) (int&, int&);
10: void ImprimirValores(VAF,int&, int&);
11:
12: int main()
13: {
14:     int valUno=1, valDos=2;

```

**LISTADO 14.9 CONTINUACIÓN**

```
15: int opcion;
16: bool fSalir = false;
17:
18: VAF apFunc;
19:
20: while (fSalir == false)
21: {
22: cout << "(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar:
23: ";
24: cin >> opcion;
25: switch (opcion)
26: {
27: case 1:apFunc = ObtenerValores; break;
28: case 2:apFunc = Cuadrado; break;
29: case 3:apFunc = Cubo; break;
30: case 4:apFunc = Intercambiar; break;
31: default:fSalir = true; break;
32: }
33: if (fSalir == true)
34: break;
35: ImprimirValores (apFunc, valUno, valDos);
36:
37: return 0;
38:
39: void ImprimirValores(VAF apFunc,int& x, int& y)
40: {
41: cout << "x: " << x << " y: " << y << endl;
42: apFunc(x,y);
43: cout << "x: " << x << " y: " << y << endl;
44: }
45:
46: void Cuadrado (int & rX, int & rY)
47: {
48: rX *= rX;
49: rY *= rY;
50: }
51:
52: void Cubo (int & rX, int & rY)
53: {
54: int tmp;
55:
56: tmp = rX;
57: rX *= rX;
58: rX = rX * tmp;
59:
60: tmp = rY;
61: rY *= rY;
62: rY = rY * tmp;
63: }
```

```

64:
65:     void Intercambiar(int & rX, int & rY)
66:     {
67:         int temp;
68:         temp = rX;
69:         rX = rY;
70:         rY = temp;
71:     }
72:
73:     void ObtenerValores (int & rValUno, int & rValDos)
74:     {
75:         cout << "Nuevo valor para valUno: ";
76:         cin >> rValUno;
77:         cout << "Nuevo valor para valDos: ";
78:         cin >> rValDos;
79:     }

```

**SALIDA**

```

(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 1
x: 1 y: 2
Nuevo valor para valUno: 2
Nuevo valor para valDos: 3
x: 2 y: 3
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 3
x: 2 y: 3
x: 8 y: 27
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 2
x: 8 y: 27
x: 64 y: 729
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 4
x: 64 y: 729
x: 729 y: 64
(0)Salir (1)Cambiar Valores (2)Cuadrado (3)Cubo (4)Intercambiar: 0

```

**ANÁLISIS**

En la línea 9 se utiliza `typedef` para declarar a `VAF` como tipo “apuntador a función que regresa void y que toma dos parámetros, ambos referencias a enteros”.

En la línea 10 se declara la función `ImprimirValores()` para tomar tres parámetros: un `VAF` y dos referencias a enteros. En la línea 18 `apFunc` se declara ahora como tipo `VAF`.

Después que se define el tipo `VAF`, todos los usos subsecuentes para declarar a `apFunc` y a `ImprimirValores()` son más limpios. Como puede ver, la salida es idéntica.

## Apuntadores a funciones miembro

Hasta este punto, todos los apuntadores a funciones que se han creado han sido para funciones generales que no pertenecen a una clase. También es posible crear apuntadores a funciones que sean miembros de clases.

Para crear un apuntador a una función miembro, se utiliza la misma sintaxis que para un apuntador a una función, pero se incluye el nombre de la clase y el operador de resolución de ámbito (::). Así que, si apFunc apunta a una función miembro de la clase Figura, la cual toma dos parámetros enteros y regresa void, la declaración para apFunc es la siguiente:

```
void (Figura::*apFunc) (int, int);
```

Los apuntadores a funciones miembro se utilizan de la misma forma que los apuntadores a funciones, excepto que se requiere de un objeto de la clase correcta para poder invocarlos. El listado 14.10 muestra el uso de apuntadores a funciones miembro.

**ENTRADA LISTADO 14.10 Apuntadores a funciones miembro**

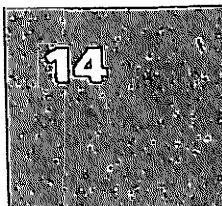
---

```
1:      //Listado 14.10 Apuntadores a funciones miembro que utilizan métodos
2:      ↵virtuales
3:      #include <iostream.h>
4:
5:      class Mamifero
6:      {
7:      public:
8:          Mamifero():suEdad(1) { }
9:          virtual ~Mamifero() { }
10:         virtual void Hablar() const = 0;
11:         virtual void Mover() const = 0;
12:     protected:
13:         int suEdad;
14:     };
15:
16:     class Perro : public Mamifero
17:     {
18:     public:
19:         void Hablar()const { cout << "¡Guau!\n"; }
20:         void Mover() const { cout << "Caminando hacia el amo...\n"; }
21:     };
22:
23:
24:     class Gato : public Mamifero
25:     {
26:     public:
27:         void Hablar()const { cout << "¡Miau!\n"; }
28:         void Mover() const { cout << "caminando sigilosamente...\n"; }
29:     };
30:
31:
32:     class Caballo : public Mamifero
33:     {
34:     public:
35:         void Hablar()const { cout << "¡Yihii!\n"; }
```

```
36:     void Mover() const { cout << "Galopando...\n"; }
37: }
38:
39:
40: int main()
41: {
42:     void (Mamifero::*apFunc)() const =0;
43:     Mamifero* aptr =NULL;
44:     int Animal;
45:     int Metodo;
46:     bool fSalir = false;
47:
48:     while (fSalir == false)
49:     {
50:         cout << "(0)Salir (1)perro (2)gato (3)caballo: ";
51:         cin >> Animal;
52:         switch (Animal)
53:         {
54:             case 1: aptr = new Perro; break;
55:             case 2: aptr = new Gato; break;
56:             case 3: aptr = new Caballo; break;
57:             default: fSalir = true; break;
58:         }
59:         if (fSalir)
60:             break;
61:
62:         cout << "(1)Hablar (2)Mover: ";
63:         cin >> Metodo;
64:         switch (Metodo)
65:         {
66:             case 1: apFunc = Mamifero::Hablar; break;
67:             default: apFunc = Mamifero::Mover; break;
68:         }
69:
70:         (aptr->*apFunc)();
71:         delete aptr;
72:     }
73:     return 0;
74: }
```

**SALIDA**

```
(0)Salir (1)perro (2)gato (3)caballo: 1
(1)Hablar (2)Mover: 1
¡Guau!
(0)Salir (1)perro (2)gato (3)caballo: 2
(1)Hablar (2)Mover: 1
¡Miau!
(0)Salir (1)perro (2)gato (3)caballo: 3
(1)Hablar (2)Mover: 2
Galopando...
(0)Salir (1)perro (2)gato (3)caballo: 0
```



**ANÁLISIS**

En las líneas 5 a 14 se declara el tipo de datos abstracto Mamífero con dos métodos virtuales puros, Hablar() y Mover(). Mamífero se divide en las subclases Perro, Gato y Caballo, cada una de las cuales redefine a Hablar() y a Mover().

El programa controlador de main() pide al usuario que elija el tipo de animal que se va a crear, y luego se crea una subclase de Animal en el heap y se asigna a aptr en las líneas 54 a 56.

Luego se pide al usuario el método a invocar, y ese método se asigna al apuntador apFunc en las líneas 66 o 67. La versión 2.7.2 de g++ compila esto sin problemas; la versión 2.9.5 emite las siguientes advertencias (debido a que se están pasando direcciones):

```
./lst14-10.cxx: In function 'int main()':
./lst14-10.cxx:66: warning: assuming & on `Mamifero::Hablar() const'
./lst14-10.cxx:67: warning: assuming & on `Mamifero::Mover() const'
```

En la línea 70, el objeto creado invoca al método elegido mediante el apuntador aptr para tener acceso al objeto y mediante apFunc para tener acceso a la función.

Finalmente, en la línea 71 se llama a delete en el apuntador aptr para regresar al heap la memoria reservada para el objeto. Observe que no hay razón para utilizar delete sobre apFunc ya que éste es un apuntador al código, no un a un objeto en el heap. De hecho, si se intenta hacer esto se generará un error en tiempo de compilación.

## Arreglos de apuntadores a funciones miembro

Al igual que los apuntadores a funciones, los apuntadores a funciones miembro se pueden guardar en un arreglo. El arreglo se puede inicializar con las direcciones de varias funciones miembro, y éstas se pueden invocar por medio de desplazamientos en el arreglo. El listado 14.11 muestra esta técnica.

**ENTRADA****LISTADO 14.11** Arreglo de apuntadores a funciones miembro

```
1: //Listado 14.11 Arreglo de apuntadores a funciones miembro
2:
3: #include <iostream.h>
4:
5: class Perro
6: {
7: public:
8:     void Hablar()const { cout << "¡Guau!\n"; }
9:     void Mover() const { cout << "Caminando hacia el amo...\n"; }
10:    void Comer() const { cout << "Devorando la comida...\n"; }
11:    void Grunir() const { cout << "Grrrrr\n"; }
12:    void Gimotear() const { cout << "Sonidos de gimoteos...\n"; }
13:    void DarVuelta() const { cout << "Dando vuelta...\n"; }
14:    void HacerMuerto() const { cout << "¿Es éste el final del pequeño
➥César?\n"; }
```

```
15:     };
16:
17:     typedef void (Perro::*AFM)()const ;
18:     int main()
19:     {
20:         const int MaxFuncs = 7;
21:         AFM PerroFunciones[MaxFuncs] =
22:             { Perro::Hablar,
23:                 Perro::Mover,
24:                 Perro::Comer,
25:                 Perro::Grunir,
26:                 Perro::Gimotear,
27:                 Perro::DarVuelta,
28:                 Perro::HacerMuerto };
29:
30:         Perro* apPerro =NULL;
31:         int Metodo;
32:         bool fSalir = false;
33:
34:         while (!fSalir)
35:         {
36:             cout << " (0)Salir (1)Hablar (2)Mover (3)Comer (4)Grunir";
37:             cout << " (5)Gimotear (6)Dar vuelta (7)Hacerse el muerto: ";
38:             cin >> Metodo;
39:             if (Metodo == 0)
40:             {
41:                 fSalir = true;
42:             }
43:             else
44:             {
45:                 apPerro = new Perro;
46:                 (apPerro->*PerroFunciones[Metodo-1])();
47:                 delete apPerro;
48:             }
49:         }
50:         return 0;
51:     }
```

**SALIDA**

```
(0)Salir (1)Hablar (2)Mover (3)Comer (4)Grunir (5)Gimotear (6)Dar
vuelta (7)Hacerse el muerto: 1
¡Guau!
(0)Salir (1)Hablar (2)Mover (3)Comer (4)Grunir (5)Gimotear (6)Dar
vuelta (7)Hacerse el muerto: 4
Grrrrr
(0)Salir (1)Hablar (2)Mover (3)Comer (4)Grunir (5)Gimotear (6)Dar
vuelta (7)Hacerse el muerto: 7
¿Es éste el final del pequeño César?
(0)Salir (1)Hablar (2)Mover (3)Comer (4)Grunir (5)Gimotear (6)Dar
vuelta (7)Hacerse el muerto: 0
```

**ANÁLISIS**

En las líneas 5 a 15 se crea la clase `Perro`, la cual tiene siete funciones miembro, y todas comparten el mismo tipo de valor de retorno y la misma firma. En la línea 17, una instrucción `typedef` declara a `AFM` como apuntador a una función miembro de `Perro` que no lleva parámetros y no regresa valores, y que es `const`, es decir, la firma de las siete funciones miembro de `Perro`.

En las líneas 21 a 28 se declara el arreglo `PerroFunciones` para guardar esas siete funciones miembro, y se inicializa con las direcciones de dichas funciones. Igual que en el listado 14.10, la versión 2.7.2 de g++ no tuvo problemas con este código; la versión 2.9.5 produjo los siguientes mensajes:

```
./lst14-11.cxx: In function 'int main()':
./lst14-11.cxx:28: warning: assuming & on 'Perro::Hablar() const'
./lst14-11.cxx:28: warning: assuming & on 'Perro::Mover() const'
./lst14-11.cxx:28: warning: assuming & on 'Perro::Comer() const'
./lst14-11.cxx:28: warning: assuming & on 'Perro::Grunir() const'
./lst14-11.cxx:28: warning: assuming & on 'Perro::Gimotear() const'
./lst14-11.cxx:28: warning: assuming & on 'Perro::DarVuelta() const'
./lst14-11.cxx:28: warning: assuming & on 'Perro::HacerMuerto() const'
```

En las líneas 36 y 37 se pide al usuario que elija un método. A menos que elija `Salir`, se crea un nuevo `Perro` en el heap y luego se invoca el método correcto en el arreglo de la línea 46. Ésta es otra buena línea para mostrar a los programadores brillantes de C++ de su compañía; pregúntales qué es lo que hace:

```
(apPerro->*PerroFunciones[Metodo-1])();
```

¿Qué cree usted que pasaría si se escribiera un valor fuera de rango (como -1 u 8)? Como C++ no tiene forma de verificar los límites de los arreglos, no obtendría un error de compilación o un error en tiempo de ejecución. Podría obtener resultados completamente inesperados si su programa intenta utilizar la dirección almacenada en esa memoria como una función. Como no estableció un valor en esa memoria (como se hizo en las líneas 21 a 28), no tiene idea de lo que hay ahí. Tal vez haya un valor numérico que será tratado como si fuera la dirección de una función, con resultados impredecibles.

Una vez más, esto es un poco esotérico, pero cuando necesite una tabla construida a partir de funciones miembro, esto puede ayudar a que su programa sea más legible.

**DEBE**

**DEBE** invocar apuntadores a funciones miembro en un objeto específico de una clase.

**DEBE** utilizar `typedef` para que las declaraciones de apuntador a función miembro sean más legibles.

**No DEBE**

**NO DEBE** utilizar apuntadores a funciones miembro cuando se puedan utilizar soluciones más sencillas.

## Resumen

Hoy aprendió cómo crear variables miembro estáticas en su clase. Cada clase, en vez de cada objeto, tiene una instancia de la variable miembro estática. Es posible tener acceso a esta variable sin un objeto del tipo de la clase si se identifica completamente el nombre (asumiendo que haya declarado el miembro estático con acceso público).

Las variables miembro estáticas se pueden utilizar como contadores a través de las instancias de la clase. Como no son parte del objeto, la declaración de variables miembro estáticas no asigna memoria, y éstas se deben definir e inicializar fuera de la declaración de la clase.

Las funciones miembro estáticas son parte de la clase de la misma manera que lo son las variables miembro estáticas. Puede tener acceso a ellas sin un objeto específico de la clase y puede utilizar para tener acceso a los datos miembro estáticos. Las funciones miembro estáticas no se pueden utilizar para tener acceso a datos miembro que no sean estáticos, ya que no tienen un apuntador `this`.

Como las funciones miembro estáticas no tienen un apuntador `this`, tampoco se pueden hacer `const`. La palabra reservada `const` en una función miembro indica que el apuntador `this` es `const`.

También aprendió cómo declarar y utilizar apuntadores a funciones y apuntadores a funciones miembro. Vio cómo crear arreglos de estos apuntadores y cómo pasarlos a las funciones.

Los apuntadores a funciones y los apuntadores a funciones miembro se pueden utilizar para crear tablas de funciones que se pueden seleccionar en tiempo de ejecución. Esto puede dar flexibilidad a su programa, lo cual no se logra fácilmente sin estos apuntadores.

## Preguntas y respuestas

**P ¿Por qué utilizar datos estáticos si se pueden utilizar datos globales?**

**R** Los datos estáticos tienen alcance sólo dentro de la clase. De esta forma, los datos estáticos están disponibles sólo mediante un miembro de la clase, mediante una llamada explícita que utilice el nombre de clase si éste es público, o mediante el uso de una función miembro estática. Sin embargo, los datos estáticos están tipificados con el tipo de la clase, y el acceso restringido y la fuerte tipificación hacen que los datos estáticos sean más seguros que los datos globales.

**P ¿Por qué utilizar funciones miembro estáticas si se pueden utilizar funciones globales?**

**R** Las funciones miembro estáticas tienen alcance sólo dentro de la clase y sólo se pueden llamar mediante el uso de un objeto de la clase o mediante una especificación explícita completa (por ejemplo, `NombreClase::NombreFuncion()`).

**P ¿Es común utilizar muchos apuntadores a funciones y apuntadores a funciones miembro?**

**R** No, éstos tienen sus usos especiales, pero no son construcciones comunes. Muchos programas complejos y poderosos no tienen.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

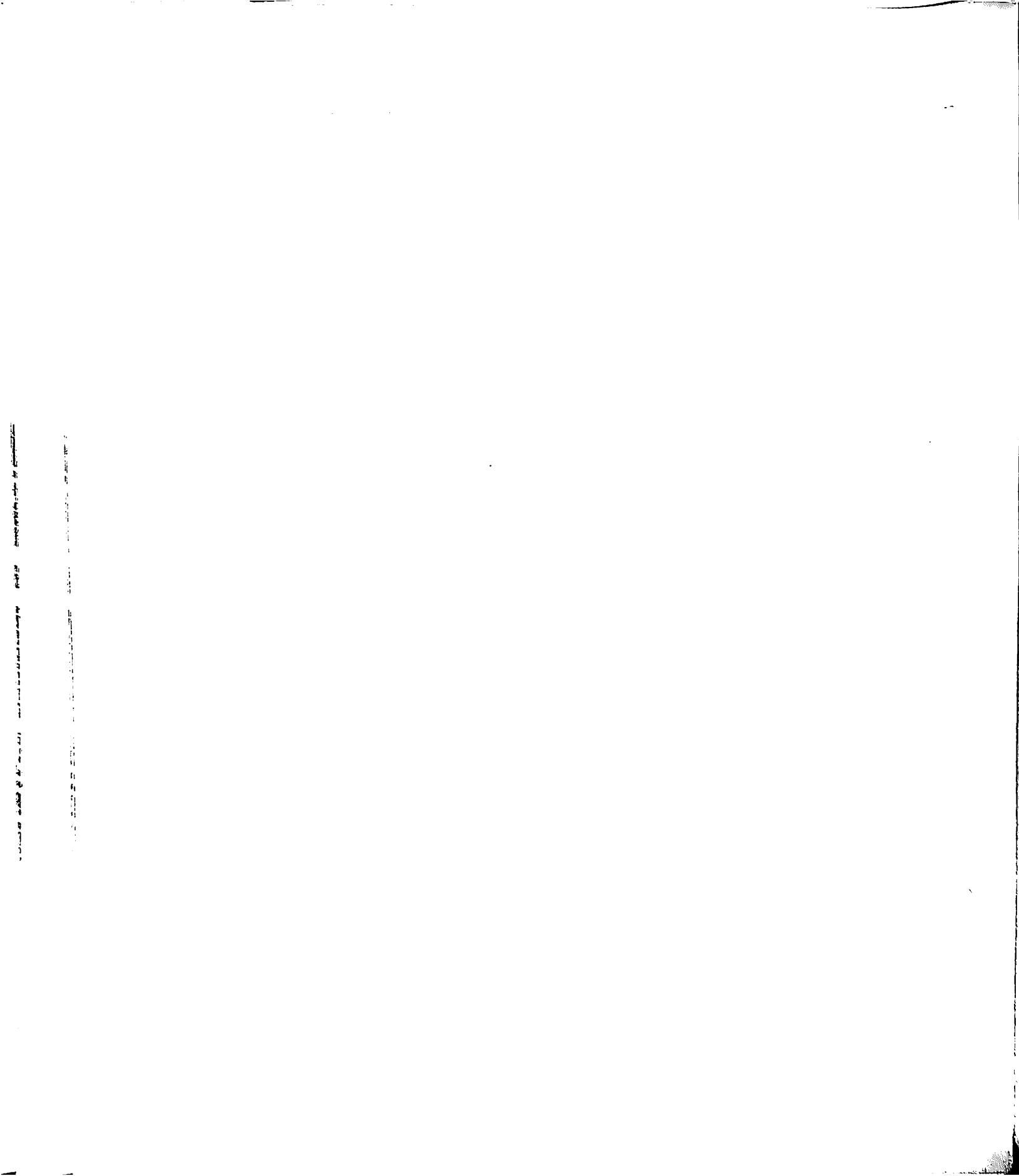
### Cuestionario

1. ¿Pueden las variables miembro estáticas ser privadas?
2. Muestre la declaración de una variable miembro estática.
3. Muestre la declaración de una función estática.
4. Muestre la declaración de un apuntador a una función que regrese un `long` y que tome un parámetro entero.
5. Modifique el apuntador de la pregunta 4 para que sea un apuntador a una función miembro de la clase `Auto`.
6. Muestre la declaración de un arreglo de 10 apuntadores como los de la pregunta 5.

### Ejercicios

1. Escriba un programa corto que declare una clase con una variable miembro y una variable miembro estática. Haga que el constructor inicialice la variable miembro e incremente la variable miembro estática. Haga que el destructor decremente la variable miembro estática.
2. Usando el programa del ejercicio 1, escriba un programa controlador corto que cree tres objetos y luego despliegue sus variables miembro y la variable miembro estática. Luego destruya cada objeto y muestre el efecto en la variable miembro estática.
3. Modifique el programa del ejercicio 2 para utilizar una función miembro estática que permita el acceso a la variable miembro estática. Haga que la variable miembro estática sea privada.

4. Escriba un apuntador a una función miembro para que tenga acceso a los datos miembro que no sean estáticos del programa del ejercicio 3, y utilice ese apuntador para imprimir el valor de esos datos.
5. Agregue dos variables miembro más a la clase de las preguntas anteriores. Agregue métodos de acceso que obtengan el valor de estos valores y proporcionen a todas las funciones miembro los mismos valores de retorno y firmas. Utilice el apuntador a una función miembro para tener acceso a estas funciones.



# SEMANA 2

## Repasso

El programa de repaso de la semana 2 reúne muchas de las habilidades que usted adquirió durante las últimas dos semanas y produce un programa poderoso.

Esta demostración de listas enlazadas utiliza funciones virtuales, funciones virtuales puras, redefinición de funciones, polimorfismo, herencia pública, sobrecarga de funciones, ciclos infinitos, apuntadores, referencias y más. Observe que éste es un tipo distinto de lista enlazada del que se muestra en el día 12, "Arreglos, cadenas tipo C y listas enlazadas"; en C++ hay muchas formas de lograr lo mismo.

El objetivo de este programa es crear una lista enlazada. Los nodos de la lista están diseñados para guardar piezas, como podría usarse en una fábrica. Aunque ésta no es la forma final del programa, hace una buena demostración de una estructura de datos bastante avanzada. El listado R2.1 tiene 289 líneas. Trate de analizar el código por su cuenta antes de leer el análisis que se encuentra después de la salida.

**ENTRADA****LISTADO R2.1** Listado de repaso de la semana 2

```
1:      //  
*****  
2:      //  
3:      // Título:          Revisión de la semana 2  
4:      //  
5:      // Archivo:         Semana2  
6:      //  
7:      // Descripción: Proporcionar un programa de  
→demostración de listas enlazadas  
8:      //  
9:      // Clases:          Pieza - guarda números de  
→pieza y potencialmente cualquier otra  
10:     //                  información relacionada  
→con las piezas
```

8

9

10

11

12

13

14

*continúa*

**LISTADO R2.1** CONTINUACIÓN

```
11: //  
12: // NodoPieza - actúa como nodo en una ListaPiezas  
13: //  
14: // ListaPiezas - provee los mecanismos para una lista  
    //→enlazada  
15: //  
16: //  
17: // ****  
18:  
19: #include <iostream.h>  
20:  
21:  
22:  
23: // ***** Pieza *****  
24: // Clase base abstracta de piezas  
25: class Pieza  
26: {  
27: public:  
28:     Pieza() : suNumeroPieza(1) {}  
29:     Pieza(int NumeroPieza) : suNumeroPieza(NumeroPieza) {}  
30:     virtual ~Pieza() {};  
31:     int ObtenerNumeroPieza() const { return suNumeroPieza; }  
32:     virtual void Desplegar() const = 0; // debe redefinirse  
33: private:  
34:     int suNumeroPieza;  
35: };  
36:  
37: // implementación de la función virtual pura para que  
38: // las clases derivadas puedan encadenarse  
39: void Pieza::Desplegar() const  
40: {  
41:     cout << "\nNúmero de pieza: " << suNumeroPieza << endl;  
42: }  
43:  
44: // ***** Pieza de Auto *****  
45: class PiezaAuto : public Pieza  
46: {  
47: public:  
48:     PiezaAuto() : suAnioModelo(94) {}  
49:     PiezaAuto(int anio, int numeroPieza);  
50:     virtual void Desplegar() const  
51:     {  
52:         Pieza::Desplegar(); cout << "Año del modelo: ";  
53:         cout << suAnioModelo << endl;  
54:     }  
55: private:
```

```
56:         int suAnioModelo;
57:     };
58:
59:     PiezaAuto::PiezaAuto(int anio, int numeroPieza):
60:         suAnioModelo(anio),
61:         Pieza(numeroPieza)
62:     {}
63:
64:
65: // ***** Pieza de AeroPlano *****
66: class PiezaAeroPlano : public Pieza
67: {
68: public:
69:     PiezaAeroPlano() : suNumeroMotor(1) {};
70:     PiezaAeroPlano(int NumeroMotor, int NumeroPieza);
71:     virtual void Desplegar() const
72:     {
73:         Pieza::Desplegar(); cout << "Motor número.: ";
74:         cout << suNumeroMotor << endl;
75:     }
76: private:
77:     int suNumeroMotor;
78: };
79:
80: PiezaAeroPlano::PiezaAeroPlano(int NumeroMotor, int NumeroPieza):
81:     suNumeroMotor(NumeroMotor),
82:     Pieza(NumeroPieza)
83: {}
84:
85: // ***** Nodo de Pieza *****
86: class NodoPieza
87: {
88: public:
89:     NodoPieza(Pieza *);
90:     ~NodoPieza();
91:     void AsignarSiguiente(NodoPieza * nodo) { suSiguiente = nodo; }
92:     NodoPieza * ObtenerSiguiente() const;
93:     Pieza * ObtenerPieza() const;
94: private:
95:     Pieza * suPieza;
96:     NodoPieza * suSiguiente;
97: };
98:
99: // Implementaciones de NodoPieza...
100: NodoPieza::NodoPieza(Pieza * apPieza):
101:     suPieza(apPieza),
102:     suSiguiente(0)
103: {}
```

*continúa*

**LISTADO R2.1 CONTINUACIÓN**

```
104:  
105:     NodoPieza::~NodoPieza()  
106:     {  
107:         delete suPieza;  
108:         suPieza = 0;  
109:         delete suSiguiente;  
110:         suSiguiente = 0;  
111:     }  
112:  
113:     // Regresa NULL si no hay siguiente NodoPieza  
114:     NodoPieza * NodoPieza::ObtenerSiguiente() const  
115:     {  
116:         return suSiguiente;  
117:     }  
118:  
119:     Pieza * NodoPieza::ObtenerPieza() const  
120:     {  
121:         if (suPieza)  
122:             return suPieza;  
123:         else  
124:             return NULL; //error  
125:     }  
126:  
127:     // ***** Lista de Piezas *****  
128: class ListaPiezas  
129: {  
130: public:  
131:     ListaPiezas();  
132:     ~ListaPiezas();  
133:     // Inesusta constructor de copia y operador igual a!  
134:     Pieza * Encontrar(int & posicion, int NumeroPieza) const;  
135:     int ObtenerCuenta() const { return suCuenta; }  
136:     Pieza * ObtenerPrimero() const;  
137:     static ListaPiezas & ObtenerListaPiezasGlobal()  
138:     {  
139:         return ListaPiezasGlobal;  
140:     }  
141:     void Insertar(Pieza *);  
142:     void Iterar(void (Pieza::*f)() const) const;  
143:     Pieza * operator[](int) const;  
144: private:  
145:     NodoPieza * apCabeza;  
146:     int suCuenta;  
147:     static ListaPiezas ListaPiezasGlobal;  
148: };  
149:  
150: ListaPiezas ListaPiezas::ListaPiezasGlobal;  
151:
```

```
152: // Implementaciones para listas...
153:
154: ListaPiezas::ListaPiezas():
155:     apCabeza(0),
156:     suCuenta(0)
157: {}
158:
159: ListaPiezas::~ListaPiezas()
160: {
161:     delete apCabeza;
162: }
163:
164: Pieza * ListaPiezas::ObtenerPrimero() const
165: {
166:     if (apCabeza)
167:         return apCabeza->ObtenerPieza();
168:     else
169:         return NULL; // atrapar error aquí
170: }
171:
172: Pieza * ListaPiezas::operator[](int desFase) const
173: {
174:     NodoPieza* apNodo = apCabeza;
175:
176:     if (!apCabeza)
177:         return NULL; // atrapar error aquí
178:     if (desFase > suCuenta)
179:         return NULL; // error
180:     for (int i = 0; i < desFase; i++)
181:         apNodo = apNodo->ObtenerSiguiente();
182:     return apNodo->ObtenerPieza();
183: }
184:
185: Pieza * ListaPiezas::Encontrar(int & posicion, int NumeroPieza) const
186: {
187:     NodoPieza * apNodo =NULL;
188:
189:     for (apNodo = apCabeza, posicion = 0;
190:          apNodo != NULL;
191:          apNodo = apNodo->ObtenerSiguiente(), posicion++)
192:     {
193:         if (apNodo->ObtenerPieza()->ObtenerNumeroPieza() == NumeroPieza)
194:             break;
195:     }
196:     if (apNodo == NULL)
197:         return NULL;
198:     else
199:         return apNodo->ObtenerPieza();
```

continúa

**LISTADO R2.1** CONTINUACIÓN

```
200:    }
201:
202:    void ListaPiezas::Iterar(void (Pieza::*func)() const) const
203:    {
204:        if (!apCabeza)
205:            return;
206:        NodoPieza * apNodo = apCabeza;
207:        do
208:            (apNodo->ObtenerPieza()->*func)();
209:        while (apNodo = apNodo->ObtenerSiguiente());
210:    }
211:
212:    void ListaPiezas::Insertar(Pieza * apPieza)
213:    {
214:        NodoPieza * apNodo = new NodoPieza(apPieza);
215:        NodoPieza * apActual = apCabeza;
216:        NodoPieza * apSiguiente = NULL;
217:        int Nuevo = apPieza->ObtenerNumeroPieza();
218:        int Siguiente = 0;
219:
220:        suCuenta++;
221:        if (!apCabeza)
222:        {
223:            apCabeza = apNodo;
224:            return;
225:        }
226:        // si éste es más pequeño que el nodo cabeza,
227:        // se convierte en el nuevo nodo cabeza
228:        if (apCabeza->ObtenerPieza()->ObtenerNumeroPieza() > Nuevo)
229:        {
230:            apNodo->AsignarSiguiente(apCabeza);
231:            apCabeza = apNodo;
232:            return;
233:        }
234:        for (;;)
235:        {
236:            // si no hay siguiente, agregar éste
237:            if (!apActual->ObtenerSiguiente())
238:            {
239:                apActual->AsignarSiguiente(apNodo);
240:                return;
241:            }
242:            // si va después de éste y antes del siguiente
243:            // entonces insertarlo aquí, de no ser así, obtener el siguiente
```

```
244:     apSiguiente = apActual->ObtenerSiguiente();
245:     Siguiente = apSiguiente->ObtenerPieza()->ObtenerNumeroPieza();
246:     if (Siguiente > Nuevo)
247:     {
248:         apActual->AsignarSiguiente(apNodo);
249:         apNodo->AsignarSiguiente(apSiguiente);
250:         return;
251:     }
252:     apActual = apSiguiente;
253: }
254:
255:
256: int main()
257: {
258:     ListaPiezas & lp = ListaPiezas::ObtenerListaPiezasGlobal();
259:     Pieza * apPieza = NULL;
260:     int NumeroPieza;
261:     int valor;
262:     int opcion;
263:
264:     while (1)
265:     {
266:         cout << "(0)Salir (1)Auto (2)Avión: ";
267:         cin >> opcion;
268:         if (!opcion)
269:             break;
270:         cout << "¿Nuevo NumeroPieza?: ";
271:         cin >> NumeroPieza;
272:         if (opcion == 1)
273:         {
274:             cout << "¿Año del modelo?: ";
275:             cin >> valor;
276:             apPieza = new PiezaAuto(valor,NumeroPieza);
277:         }
278:         else
279:         {
280:             cout << "¿Número de motor?: ";
281:             cin >> valor;
282:             apPieza = new PiezaAeroPlano(valor, NumeroPieza);
283:         }
284:         lp.Insertar(apPieza);
285:     }
286:     void (Pieza::*apFunc)()const = &Pieza::Desplegar;
287:     lp.Iterar(apFunc);
288:     return 0;
289: }
```

**SALIDA**

```
(0)Salir (1)Auto (2)Avión: 1
¿Nuevo NumeroPieza?: 2837
¿Año del modelo? 90
(0)Salir (1)Auto (2)Avión: 2
¿Nuevo NumeroPieza?: 378
¿Número de motor?: 4938
(0)Salir (1)Auto (2)Avión: 1
¿Nuevo NumeroPieza?: 4499
¿Año del modelo? 94
(0)Salir (1)Auto (2)Avión: 1
¿Nuevo NumeroPieza?: 3000
¿Año del modelo? 93
(0)Salir (1)Auto (2)Avión: 0

Número de pieza: 378
Motor número.: 4938

Número de pieza: 2837
Año del modelo: 90

Número de pieza: 3000
Año del modelo: 93

Número de pieza: 4499
Año del modelo: 94
```

**ANÁLISIS**

El listado R2.1 proporciona la implementación de una lista enlazada para objetos Pieza. Una lista enlazada es una estructura de datos dinámica; es decir, es como un arreglo pero se ajusta su tamaño a medida que se agregan o eliminan objetos.

Esta lista enlazada específica está diseñada para guardar objetos de la clase Pieza, mientras que Pieza es un tipo de datos abstracto que sirve como clase base para cualquier objeto que tenga un número de pieza. En este ejemplo, Pieza se ha dividido en las subclases PiezaAuto y PiezaAeroPlano.

La clase Pieza, que se declara en las líneas 25 a 35, consiste en un número de pieza y algunos métodos de acceso. Probablemente esta clase podría desarrollarse para guardar más información importante sobre las piezas, como los componentes que se utilizan, cuántos hay en existencia, etc. Pieza es un tipo de datos abstracto, reforzado por la función virtual pura Desplegar().

Observe que Desplegar() sí tiene implementación, en las líneas 39 a 42. El propósito del diseñador es obligar a las clases derivadas a crear su propio método Desplegar(), pero también se pueden encadenar con este método.

En las líneas 45 a 57 y 66 a 78 se declaran dos clases derivadas simples, llamadas PiezaAuto y PiezaAeroPlano. Cada una proporciona un método Desplegar() redefinido, el cual hace efectivamente se encadena con el método Desplegar() de la clase base.

La clase `NodoPieza` sirve como interfaz entre la clase `Pieza` y la clase `ListaPiezas`. Contiene un apuntador a una pieza y un apuntador al siguiente nodo de la lista. Sus únicos métodos son obtener y asignar el siguiente nodo en la lista y regresar el objeto `Pieza` al que apunta.

La inteligencia de la lista se encuentra, apropiadamente, en la clase `ListaPiezas`, cuya declaración se encuentra en las líneas 128 a 148. `ListaPiezas` mantiene un apuntador al primer elemento de la lista (`apCabeza`) y lo utiliza para tener acceso a los demás métodos al avanzar por la lista. Avanzar por la lista significa pedir a cada nodo de la lista el siguiente nodo, hasta llegar a un nodo cuyo siguiente apuntador sea `NULL`.

Ésta es sólo una implementación parcial; una lista completamente desarrollada proporcionaría un mayor acceso al primer y último nodos, o proporcionaría un objeto de iteración, el cual permite que los clientes avancen fácilmente por la lista.

`ListaPiezas` proporciona sin duda una variedad de métodos interesantes, los cuales se enlistan en orden alfabético. Esto es a menudo una buena idea, ya que facilita la búsqueda de las funciones.

El método `Encontrar()` toma como argumentos un `NumeroPieza` y un número entero (`int NumeroPieza`). Si encuentra la pieza correspondiente a `NumeroPieza`, regresa un apuntador a esa `Pieza` y asigna al entero la posición de esa pieza dentro de la lista. Si no encuentra a `NumeroPieza`, regresa `NULL`, y la posición no tiene valor significativo.

El método `ObtenerCuenta()` regresa el número de elementos que hay en la lista. `ListaPiezas` mantiene este número como una variable miembro llamada `suCuenta`, aunque podría, desde luego, calcular este número al avanzar por la lista.

Por su parte, el método `ObtenerPrimero()` regresa un apuntador a la primera `Pieza` de la lista, o regresa `NULL` si la lista está vacía.

`ObtenerListaPiezasGlobal()` regresa una referencia a la variable miembro estática `ListaPiezasGlobal`, la cual es una instancia estática de esta clase; todo programa que tiene una `ListaPiezas` también tiene una `ListaPiezasGlobal`, aunque, desde luego, tiene la libertad de crear otras `ListaPiezas` también. Una implementación completa de esta idea modificaría el constructor de `Pieza` para asegurar que cada pieza se cree en `ListaPiezasGlobal`.

`Insertar()` toma un apuntador a un objeto `Pieza`, crea un `NodoPieza` para este objeto, y agrega la `Pieza` a la lista, ordenada por `NumeroPieza`.

`Iterar()` toma un apuntador a una función miembro de `Pieza`, la cual no toma parámetros, regresa `void`, y es `const`. Llama a esa función por cada objeto `Pieza` de la lista. En el programa de muestra se usa en `Desplegar()`, la cual es una función virtual, por lo que se llamará el método `Desplegar()` apropiado con base en el tipo del objeto `Pieza` llamado en tiempo de ejecución.

`Operator[]` permite un acceso directo al objeto `Pieza` que se encuentra en el desplazamiento proporcionado. Se cuenta con una verificación de límites rudimentaria; si la lista es `NULL`, o si el desplazamiento requerido es mayor que el tamaño de la lista, se regresa `NULL` como una condición de error.

Observe que en un programa real, estos comentarios en las funciones se hubieran escrito en la declaración de la clase.

El programa controlador se encuentra en las líneas 256 a 289. En la línea 258 se declara una referencia a `ListaPiezas` y se inicializa con `ListaPiezasGlobal`. Observe que `ListaPiezasGlobal` se inicializa en la línea 150. Esto es necesario debido a que la declaración de una variable miembro estática no la define; la definición debe hacerse afuera de la declaración de la clase.

En las líneas 264 a 285 se pide varias veces al usuario que elija entre escribir una pieza de auto o una pieza de avión. Dependiendo de la opción, se pide el valor apropiado, y se crea la pieza apropiada. Después de esto, la pieza se inserta en la lista en la línea 284.

La implementación del método `Insertar()` de `ListaPiezas` se encuentra en las líneas 212 a 254. Cuando se escribe el primer número de pieza, 2837, se crea un objeto `PiezaAuto` con ese número de pieza y con 90 como año del modelo y se pasa a `ListaPiezas::Insertar()`.

En la línea 214 se crea un nuevo `NodoPieza` con esa pieza, y la variable `Nuevo` se inicializa con el número de pieza. La variable miembro `suCuenta` de `ListaPiezas` se incrementa en la línea 220.

En la línea 221, la prueba de si `apCabeza` es `NULL` resulta verdadera (`true`). Como éste es el primer nodo, es verdadero que el apuntador `apCabeza` de `ListaPiezas` tiene el valor cero. Por lo tanto, en la línea 223 se asigna a `apCabeza` para que apunte al nuevo nodo, y esta función regresa.

Se pide al usuario que escriba una segunda pieza, y esta vez se escribe una pieza de `AeroPlano` con el número de pieza 378 y el número de motor 4938. Una vez más se llama a `ListaPiezas::Insertar()`, y una vez más `apNodo` se inicializa con el nuevo nodo. La variable miembro estática `suCuenta` se incrementa a 2, y se evalúa `apCabeza`. Como la última vez se había asignado el primer nodo a `apCabeza`, ya no tiene el valor `NULL`, por lo que falla la prueba.

En la línea 228 se compara el número de pieza que guarda `apCabeza`, 2837, contra el número de pieza actual, 378. Ya que el nuevo es menor que el que guarda `apCabeza`, el nuevo debe convertirse en el nuevo apuntador a la cabeza, y la prueba de la línea 228 resulta verdadera.

En la línea 230 el nuevo nodo se asigna para apuntar al nodo al que apunta actualmente `apCabeza`. Observe que esto no hace que el nuevo nodo apunte a `apCabeza`, ¡sino al nodo al que `apCabeza` estaba apuntando! En la línea 231, `apCabeza` se asigna para apuntar al nuevo nodo.

La tercera vez que se pasa por el ciclo, el usuario escribe el número de pieza 4499 para un Auto con 94 como año del modelo. El contador se incrementa y esta vez el número no es menor que el número al que apunta `apCabeza`, por lo que se entra al ciclo `for` que empieza en la línea 234.

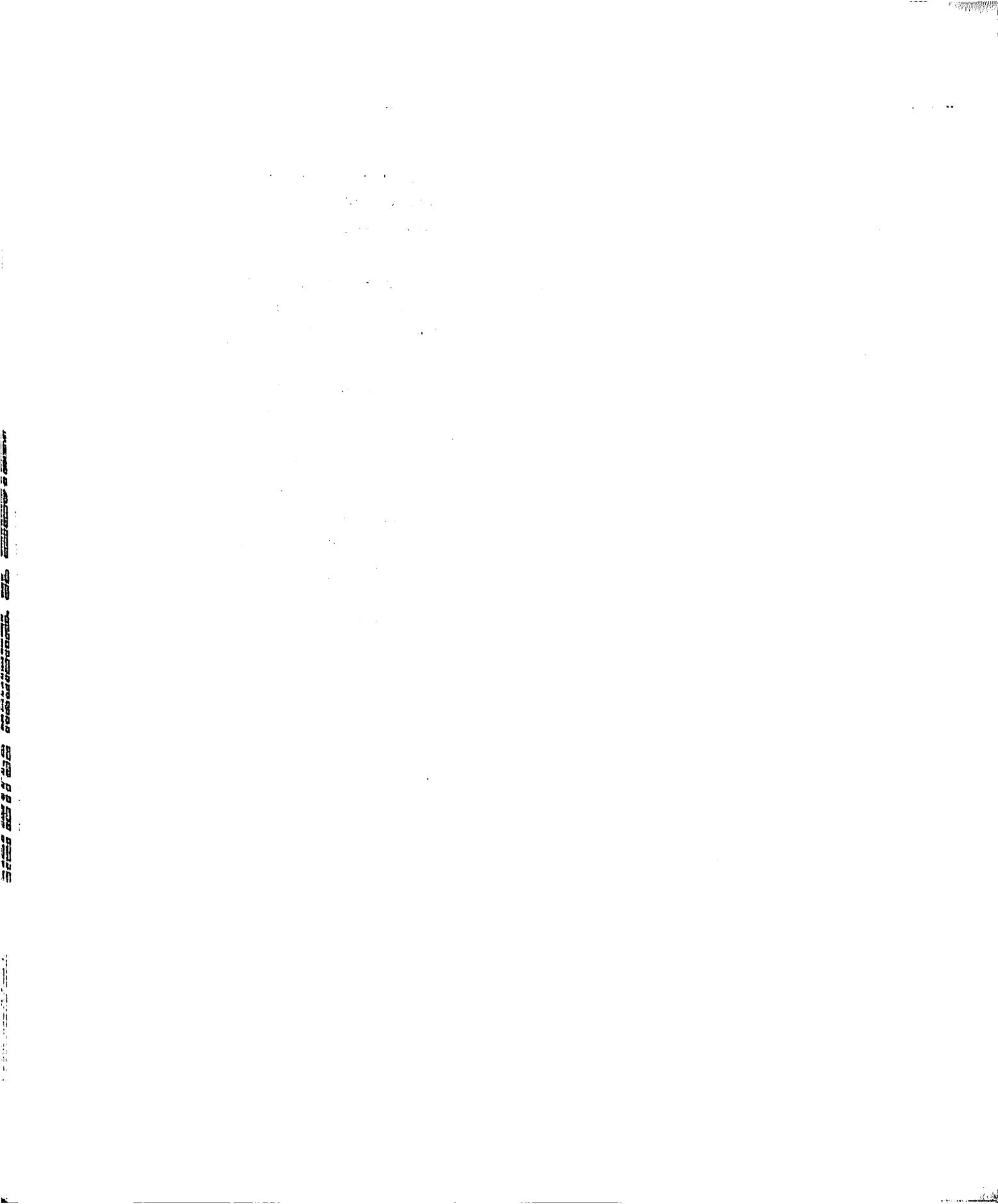
El valor al que apunta `apCabeza` es 378. El valor al que apunta el segundo nodo es 2837. El valor actual es 4499. El apuntador `apActual` apunta al mismo nodo que `apCabeza`, y por lo tanto la variable `Siguiente` tiene un valor diferente de cero; `apActual` apunta al segundo nodo, por lo que la prueba de la línea 237 falla.

El apuntador `apActual` se asigna para apuntar al siguiente nodo, y se repite el ciclo. Esta vez la prueba de la línea 237 tiene éxito. No hay un siguiente elemento, por lo que se indica al nodo actual que apunte al nuevo nodo en la línea 239, y termina la inserción.

La cuarta vez que se pasa por el ciclo, se escribe el número de pieza 3000. Se procede de la misma forma que en la iteración anterior, pero esta vez el nodo actual está apuntando a 2837, y el siguiente nodo tiene 4499, la prueba de la línea 246 resulta `true`, y el nuevo nodo se inserta en su posición.

Cuando el usuario finalmente oprime `0`, la prueba de la línea 268 se evalúa como `true` y se rompe el ciclo `while(1)`. En la línea 286 se asigna la función `Desplegar()` al apuntador a función miembro llamado `apFunc`. En un programa real, esto se asignaría en forma dinámica, con base en el método que elija el usuario.

El apuntador a la función `miembro` se pasa al método `Iterar()` de `ListaPiezas`. En la línea 204 el método `Iterar()` se asegura que la lista no esté vacía. Luego, en las líneas 207 a 209 se llama a cada `Pieza` de la lista por medio del apuntador a la función `miembro`. Esto llama al método `Desplegar()` adecuado para `Pieza`, como se muestra en la salida.



# SEMANA 3

## De un vistazo

Acaba de terminar la segunda semana de aprendizaje de C++. Para estos momentos debe estar familiarizado con algunos de los aspectos más avanzados de la programación orientada a objetos, incluyendo la encapsulación y el polimorfismo.

## Objetivos

Esta última semana regular empieza con una discusión sobre la herencia avanzada. En el día 16, “Flujos”, conocerá con detalle los flujos, y en el día 17, “Espacios de nombres”, aprenderá cómo trabajar con esta excitante adición al estándar de C++. El día 18, “Análisis y diseño orientados a objetos”, es una partida: en lugar de enfocarse en la sintaxis del lenguaje, tomará un día de descanso para conocer el análisis y el diseño orientados a objetos. En el día 19, “Plantillas”, se presentan las plantillas, y en el día 20, “Excepciones y manejo de errores”, se explica lo que son las excepciones. El día 21, “Qué sigue”, el último día regular de este libro, trata sobre algunos temas variados que no se cubren en ninguna otra parte, y luego hay una explicación sobre los siguientes pasos a seguir para convertirse en un gurú de C++.

15

16

17

18

19

20

21



# SEMANA 3

DÍA 15

## Herencia avanzada

Hasta ahora, ha trabajado con herencias simple y múltiple para crear relaciones del tipo *es un*. Hoy aprenderá lo siguiente:

- Qué es la contención y cómo modelarla
- Qué es la delegación y cómo modelarla
- Cómo implementar una clase con base en otra
- Cómo utilizar la herencia privada

### Contención

Como ha visto en ejemplos anteriores, es posible que los datos miembro de una clase incluyan objetos de otra clase. Los programadores de C++ dicen que la clase externa contiene a la clase interna. Por lo tanto, una clase llamada `Empleado` podría contener objetos tales como cadenas (para el nombre del empleado) y enteros (para el salario del empleado, y así sucesivamente).

El listado 15.1 describe una clase `Cadena` incompleta, pero útil. Este listado no produce ninguna salida; en lugar de eso, se utilizará con listados posteriores.

**ENTRADA****LISTADO 15.1 La clase Cadena**

---

```
1: // Listado 15.1: La clase Cadena - lst15-01.hpp
2: //           usada por los listados 15.2, 15.3 y 15.4
3:
4: #include <iostream.h>
5: #include <string.h>
6:
7:
8: class Cadena
9: {
10: public:
11:     // constructores
12:     Cadena();
13:     Cadena(const char * const);
14:     Cadena(const Cadena &);
15:     ~Cadena();
16:     // operadores sobrecargados
17:     char & operator[](int desplazamiento);
18:     char operator[](int desplazamiento) const;
19:     Cadena operator+(const Cadena &);
20:     void operator+=(const Cadena &);
21:     Cadena & operator=(const Cadena &);
22:     // Métodos generales de acceso
23:     int ObtenerLongitud()const { return suLongitud; }
24:     const char * ObtenerCadena() const { return suCadena; }
25:     // static int ConstructorCuenta;
26: private:
27:     Cadena (int); // constructor privado
28:     char * suCadena;
29:     unsigned short suLongitud;
30: };
31:
32: // constructor predeterminado, crea una cadena de 0 bytes
33: Cadena::Cadena()
34: {
35:     suCadena = new char[ 1 ];
36:     suCadena[ 0 ] = '\0';
37:     suLongitud = 0;
38:     // cout << "\tConstructor de cadena predeterminado \n";
39:     // ConstructorCuenta++;
40: }
41:
42: // constructor privado (auxiliar), lo utilizan sólo
43: // los métodos de la clase para crear una nueva cadena del
44: // tamaño requerido. Se llena con caracteres nulos.
45: Cadena::Cadena(int longitud)
46: {
47:     suCadena = new char[ longitud+1 ];
48:     for (int i = 0; i <= longitud; i++)
```

```
49:         suCadena[ i ] = '\0';
50:         suLongitud = longitud;
51:         // cout << "\tConstructor de Cadena(int)\n";
52:         // ConstructorCuenta++;
53:     }
54:
55:     // Convierte un arreglo de caracteres en una Cadena
56:     Cadena::Cadena(const char * const cCadena)
57:     {
58:         suLongitud = strlen(cCadena);
59:         suCadena = new char[ suLongitud+1 ];
60:         for (int i = 0; i < suLongitud; i++)
61:             suCadena[ i ] = cCadena[ i ];
62:         suCadena[ suLongitud ] = '\0';
63:         // cout << "\tConstructor de Cadena(char *) constructor\n";
64:         // ConstructorCuenta++;
65:     }
66:
67:     // constructor de copia
68:     Cadena::Cadena (const Cadena & rhs)
69:     {
70:         suLongitud = rhs.ObtenerLongitud();
71:         suCadena = new char[ suLongitud+1 ];
72:         for (int i = 0; i < suLongitud; i++)
73:             suCadena[ i ] = rhs[ i ];
74:         suCadena[ suLongitud ] = '\0';
75:         // cout << "\tConstructor de Cadena(Cadena &)\n";
76:         // ConstructorCuenta++;
77:     }
78:
79:     // destructor, libera la memoria asignada
80:     Cadena::~Cadena ()
81:     {
82:         delete [] suCadena;
83:         suLongitud = 0;
84:         // cout << "\tDestructor de Cadena\n";
85:     }
86:
87:     // operador igual a, libera la memoria existente
88:     // y luego copia la cadena y el tamaño
89:     Cadena& Cadena::operator=(const Cadena & rhs)
90:     {
91:         if (this == &rhs)
92:             return *this;
93:         delete [] suCadena;
94:         suLongitud = rhs.ObtenerLongitud();
95:         suCadena = new char[ suLongitud+1 ];
96:         for (int i = 0; i < suLongitud; i++)
97:             suCadena[ i ] = rhs[ i ];
```

*continúa*

**LISTADO 15.1** CONTINUACIÓN

```
98:     suCadena[ suLongitud ] = '\0';
99:     // cout << "\tOperador = de Cadena\n";
100:    return * this;
101: }
102:
103: //Operador de desplazamiento no constante, i regresa
104: // una referencia a un carácter para que se pueda
105: // cambiar!
106: char & Cadena::operator[](int desplazamiento)
107: {
108:     if (desplazamiento > suLongitud)
109:         return suCadena[ suLongitud-1 ];
110:     else
111:         return suCadena[ desplazamiento ];
112: }
113:
114: // operador de desplazamiento constante para utilizar
115: // en objetos tipo const (ver constructor de copia)
116: char Cadena::operator[](int desplazamiento) const
117: {
118:     if (desplazamiento > suLongitud)
119:         return suCadena[ suLongitud-1 ];
120:     else
121:         return suCadena[ desplazamiento ];
122: }
123:
124: // crea una nueva cadena agregando la cadena
125: // actual a rhs
126: Cadena Cadena::operator+(const Cadena & rhs)
127: {
128:     int longitudTotal = suLongitud + rhs.ObtenerLongitud();
129:     Cadena temp(longitudTotal);
130:     int i, j;
131:
132:     for (i = 0; i < suLongitud; i++)
133:         temp[ i ] = suCadena[ i ];
134:     for (j = 0; j < rhs.ObtenerLongitud(); j++, i++)
135:         temp[ i ] = rhs[ j ];
136:     temp[ longitudTotal ]='\'0';
137:     return temp;
138: }
139:
140: // cambia la cadena actual, no regresa nada
141: void Cadena::operator+=(const Cadena & rhs)
142: {
143:     unsigned short rhsLong = rhs.ObtenerLongitud();
144:     unsigned short longitudTotal = suLongitud + rhsLong;
145:     Cadena temp(longitudTotal);
146:     int i, j;
```

```

147:
148:     for (i = 0; i < suLongitud; i++)
149:         temp[ i ] = suCadena[ i ];
150:     for (j = 0; j < rhs.ObtenerLongitud(); j++, i++)
151:         temp[ i ] = rhs[ i - suLongitud ];
152:     temp[ longitudTotal ] = '\0';
153:     *this = temp;
154: }
155:
156: // int Cadena::ConstructorCuenta = 0;

```

**SALIDA** Ninguna.

**ANÁLISIS** El listado 15.1 proporciona una clase `Cadena` que es muy similar a la que se utiliza en el listado 12.12 del día 12, “Arreglos, cadenas tipo C y listas enlazadas”. La diferencia considerable aquí es que los constructores y unas cuantas funciones del listado 12.12 tienen instrucciones de impresión para mostrar su uso, las cuales se dejan como comentarios en el listado 15.1. Estas funciones se utilizarán en ejemplos posteriores.

En la línea 25 se declara la variable miembro estática `ConstructorCuenta`, y se inicializa en la línea 156. Esta variable se incrementa en cada constructor de cadena. Todo esto está actualmente como comentarios, los cuales se utilizarán en un listado posterior.

El listado 15.2 describe una clase `Empleado` que contiene tres objetos de tipo cadena.

#### **ENTRADA** LISTADO 15.2 La clase Empleado y el programa controlador

```

1:  // Listado 15.1b - Programa que utiliza el listado 15.1
2:  //                                     con el archivo de encabezado lst15-01.hpp
3:
4:  #include "lst15-01.hpp"
5:
6:
7:  class Empleado
8:  {
9:  public:
10:    Empleado();
11:    Empleado(char *, char *, char *, long);
12:    ~Empleado();
13:    Empleado(const Empleado &);
14:    Empleado & operator= (const Empleado &);
15:    const Cadena & ObtenerPrimerNombre() const
16:        { return suPrimerNombre; }
17:    const Cadena & ObtenerApellido() const
18:        { return suApellido; }
19:    const Cadena & ObtenerDireccion() const
20:        { return suDireccion; }

```

*continúa*

**LISTADO 15.2** CONTINUACIÓN

```
21:     long ObtenerSalario() const
22:         { return suSalario; }
23:     void AsignarPrimerNombre(const Cadena & primNombre)
24:         { suPrimerNombre = primNombre; }
25:     void AsignarApellido(const Cadena & Apellido)
26:         { suApellido = Apellido; }
27:     void AsignarDireccion(const Cadena & direccion)
28:         { suDireccion = direccion; }
29:     void AsignarSalario(long salario)
30:         { suSalario = salario; }
31: private:
32:     Cadena suPrimerNombre;
33:     Cadena suApellido;
34:     Cadena suDireccion;
35:     long suSalario;
36: };
37:
38: Empleado::Empleado():
39:     suPrimerNombre(""),
40:     suApellido(""),
41:     suDireccion(""),
42:     suSalario(0)
43: {}
44:
45: Empleado::Empleado(char * primerNombre, char * apellido,
46:                     char * direccion, long salario):
47:     suPrimerNombre(primerNombre),
48:     suApellido(apellido),
49:     suDireccion(direccion),
50:     suSalario(salario)
51: {}
52:
53: Empleado::Empleado(const Empleado & rhs):
54:     suPrimerNombre(rhs.ObtenerPrimerNombre()),
55:     suApellido(rhs.ObtenerApellido()),
56:     suDireccion(rhs.ObtenerDireccion()),
57:     suSalario(rhs.ObtenerSalario())
58: {}
59:
60: Empleado::~Empleado() {}
61:
62: Empleado & Empleado::operator=(const Empleado & rhs)
63: {
64:     if (this == &rhs)
65:         return *this;
66:     suPrimerNombre = rhs.ObtenerPrimerNombre();
67:     suApellido = rhs.ObtenerApellido();
68:     suDireccion = rhs.ObtenerDireccion();
69:     suSalario = rhs.ObtenerSalario();
70:     return *this;
71: }
72:
```

```
73: int main()
74: {
75:     Empleado Edie("Jane", "Doe", "1461 Shore Parkway", 20000);
76:     Edie.AsignarSalario(50000);
77:     Cadena cApellido("Levine");
78:     Edie.AsignarApellido(cApellido);
79:     Edie.AsignarPrimerNombre("Edythe");
80:     cout << "Nombre: ";
81:     cout << Edie.ObtenerPrimerNombre().ObtenerCadena();
82:     cout << " " << Edie.ObtenerApellido().ObtenerCadena();
83:     cout << ".\nDirección: ";
84:     cout << Edie.ObtenerDireccion().ObtenerCadena();
85:     cout << ".\nSalario: " ;
86:     cout << Edie.ObtenerSalario();
87:     cout << endl;
88:     return 0;
89: }
```

**Nota**

Puede colocar el código del listado 15.1 en un archivo llamado Cadena.hpp. Así, cada vez que necesite la clase Cadena puede incluir el listado 15.1 por medio de `#include "Cadena.hpp"`, en lugar de la línea 4 `#include "lst15-01.hpp"` del listado 15.2.

Por conveniencia en este libro, incluí la implementación con la declaración de la clase. En un programa real, usted guardaría la declaración de la clase en Cadena.hpp y la implementación en Cadena.cpp. Luego compilaría Cadena.cpp para crear el programa objeto Cadena.o, que usaría como una biblioteca dinámica (también puede hacerlo con un archivo make) y utilizaría la instrucción `#include Cadena.hpp` en los programas que utilicen esta biblioteca.

**SALIDA**

Nombre: Edythe Levine.  
Dirección: 1461 Shore Parkway.  
Salario: 50000

**ANÁLISIS**

El listado 15.2 muestra la clase Empleado, la cual contiene tres objetos de tipo cadena: suPrimerNombre, suApellido y suDireccion.

En la línea 75 se crea un objeto Empleado, y se pasan cuatro valores para inicializarlo. En la línea 76 se llama a la función de acceso AsignarSalario() de Empleado, con el valor constante 50000. Hay que tener en cuenta que en un programa real, esto sería un valor dinámico (establecido en tiempo de ejecución) o una constante.

En la línea 77 se crea una cadena y se inicializa con una cadena constante de C++. En la línea 78, este objeto de tipo cadena se utiliza como argumento para AsignarApellido().

En la línea 79 se llama a la función AsignarPrimerNombre() de Empleado con otra cadena constante. Sin embargo, si pone mucha atención, observará que Empleado no tiene una

función `AsignarPrimerNombre()` que tome una cadena de caracteres como argumento; `AsignarPrimerNombre()` requiere una referencia a una cadena constante.

El compilador resuelve esto debido a que sabe cómo crear una cadena a partir de una cadena de caracteres constante. Sabe esto porque le dijo cómo hacerlo en la línea 13 del listado 15.1.

## Cómo tener acceso a miembros de una clase contenida

Los objetos de la clase `Empleado` no tienen acceso especial a las variables miembro de la clase `Cadena`. Si el objeto `Edie` de la clase `Empleado` tratara de tener acceso a la variable miembro `suLongitud` de su propia variable miembro `suPrimerNombre`, se generaría un error en tiempo de compilación. Sin embargo, esto no es un problema grave. Los métodos de acceso proporcionan una interfaz para la clase `Cadena`, y la clase `Empleado` no necesita preocuparse por los detalles de implementación, de la misma forma que no necesita preocuparse por la manera en que la variable entera `suSalario` guarda su información.

## Cómo filtrar el acceso a los miembros contenidos

Observe que la clase `Cadena` proporciona la implementación para soportar el operador`+`. El diseñador de la clase `Empleado` ha bloqueado el acceso al operador`+` que se llama en los objetos `Empleado`, declarando que todos los métodos de acceso de cadena, como `ObtenerPrimerNombre()`, regresen una referencia constante. Debido a que `operator+` no es (y no puede ser) una función `const` (cambia al objeto en el que se llama), si trata de escribir lo siguiente se generará un error en tiempo de compilación:

```
Cadena bufer = Edie.ObtenerPrimerNombre() + Edie.ObtenerApellido();
```

`ObtenerPrimerNombre()` regresa un objeto `Cadena` constante, y no se puede llamar a `operator+` para actuar sobre un objeto constante.

Para solucionar esto, sobrecargue a `ObtenerPrimerNombre()` para que no sea `const`:

```
const Cadena & ObtenerPrimerNombre() const { return suPrimerNombre; }  
Cadena & ObtenerPrimerNombre() { return suPrimerNombre; }
```

Observe que el valor de retorno ya no es `const` y que la propia función miembro ya no es `const`. Cambiar el valor de retorno no es suficiente para sobrecargar el nombre de la función; debe cambiar el estado constante de la propia función.

## El costo de la contención

Es importante observar que el usuario de la clase `Empleado` paga el precio por cada uno de esos objetos de tipo cadena cada vez que se construye uno o que se crea una copia de `Empleado`.

Si se quitan las marcas de comentarios de las instrucciones `cout` del listado 15.1, líneas 38, 51, 63, 75, 84 y 99, se revela la frecuencia con que se llaman. El listado 15.3 modifica el programa controlador agregando instrucciones `cout` para indicar en qué parte del programa se crean los objetos.

**Nota**

Para compilar este listado, quite las marcas de comentario de las líneas 38, 51, 63, 75, 84 y 99 del listado 15.1.

**15****ENTRADA LISTADO 15.3 Constructores de la clase contenida**

```
1: // Listado 15.3 - Otro ejemplo de la clase Empleado
2: // que utiliza la clase Cadena (lst15-01.hpp)
3:
4: #include "lst15-01.hpp"
5:
6:
7: class Empleado
8: {
9: public:
10:    Empleado();
11:    Empleado(char *, char *, char *, long);
12:    ~Empleado();
13:    Empleado(const Empleado &);
14:    Empleado & operator= (const Empleado &);
15:    const Cadena & ObtenerPrimerNombre() const
16:        { return suPrimerNombre; }
17:    const Cadena & ObtenerApellido() const
18:        { return suApellido; }
19:    const Cadena & ObtenerDireccion() const
20:        { return suDireccion; }
21:    long ObtenerSalario() const
22:        { return suSalario; }
23:    void AsignarPrimerNombre(const Cadena & primNombre)
24:        { suPrimerNombre = primNombre; }
25:    void AsignarApellido(const Cadena & Apellido)
26:        { suApellido = Apellido; }
27:    void AsignarDireccion(const Cadena & direccion)
28:        { suDireccion = direccion; }
29:    void AsignarSalario(long salario)
30:        { suSalario = salario; }
31: private:
32:    Cadena suPrimerNombre;
33:    Cadena suApellido;
34:    Cadena suDireccion;
35:    long suSalario;
36: };
37:
38: Empleado::Empleado():
39:     suPrimerNombre(""),
40:     suApellido(""),
41:     suDireccion("")
```

*continúa*

**LISTADO 15.3** CONTINUACIÓN

```
42:     suSalario(0)
43:     {}
44:
45:     Empleado::Empleado(char * primerNombre, char * apellido,
46:                           char * direccion, long salario):
47:         suPrimerNombre(primerNombre),
48:         suApellido(apellido),
49:         suDireccion(direccion),
50:         suSalario(salario)
51:     {}
52:
53:     Empleado::Empleado(const Empleado & rhs):
54:         suPrimerNombre(rhs.ObtenerPrimerNombre()),
55:         suApellido(rhs.ObtenerApellido()),
56:         suDireccion(rhs.ObtenerDireccion()),
57:         suSalario(rhs.ObtenerSalario())
58:     {}
59:
60:     Empleado::~Empleado() {}
61:
62:     Empleado & Empleado::operator= (const Empleado & rhs)
63:     {
64:         if (this == &rhs)
65:             return *this;
66:         suPrimerNombre = rhs.ObtenerPrimerNombre();
67:         suApellido = rhs.ObtenerApellido();
68:         suDireccion = rhs.ObtenerDireccion();
69:         suSalario = rhs.ObtenerSalario();
70:         return *this;
71:     }
72:
73:     int main()
74:     {
75:         cout << "Creando a Edie...\n";
76:         Empleado Edie("Jane", "Doe", "1461 Shore Parkway", 20000);
77:         Edie.AsignarSalario(20000);
78:         cout << "Llamando a AsignarPrimerNombre con char *...\n";
79:         Edie.AsignarPrimerNombre("Edythe");
80:         cout << "Creando cadena cApellido temporal...\n";
81:         Cadena cApellido("Levine");
82:         Edie.AsignarApellido(cApellido);
83:         cout << "Nombre: ";
84:         cout << Edie.ObtenerPrimerNombre().ObtenerCadena();
85:         cout << " " << Edie.ObtenerApellido().ObtenerCadena();
86:         cout << "\nDirección: ";
87:         cout << Edie.ObtenerDireccion().ObtenerCadena();
88:         cout << "\nSalario: ";
89:         cout << Edie.ObtenerSalario();
90:         cout << endl;
91:         return 0;
92:     }
```

**SALIDA**

```

1: Creando a Edie...
2:     Constructor de Cadena(char *) constructor
3:     Constructor de Cadena(char *) constructor
4:     Constructor de Cadena(char *) constructor
5: Llamando a AsignarPrimerNombre con char *...
6:     Constructor de Cadena(char *) constructor
7:     Operador = de Cadena
8: Destructor de Cadena
9:     Creando cadena cApellido temporal...
10:    Constructor de Cadena(char *) constructor
11:    Operador = de Cadena
12: Nombre: Edythe Levine
13: Dirección: 1461 Shore Parkway
14: Salario: 20000
15:     Destructor de Cadena
16:     Destructor de Cadena
17:     Destructor de Cadena
18:     Destructor de Cadena

```

**ANÁLISIS**

El listado 15.3 utiliza las mismas declaraciones de clases que los listados 15.1 y 15.2. Sin embargo, se les han quitado las marcas de comentario a las instrucciones cout. La salida del listado 15.3 se ha numerado para que el análisis sea más claro.

En la línea 75 del listado 15.3 se imprime el enunciado `Creando a Edie...`, como se ve en la línea 1 de la salida. En la línea 76 se crea un objeto Empleado llamado Edie que tiene cuatro parámetros. La salida refleja que el constructor para Cadena se llama tres veces, como era de esperarse.

La línea 78 imprime un enunciado informativo, y en la línea 79 se encuentra la instrucción `Edie.AsignarPrimerNombre("Edythe")`. Esta instrucción ocasiona que se cree una cadena temporal a partir de la cadena de caracteres "Edythe", como se refleja en las líneas 6 y 8 de la salida. Observe que la cadena temporal se destruye inmediatamente después de utilizarla en la instrucción de asignación.

En la línea 81 se crea un objeto de tipo Cadena en el cuerpo del programa. Aquí el programador está haciendo en forma explícita lo que el compilador hizo en forma implícita en la instrucción anterior. Esta vez se ve el constructor en la línea 10 de la salida, pero no se ve el destructor. Este objeto se destruirá hasta que quede fuera de alcance al final de la función.

Al ejecutarse la instrucción `return 0` de la línea 91, se destruyen las cadenas del objeto Empleado al quedar éste fuera de alcance, y la cadena `cApellido` que se creó en la línea 81 también se destruye por quedar fuera de alcance.

## Cómo copiar por valor

El listado 15.3 muestra cómo la creación de un objeto Empleado produjo tres llamadas al constructor de cadena. El listado 15.4 vuelve a utilizar el mismo programa controlador. Esta vez no se utilizan las instrucciones de impresión, pero se quitan las marcas de comentario a la variable miembro estática de tipo cadena llamada `ConstructorCuenta`, y ésta se utiliza.

El análisis del listado 15.1 muestra que `ConstructorCuenta` se incrementa cada vez que se llama a un constructor de cadena. El programa controlador del listado 15.4 llama a las funciones de impresión pasando el objeto `Empleado`, primero por referencia y luego por valor. `ConstructorCuenta` mantiene la cuenta de cuántos objetos tipo cadena se crean cuando el empleado se pasa como parámetro.

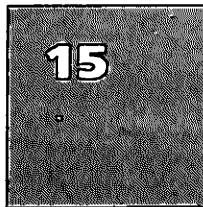
**Nota**

Para compilar este listado, deje las líneas a las que quitó las marcas de comentario en el listado 15.1 para ejecutar el listado 15.3, y también quite las marcas de comentario a las líneas 25, 39, 52, 64, 76 y 156 del listado 15.1.

**ENTRADA****LISTADO 15.4** Paso de parámetros por valor

```
1:  //Listado 15.4 - Este programa muestra el paso de parámetros
2:  //                por valor. Nuevamente se utiliza la clase Cadena
3:
4:  #include "lst15-01.hpp"
5:
6:
7:  class Empleado
8:  {
9:  public:
10:    Empleado();
11:    Empleado(char *, char *, char *, long);
12:    ~Empleado();
13:    Empleado(const Empleado &);
14:    Empleado & operator= (const Empleado &);
15:    const Cadena & ObtenerPrimerNombre() const
16:      { return suPrimerNombre; }
17:    const Cadena & ObtenerApellido() const
18:      { return suApellido; }
19:    const Cadena & ObtenerDireccion() const
20:      { return suDireccion; }
21:    long ObtenerSalario() const
22:      { return suSalario; }
23:    void AsignarPrimerNombre(const Cadena & primNombre)
24:      { suPrimerNombre = primNombre; }
25:    void AsignarApellido(const Cadena & Apellido)
26:      { suApellido = Apellido; }
27:    void AsignarDireccion(const Cadena & direccion)
28:      { suDireccion = direccion; }
29:    void AsignarSalario(long salario)
30:      { suSalario = salario; }
31: private:
32:   Cadena suPrimerNombre;
33:   Cadena suApellido;
```

```
34:     Cadena suDireccion;
35:     long suSalario;
36: };
37:
38: Empleado::Empleado():
39:     suPrimerNombre(""),
40:     suApellido(""),
41:     suDireccion(""),
42:     suSalario(0)
43: {}
44:
45: Empleado::Empleado(char * primerNombre, char * apellido,
46:                     char * direccion, long salario):
47:     suPrimerNombre(primerNombre),
48:     suApellido(apellido),
49:     suDireccion(direccion),
50:     suSalario(salario)
51: {}
52:
53: Empleado::Empleado(const Empleado & rhs):
54:     suPrimerNombre(rhs.ObtenerPrimerNombre()),
55:     suApellido(rhs.ObtenerApellido()),
56:     suDireccion(rhs.ObtenerDireccion()),
57:     suSalario(rhs.ObtenerSalario())
58: {}
59:
60: Empleado::~Empleado() {}
61:
62: Empleado & Empleado::operator= (const Empleado & rhs)
63: {
64:     if (this == &rhs)
65:         return *this;
66:     suPrimerNombre = rhs.ObtenerPrimerNombre();
67:     suApellido = rhs.ObtenerApellido();
68:     suDireccion = rhs.ObtenerDireccion();
69:     suSalario = rhs.ObtenerSalario();
70:     return *this;
71: }
72:
73: void FuncImpr(Empleado);
74:
75: void rFuncImpr(const Empleado&);
76:
77: int main()
78: {
79:     Empleado Edie("Jane", "Doe", "1461 Shore Parkway", 20000);
80:     Edie.AsignarSalario(20000);
81:     Edie.AsignarPrimerNombre("Edythe");
82:     Cadena cApellido("Levine");
83:     Edie.AsignarApellido(cApellido);
```

15

**LISTADO 15.4** CONTINUACIÓN

```
84:     cout << "Cuenta de constructores: " ;
85:     cout << Cadena::ConstructorCuenta << endl;
86:     rFuncImpr(Edie);
87:     cout << "Cuenta de constructores: " ;
88:     cout << Cadena::ConstructorCuenta << endl;
89:     FuncImpr(Edie);
90:     cout << "Cuenta de constructores: " ;
91:     cout << Cadena::ConstructorCuenta << endl;
92:     return 0;
93: }
94:
95: void FuncImpr(Empleado Edie)
96: {
97:     cout << "Nombre: ";
98:     cout << Edie.ObtenerPrimerNombre().ObtenerCadena();
99:     cout << " " << Edie.ObtenerApellido().ObtenerCadena();
100:    cout << ".\nDirección: ";
101:    cout << Edie.ObtenerDireccion().ObtenerCadena();
102:    cout << ".\nSalario: ";
103:    cout << Edie.ObtenerSalario();
104:    cout << endl;
105: }
106:
107: void rFuncImpr(const Empleado & Edie)
108: {
109:     cout << "Nombre: ";
110:    cout << Edie.ObtenerPrimerNombre().ObtenerCadena();
111:    cout << " " << Edie.ObtenerApellido().ObtenerCadena();
112:    cout << "\nDirección: ";
113:    cout << Edie.ObtenerDireccion().ObtenerCadena();
114:    cout << "\nSalario: ";
115:    cout << Edie.ObtenerSalario();
116:    cout << endl;
117: }
```

**SALIDA**

```
Constructor de Cadena(char *) constructor
Operador = de Cadena
Destructor de Cadena
Constructor de Cadena(char *) constructor
Operador = de Cadena
Cuenta de constructores: 5
Nombre: Edythe Levine
Dirección: 1461 Shore Parkway
Salario: 20000
Cuenta de constructores: 5
Constructor de Cadena(Cadena &)
Constructor de Cadena(Cadena &)
```

```
Constructor de Cadena(Cadena &)
Nombre: Edythe Levine.
Dirección: 1461 Shore Parkway.
Salario: 20000
    Destructor de Cadena
    Destructor de Cadena
    Destructor de Cadena
Cuenta de constructores: 8
    Destructor de Cadena
    Destructor de Cadena
    Destructor de Cadena
    Destructor de Cadena
    Destructor de Cadena
```

15

**ANÁLISIS** La salida muestra que se crearon cinco objetos de tipo Cadena, tres como parte de la creación de un objeto Empleado (línea 79), uno al asignar el nombre (línea 81) y otro más al crear la cadena cApellido (línea 82). Cuando el objeto Empleado se pasa a rFuncImpr() por referencia, no se crean objetos Empleado adicionales, por lo que no se crean objetos Cadena adicionales. (Éstos también se pasan por referencia.)

Cuando, en la línea 89, el objeto Empleado se pasa por valor a FuncImpr(), se crea una copia del objeto Empleado, y se crean tres objetos más de tipo Cadena (mediante llamadas al constructor de copia).

## Implementación con base en la herencia/contención en comparación con la delegación

Algunas veces una clase necesita algunos de los atributos de otra clase. Por ejemplo, suponga que necesita crear una clase llamada CatalogoPiezas. La especificación que dio define a CatalogoPiezas como una colección de piezas; cada pieza tiene un número de pieza único. CatalogoPiezas no permite entradas duplicadas y permite el acceso mediante el número de pieza.

El listado del repaso de la semana 2 proporciona una clase llamada ListaPiezas. Ya está comprobado y bien entendido el funcionamiento de ListaPiezas, por lo que puede apoyarse en esa tecnología al crear su CatalogoPiezas, en lugar de crear esta clase desde cero. La reutilización es una de las maneras más productivas de programar, es decir, puede basarse en lo que ya tiene.

Podría crear una nueva clase CatalogoPiezas y hacer que contenga a ListaPiezas. CatalogoPiezas podría delegar el manejo de la lista enlazada a la clase contenida ListaPiezas.

Una alternativa sería hacer que CatalogoPiezas se derivara de ListaPiezas y que, por consiguiente, heredara las propiedades de ListaPiezas. No obstante, si recuerda que la herencia pública proporciona una relación del tipo *es un*, debería preguntarse si CatalogoPiezas es realmente un tipo de ListaPiezas.

Una manera de responder a la pregunta de si `CatalogoPiezas` es una `ListaPiezas` sería asumir que `ListaPiezas` es la base y `CatalogoPiezas` es la clase derivada, y luego hacer estas otras preguntas:

1. ¿Hay algo en la clase base que no deba estar en la clase derivada? Por ejemplo, ¿tiene la clase base `ListaPiezas` funciones que sean inapropiadas para la clase `CatalogoPiezas`? De ser así, probablemente no sea conveniente la herencia pública.
2. ¿Podría la clase que usted está creando tener más de una clase base? Por ejemplo, ¿podría `CatalogoPiezas` necesitar dos clases `ListaPiezas` en cada objeto? De ser así, sería muy conveniente utilizar la contención.
3. ¿Necesita heredar de la clase base para poder aprovechar las funciones virtuales o los miembros de acceso protegido? De ser así, debe utilizar herencia, pública o privada.

Con base en las respuestas a estas preguntas, debe elegir ya sea entre herencia pública (la relación de tipo *es un*), herencia privada (lo que se explica más adelante en este día) o contención.

- **Contención:** La manera de declarar una clase como miembro de otra clase que es contenida por esa clase.
- **Delegación:** Uso de los atributos de una clase contenida para lograr funciones que no están disponibles de otra forma para la clase contenedora.
- **Implementación con base en:** Construir una clase con base en las capacidades de otra, sin utilizar herencia pública.

## Delegación

¿Por qué no derivar `CatalogoPiezas` de `ListaPiezas`? `CatalogoPiezas` no es una `ListaPiezas` porque los objetos `ListaPiezas` son colecciones ordenadas, y cada miembro de la colección se puede repetir. `CatalogoPiezas` tiene entradas únicas que no están ordenadas. El quinto miembro de `CatalogoPiezas` no es el número de pieza 5.

Evidentemente, hubiera sido posible heredar públicamente de `ListaPiezas` y luego redefinir `Insertar()` y los operadores de desplazamiento (`[]`) para hacer lo correcto, pero entonces hubiera cambiado la esencia de la clase `ListaPiezas`. En vez de esto, puede crear una clase `CatalogoPiezas` que no tenga operador de desplazamiento, que no permita duplicados y que defina a `operator+` para combinar dos conjuntos.

La primera forma de lograr esto es con la contención. `CatalogoPiezas` delegará el manejo de la lista a una `ListaPiezas` contenida. El listado 15.5 ejemplifica este método.

**LISTADO 15.5 Delegación a una ListaPiezas contenida**

```
1: // Listado 15.5 · Ejemplo de la delegación de responsabilidades
2: // a los miembros de una lista
3:
4: #include <iostream.h>
5:
6:
7: // ***** Pieza *****
8: // Clase base abstracta de piezas
9: class Pieza
10: {
11: public:
12:     Pieza() : suNumeroPieza(1) {}
13:     Pieza(int NumeroPieza):
14:         suNumeroPieza(NumeroPieza) {}
15:     virtual ~Pieza() {}
16:     int ObtenerNumeroPieza() const
17:         { return suNumeroPieza; }
18:     virtual void Desplegar() const = 0;
19: private:
20:     int suNumeroPieza;
21: };
22:
23: // implementación de una función virtual pura para que
24: // las clases derivadas se puedan encadenar
25: void Pieza::Desplegar() const
26: {
27:     cout << "\nNúmero de pieza: " << suNumeroPieza << endl;
28: }
29:
30: // ***** Pieza de Auto *****
31: class PiezaAuto : public Pieza
32: {
33: public:
34:     PiezaAuto() : suAnioModelo(94){}
35:     PiezaAuto(int anio, int numeroPieza);
36:     virtual void Desplegar() const
37:     {
38:         Pieza::Desplegar();
39:         cout << "Año del modelo: ";
40:         cout << suAnioModelo << endl;
41:     }
42: private:
43:     int suAnioModelo;
44: };
45:
46: PiezaAuto::PiezaAuto(int anio, int numeroPieza):
```

15

*continúa*

**LISTADO 15.5 CONTINUACIÓN**

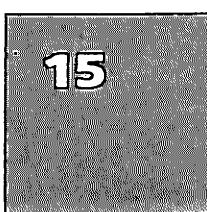
```
47:     suAnioModelo(anio),
48:     Pieza(numeroPieza)
49: {}
50:
51:
52: // ***** Pieza de AeroPlano *****
53: class PiezaAeroPlano : public Pieza
54: {
55: public:
56:     PiezaAeroPlano() : suNumeroMotor(1){};
57:     PiezaAeroPlano(int NumeroMotor, int NumeroPieza);
58:     virtual void Desplegar() const
59:     {
60:         Pieza::Desplegar();
61:         cout << "Motor número: ";
62:         cout << suNumeroMotor << endl;
63:     }
64: private:
65:     int suNumeroMotor;
66: };
67:
68: PiezaAeroPlano::PiezaAeroPlano(int NumeroMotor, int NumeroPieza):
69:     suNumeroMotor(NumeroMotor),
70:     Pieza(NumeroPieza)
71: {}
72:
73: // ***** Nodo Pieza *****
74: class NodoPieza
75: {
76: public:
77:     NodoPieza(Pieza *);
78:     ~NodoPieza();
79:     void AsignarSiguiente(NodoPieza * nodo)
80:     { suSiguiente = nodo; }
81:     NodoPieza * ObtenerSiguiente() const;
82:     Pieza * ObtenerPieza() const;
83: private:
84:     Pieza * suPieza;
85:     NodoPieza * suSiguiente;
86: };
87:
88: // Implementaciones de NodoPieza...
89: NodoPieza::NodoPieza(Pieza * apPieza):
90:     suPieza(apPieza),
91:     suSiguiente(0)
92: {}
93:
94: NodoPieza::~NodoPieza()
```

```
95:      {
96:          delete suPieza;
97:          suPieza = NULL;
98:          delete suSiguiente;
99:          suSiguiente = NULL;
100:     }
101:
102:     // Regresa NULL si no hay NodoPieza siguiente
103:     NodoPieza * NodoPieza::ObtenerSiguiente() const
104:     {
105:         return suSiguiente;
106:     }
107:
108:     Pieza * NodoPieza::ObtenerPieza() const
109:     {
110:         if (suPieza)
111:             return suPieza;
112:         else
113:             return NULL; //error
114:     }
115:
116:     // ***** Lista de Piezas *****
117: class ListaPiezas
118: {
119: public:
120:     ListaPiezas();
121:     ~ListaPiezas();
122:     // inecesita constructor de copia y operador igual a!
123:     void Iterar(void (Pieza::*f)() const) const;
124:     Pieza * Encontrar(int & posicion, int NumeroPieza) const;
125:     Pieza * ObtenerPrimero() const;
126:     void Insertar(Pieza *);
127:     Pieza * operator[](int) const;
128:     int ObtenerCuenta() const
129:         { return suCuenta; }
130:     static ListaPiezas& ObtenerListaPiezasGlobal()
131:         { return ListaPiezasGlobal; }
132: private:
133:     NodoPieza * apCabeza;
134:     int suCuenta;
135:     static ListaPiezas ListaPiezasGlobal;
136: };
137:
138: ListaPiezas ListaPiezas::ListaPiezasGlobal;
139:
140: ListaPiezas::ListaPiezas():
141:     apCabeza(0),
142:     suCuenta(0)
143: {}
```

**LISTADO 15.5 CONTINUACIÓN**

```
144:  
145:     ListaPiezas::~ListaPiezas()  
146:     {  
147:         delete apCabeza;  
148:     }  
149:  
150:     Pieza* ListaPiezas::ObtenerPrimero() const  
151:     {  
152:         if (apCabeza)  
153:             return apCabeza->ObtenerPieza();  
154:         else  
155:             return NULL; // atrapar error aquí  
156:     }  
157:  
158:     Pieza * ListaPiezas::operator[](int desplazamiento) const  
159:     {  
160:         NodoPieza * apNodo = apCabeza;  
161:  
162:         if (!apCabeza)  
163:             return NULL; // atrapar error aquí  
164:         if (desplazamiento > suCuenta)  
165:             return NULL; // error  
166:         for (int i = 0; i < desplazamiento; i++)  
167:             apNodo = apNodo->ObtenerSiguiente();  
168:         return apNodo->ObtenerPieza();  
169:     }  
170:  
171:     Pieza* ListaPiezas::Encontrar(int & posicion, int NumeroPieza) const  
172:     {  
173:         NodoPieza * apNodo = NULL;  
174:  
175:         for (apNodo = apCabeza, posicion = 0;  
176:              apNodo!=NULL;  
177:              apNodo = apNodo->ObtenerSiguiente(), posicion++)  
178:         {  
179:             if (apNodo->ObtenerPieza()->ObtenerNumeroPieza() == NumeroPieza)  
180:                 break;  
181:         }  
182:         if (apNodo == NULL)  
183:             return NULL;  
184:         else  
185:             return apNodo->ObtenerPieza();  
186:     }  
187:
```

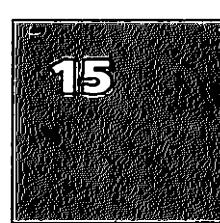
```
188: void ListaPiezas::Iterar(void (Pieza::*func)() const) const
189: {
190:     if (!apCabeza)
191:         return;
192:     NodoPieza * apNodo = apCabeza;
193:     do
194:         (apNodo->ObtenerPieza()->*func)();
195:     while (apNodo = apNodo->ObtenerSiguiente());
196: }
197:
198: void ListaPiezas::Insertar(Pieza * apPieza)
199: {
200:     NodoPieza * apNodo = new NodoPieza(apPieza);
201:     NodoPieza * apActual = apCabeza;
202:     NodoPieza * apSiguiente = NULL;
203:     int Nuevo = apPieza->ObtenerNumeroPieza();
204:     int Siguiente = 0;
205:
206:     suCuenta++;
207:     if (!apCabeza)
208:     {
209:         apCabeza = apNodo;
210:         return;
211:     }
212:     // si éste es más pequeño que el nodo cabeza
213:     // entonces se convierte en el nuevo nodo cabeza
214:     if (apCabeza->ObtenerPieza()->ObtenerNumeroPieza() > Nuevo)
215:     {
216:         apNodo->AsignarSiguiente(apCabeza);
217:         apCabeza = apNodo;
218:         return;
219:     }
220:     for (;;)
221:     {
222:         // si no hay siguiente, agregar éste
223:         if (!apActual->ObtenerSiguiente())
224:         {
225:             apActual->AsignarSiguiente(apNodo);
226:             return;
227:         }
228:         // si va después de éste y antes del siguiente
229:         // entonces insertarlo aquí, de no ser así
230:         // obtener el siguiente
231:         apSiguiente = apActual->ObtenerSiguiente();
```

*continúa*

**LISTADO 15.5 CONTINUACIÓN**

```
232:         Siguiente = apSiguiente->ObtenerPieza()->ObtenerNumeroPieza();
233:         if (Siguiente > Nuevo)
234:         {
235:             apActual->AsignarSiguiente(apNodo);
236:             apNodo->AsignarSiguiente(apSiguiente);
237:             return;
238:         }
239:         apActual = apSiguiente;
240:     }
241: }
242:
243: class CatalogoPiezas
244: {
245: public:
246:     void Insertar(Pieza *);
247:     int Existe(int NumeroPieza);
248:     Pieza * Obtener(int NumeroPieza);
249:     operator+(const CatalogoPiezas &);
250:     void MostrarTodo()
251:     { laListaPiezas.Iterar(&Pieza::Desplegar); }
252: private:
253:     ListaPiezas laListaPiezas;
254: };
255:
256: void CatalogoPiezas::Insertar(Pieza * nuevaPieza)
257: {
258:     int numeroPieza = nuevaPieza->ObtenerNumeroPieza();
259:     int desplazamiento;
260:
261:     if (!laListaPiezas.Encontrar(desplazamiento, numeroPieza))
262:         laListaPiezas.Insertar(nuevaPieza);
263:     else
264:     {
265:         cout << numeroPieza << " fue la ";
266:         switch (desplazamiento)
267:         {
268:             case 0: cout << "primera ";
269:                     break;
270:             case 1: cout << "segunda ";
271:                     break;
272:             case 2: cout << "tercera ";
273:                     break;
274:             default: cout << desplazamiento+1 << "a ";
275:         }
276:     }
277: }
```

```
276:         cout << "entrada. ¡Rechazada!\n";
277:     }
278: }
279:
280: int CatalogoPiezas::Existe(int NumeroPieza)
281: {
282:     int desplazamiento;
283:
284:     laListaPiezas.Encontrar(desplazamiento,NumeroPieza);
285:     return desplazamiento;
286: }
287:
288: Pieza * CatalogoPiezas::Obtener(int NumeroPieza)
289: {
290:     int desplazamiento;
291:
292:     Pieza * laPieza = laListaPiezas.Encontrar
293:     (desplazamiento, NumeroPieza);
294:     return laPieza;
295:
296:
297: int main()
298: {
299:     CatalogoPiezas cp;
300:     Pieza * apPieza = NULL;
301:     int NumeroPieza;
302:     int valor;
303:     int opcion;
304:
305:     while (1)
306:     {
307:         cout << "(0)Salir (1)Auto (2)Avión: ";
308:         cin >> opcion;
309:         if (!opcion)
310:             break;
311:         cout << "¿Nuevo NumeroPieza?: ";
312:         cin >> NumeroPieza;
313:         if (opcion == 1)
314:         {
315:             cout << "¿Año del modelo?: ";
316:             cin >> valor;
317:             apPieza = new PiezaAuto(valor, NumeroPieza);
318:         }
```



**LISTADO 15.5** CONTINUACIÓN

```
319:         else
320:         {
321:             cout << "¿Número de motor?: ";
322:             cin >> valor;
323:             apPieza = new PiezaAeroPlano(valor, NumeroPieza);
324:         }
325:         cp.Insertar(apPieza);
326:     }
327:     cp.MostrarTodo();
328:     return 0;
329: }
```

---

**SALIDA**

```
(0)Salir (1)Auto (2)Avión: 1
¿Nuevo NumeroPieza?: 1234
¿Año del modelo?: 94
(0)Salir (1)Auto (2)Avión: 1
¿Nuevo NumeroPieza?: 4434
¿Año del modelo?: 93
(0)Salir (1)Auto (2)Avión: 1
¿Nuevo NumeroPieza?: 1234
¿Año del modelo?: 94
1234 fue la primera entrada. ¡Rechazada!
(0)Salir (1)Auto (2)Avión: 1
¿Nuevo NumeroPieza?: 2345
¿Año del modelo?: 93
(0)Salir (1)Auto (2)Avión: 0

Número de pieza: 1234
Año del modelo: 94

Número de pieza: 2345
Año del modelo: 93

Número de pieza: 4434
Año del modelo: 93
```

**ANÁLISIS**

El listado 15.5 reproduce las clases Pieza, NodoPieza y ListaPiezas del repaso de la semana 2.

En las líneas 243 a 254 se declara una nueva clase llamada CatalogoPiezas. CatalogoPiezas tiene una clase ListaPiezas como dato miembro, a la que delega el manejo

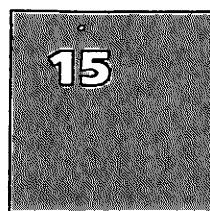
de la lista. Otra forma de decir esto es que `CatalogoPiezas` se implementa con base en `ListaPiezas`.

Observe que los clientes de `CatalogoPiezas` no tienen acceso directo a `ListaPiezas`. La interfaz es por medio de `CatalogoPiezas`, y como tal, el comportamiento de `ListaPiezas` cambia dramáticamente. Por ejemplo, el método `CatalogoPiezas::Insertar()` no permite entradas duplicadas en `ListaPiezas`.

La implementación de `CatalogoPiezas::Insertar()` empieza en la línea 256. A la `Pieza` que se pasa como parámetro se le pide el valor de la variable miembro `suNumeroPieza`. Este valor se proporciona para el método `Encontrar()` de `ListaPiezas`, y si no se encuentra ese valor, se inserta el número de pieza; si se encuentra el valor, se imprime un mensaje informativo de error.

Observe que `CatalogoPiezas` se encarga de la inserción al llamar a `Insertar()` sobre su variable miembro `laListaPiezas`, que es un objeto de la clase `ListaPiezas`. La mecánica de la inserción en sí y el mantenimiento de la lista enlazada, junto con la búsqueda y la recuperación de la lista enlazada, se mantienen en el miembro `ListaPiezas` contenido en `CatalogoPiezas`. No hay razón para que `CatalogoPiezas` reproduzca este código; puede aprovechar completamente la interfaz bien definida.

Ésta es la esencia de la reutilización dentro de C++: `CatalogoPiezas` puede reutilizar el código de `ListaPiezas`, y el diseñador de `CatalogoPiezas` puede ignorar los detalles de implementación de `ListaPiezas`. La interfaz para `ListaPiezas` (es decir, la declaración de la clase) proporciona toda la información que necesita el diseñador de la clase `CatalogoPiezas`.



## Herencia privada

Si `CatalogoPiezas` necesitara tener acceso a los miembros protegidos de `ListaPiezas` (en este caso no existe ninguno), o necesitara redefinir cualquiera de los métodos de `ListaPiezas`, `CatalogoPiezas` estaría obligada a heredar de `ListaPiezas`.

Pero como los objetos de la clase `CatalogoPiezas` no son objetos de la clase `ListaPiezas`, y como usted no quiere exponer todo el conjunto de funcionalidad de `ListaPiezas` a los clientes de `CatalogoPiezas`, necesita usar herencia privada.

Lo primero que hay que saber acerca de la herencia privada es que todas las variables y funciones miembro de la clase base se tratan como si fueran declaradas como privadas, sin importar su nivel de acceso actual en la clase base. Por lo tanto, para cualquier función que no sea miembro de `CatalogoPiezas`, son inaccesibles todas las funciones heredadas de `ListaPiezas`. Esto es crucial: la herencia privada no involucra la interfaz heredada, sólo la implementación.

Para los clientes de la clase CatalogoPiezas, la clase ListaPiezas es invisible. Ningún componente de la interfaz está disponible para dichos clientes: no pueden llamar a ninguno de sus métodos. Sin embargo, pueden llamar a los métodos de CatalogoPiezas; entonces los métodos de CatalogoPiezas pueden tener acceso a todo lo que haya en ListaPiezas ya que CatalogoPiezas se deriva de ListaPiezas. Lo importante aquí es que los objetos de CatalogoPiezas no son objetos de ListaPiezas, como hubiera sido con la herencia pública. CatalogoPiezas se implementa con base en ListaPiezas, como hubiera sido con la contención. La herencia privada es sólo una conveniencia.

El listado 15.6 muestra el uso de la herencia privada, para lo cual modifica la clase CatalogoPiezas como derivada en forma privada de ListaPiezas.

**ENTRADA LISTADO 15.6 Herencia privada**

---

```
1: // Listado 15.6 - Muestra de la herencia privada
2:
3: #include <iostream.h>
4:
5:
6: // ***** Pieza *****
7: // Clase base abstracta de piezas
8: class Pieza
9: {
10: public:
11:     Pieza() : suNumeroPieza(1) {}
12:     Pieza(int NumeroPieza):
13:         suNumeroPieza(NumeroPieza) {}
14:     virtual ~Pieza() {}
15:     int ObtenerNumeroPieza() const
16:         { return suNumeroPieza; }
17:     virtual void Desplegar() const =0;
18: private:
19:     int suNumeroPieza;
20: };
21:
22: // implementación de la función virtual pura para que
23: // las clases derivadas se puedan encadenar
24: void Pieza::Desplegar() const
25: {
26:     cout << "\nNúmero de pieza: " << suNumeroPieza << endl;
27: }
28:
29: // ***** Pieza de Auto *****
30: class PiezaAuto : public Pieza
31: {
32: public:
33:     PiezaAuto() : suAnioModelo(94) {}
34:     PiezaAuto(int anio, int numeroPieza);
35:     virtual void Desplegar() const
```

```
36:      {
37:          Pieza::Desplegar();
38:          cout << "Año del modelo: ";
39:          cout << suAnioModelo << endl;
40:      }
41:  private:
42:      int suAnioModelo;
43:  };
44:
45: PiezaAuto::PiezaAuto(int anio, int numeroPieza):
46:     suAnioModelo(anio),
47:     Pieza(numeroPieza)
48: {}
49:
50: // ***** Pieza de AeroPlano *****
51: class PiezaAeroPlano : public Pieza
52: {
53: public:
54:     PiezaAeroPlano() : suNumeroMotor(1) {};
55:     PiezaAeroPlano(int NumeroMotor, int NumeroPieza);
56:     virtual void Desplegar() const
57:     {
58:         Pieza::Desplegar();
59:         cout << "Motor número: ";
60:         cout << suNumeroMotor << endl;
61:     }
62:  private:
63:      int suNumeroMotor;
64:  };
65:
66: PiezaAeroPlano::PiezaAeroPlano(int NumeroMotor, int NumeroPieza):
67:     suNumeroMotor(NumeroMotor),
68:     Pieza(NumeroPieza)
69: {}
70:
71: // ***** Nodo Pieza *****
72: class NodoPieza
73: {
74: public:
75:     NodoPieza(Pieza *);
76:     ~NodoPieza();
77:     void AsignarSiguiente(NodoPieza * nodo)
78:     {
79:         suSiguiente = nodo;
80:     }
81:     NodoPieza * ObtenerSiguiente() const;
82:     Pieza * ObtenerPieza() const;
83:  private:
84:     Pieza * suPieza;
85:     NodoPieza * suSiguiente;
```

*continúa*

**LISTADO 15.6** CONTINUACIÓN

```
84:     };
85:
86:     // Implementaciones de NodoPieza...
87:     NodoPieza::NodoPieza(Pieza * apPieza):
88:         suPieza(apPieza),
89:         suSiguiente(0)
90:     {}
91:
92:     NodoPieza::~NodoPieza()
93:     {
94:         delete suPieza;
95:         suPieza = NULL;
96:         delete suSiguiente;
97:         suSiguiente = NULL;
98:     }
99:
100:    // Regresa NULL si no hay NodoPieza siguiente
101:    NodoPieza * NodoPieza::ObtenerSiguiente() const
102:    {
103:        return suSiguiente;
104:    }
105:
106:    Pieza * NodoPieza::ObtenerPieza() const
107:    {
108:        if (suPieza)
109:            return suPieza;
110:        else
111:            return NULL; //error
112:    }
113:
114:    // ***** Lista de Piezas *****
115: class ListaPiezas
116: {
117: public:
118:     ListaPiezas();
119:     ~ListaPiezas();
120:     // inecesita constructor de copia y operador igual a!
121:     void Iterar(void (Pieza::*f)() const) const;
122:     Pieza * Encontrar(int & posicion, int NumeroPieza) const;
123:     Pieza * ObtenerPrimero() const;
124:     void Insertar(Pieza *);
125:     Pieza * operator[](int) const;
126:     int ObtenerCuenta() const
127:         { return suCuenta; }
128:     static ListaPiezas& ObtenerListaPiezasGlobal()
129:         { return ListaPiezasGlobal; }
130: private:
131:     NodoPieza * apCabeza;
```

```
132:     int suCuenta;
133:     static ListaPiezas ListaPiezasGlobal;
134: };
135:
136: ListaPiezas ListaPiezas::ListaPiezasGlobal;
137:
138: ListaPiezas::ListaPiezas():
139:     apCabeza(0),
140:     suCuenta(0)
141: {}
142:
143: ListaPiezas::~ListaPiezas()
144: {
145:     delete apCabeza;
146: }
147:
148: Pieza* ListaPiezas::ObtenerPrimero() const
149: {
150:     if (apCabeza)
151:         return apCabeza->ObtenerPieza();
152:     else
153:         return NULL; // atrapar error aqui
154: }
155:
156: Pieza * ListaPiezas::operator[](int desplazamiento) const
157: {
158:     NodoPieza * apNodo = apCabeza;
159:
160:     if (!apCabeza)
161:         return NULL; // atrapar error aqui
162:     if (desplazamiento > suCuenta)
163:         return NULL; // error
164:     for (int i = 0; i < desplazamiento; i++)
165:         apNodo = apNodo->ObtenerSiguiente();
166:     return apNodo->ObtenerPieza();
167: }
168:
169: Pieza* ListaPiezas::Encontrar(int & posicion, int NumeroPieza) const
170: {
171:     NodoPieza * apNodo = NULL;
172:
173:     for (apNodo = apCabeza, posicion = 0;
174:          apNodo!=NULL;
175:          apNodo = apNodo->ObtenerSiguiente(), posicion++)
176:     {
177:         if (apNodo->ObtenerPieza()->ObtenerNumeroPieza() == NumeroPieza)
178:             break;
179:     }
```

*continúa*

**LISTADO 15.6** CONTINUACIÓN

```
180:     if (apNodo == NULL)
181:         return NULL;
182:     else
183:         return apNodo->ObtenerPieza();
184: }
185:
186: void ListaPiezas::Iterar(void (Pieza::*func)() const) const
187: {
188:     if (!apCabeza)
189:         return;
190:     NodoPieza * apNodo = apCabeza;
191:     do
192:         (apNodo->ObtenerPieza()->*func)();
193:     while (apNodo = apNodo->ObtenerSiguiente());
194: }
195:
196: void ListaPiezas::Insertar(Pieza * apPieza)
197: {
198:     NodoPieza * apNodo = new NodoPieza(apPieza);
199:     NodoPieza * apActual = apCabeza;
200:     NodoPieza * apSiguiente = NULL;
201:     int Nuevo = apPieza->ObtenerNumeroPieza();
202:     int Siguiente = 0;
203:
204:     suCuenta++;
205:     if (!apCabeza)
206:     {
207:         apCabeza = apNodo;
208:         return;
209:     }
210:     // si éste es más pequeño que el nodo cabeza
211:     // se convierte en el nuevo nodo cabeza
212:     if (apCabeza->ObtenerPieza()->ObtenerNumeroPieza() > Nuevo)
213:     {
214:         apNodo->AsignarSiguiente(apCabeza);
215:         apCabeza = apNodo;
216:         return;
217:     }
218:     for (;;)
219:     {
220:         // si no hay siguiente, agregar éste
221:         if (!apActual->ObtenerSiguiente())
222:         {
223:             apActual->AsignarSiguiente(apNodo);
224:             return;
225:         }
226:         // si va después de éste y antes del siguiente
227:         // entonces insertarlo aquí, de no ser así, obtener el siguiente
```

```
228:         apSiguiente = apActual->ObtenerSiguiente();
229:         Siguiente = apSiguiente->ObtenerPieza()->ObtenerNumeroPieza();
230:         if (Siguiente > Nuevo)
231:         {
232:             apActual->AsignarSiguiente(apNodo);
233:             apNodo->AsignarSiguiente(apSiguiente);
234:             return;
235:         }
236:         apActual = apSiguiente;
237:     }
238: }
239:
240: class CatalogoPiezas : private ListaPiezas
241: {
242: public:
243:     void Insertar(Pieza *);
244:     int Existe(int NumeroPieza);
245:     Pieza * Obtener(int NumeroPieza);
246:     operator+(const CatalogoPiezas &);
247:     void MostrarTodo()
248:     { Iterar(&Pieza::Desplegar); }
249: private:
250: };
251:
252: void CatalogoPiezas::Insertar(Pieza * nuevaPieza)
253: {
254:     int numeroPieza = nuevaPieza->ObtenerNumeroPieza();
255:     int desplazamiento;
256:
257:     if (!Encontrar(desplazamiento, numeroPieza))
258:         ListaPiezas::Insertar(nuevaPieza);
259:     else
260:     {
261:         cout << numeroPieza << " fue la ";
262:         switch (desplazamiento)
263:         {
264:             case 0: cout << "primera ";
265:                     break;
266:             case 1: cout << "segunda ";
267:                     break;
268:             case 2: cout << "tercera ";
269:                     break;
270:             default: cout << desplazamiento+1 << "a ";
271:         }
272:         cout << "entrada. ¡Rechazada!\n";
273:     }
274: }
275:
276: int CatalogoPiezas::Existe(int NumeroPieza)
```

*continúa*

**LISTADO 15.6** CONTINUACIÓN

---

```
277:  {
278:      int desplazamiento;
279:
280:      Encontrar(desplazamiento,NumeroPieza);
281:      return desplazamiento;
282:  }
283:
284:  Pieza * CatalogoPiezas::Obtener(int NumeroPieza)
285:  {
286:      int desplazamiento;
287:
288:      return (Encontrar(desplazamiento, NumeroPieza));
289:  }
290:
291:  int main()
292:  {
293:      CatalogoPiezas cp;
294:      Pieza * apPieza = NULL;
295:      int NumeroPieza;
296:      int valor;
297:      int opcion;
298:
299:      while (1)
300:      {
301:          cout << "(0)Salir (1)Auto (2)Avión: ";
302:          cin >> opcion;
303:          if (!opcion)
304:              break;
305:          cout << "¿Nuevo NumeroPieza?: ";
306:          cin >> NumeroPieza;
307:          if (opcion == 1)
308:          {
309:              cout << "¿Año del modelo?: ";
310:              cin >> valor;
311:              apPieza = new PiezaAuto(valor, NumeroPieza);
312:          }
313:          else
314:          {
315:              cout << "¿Número de motor?: ";
316:              cin >> valor;
317:              apPieza = new PiezaAeroPlano(valor, NumeroPieza);
318:          }
319:          cp.Insertar(apPieza);
320:      }
321:      cp.MostrarTodo();
322:      return 0;
323:  }
```

---

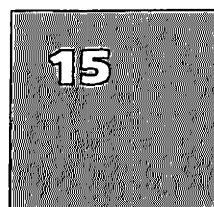
**SALIDA**

```
(0)Salir (1)Auto (2)Avión: 1
¿Nuevo NumeroPieza?: 1234
¿Año del modelo?: 94
(0)Salir (1)Auto (2)Avión: 1
¿Nuevo NumeroPieza?: 4434
¿Año del modelo?: 93
(0)Salir (1)Auto (2)Avión: 1
¿Nuevo NumeroPieza?: 1234
¿Año del modelo?: 94
1234 fue la primera entrada. ¡Rechazada!
(0)Salir (1)Auto (2)Avión: 1
¿Nuevo NumeroPieza?: 2345
¿Año del modelo?: 93
(0)Salir (1)Auto (2)Avión: 0
```

Número de pieza: 1234  
Año del modelo: 94

Número de pieza: 2345  
Año del modelo: 93

Número de pieza: 4434  
Año del modelo: 93

**ANÁLISIS**

El listado 15.6 muestra una interfaz cambiada para CatalogoPiezas y el programa controlador modificado. Las interfaces para las otras clases permanecen sin cambio, quedando igual que en el listado 15.5.

En la línea 240 del listado 15.6, CatalogoPiezas se declara para que se derive en forma privada de ListaPiezas. La interfaz para CatalogoPiezas no cambia y queda igual que en el listado 15.5, aunque, por supuesto, ya no necesita un objeto de tipo ListaPiezas como dato miembro.

La función MostrarTodo() de CatalogoPiezas llama a Iterar() de ListaPiezas con el apuntador apropiado a la función miembro de la clase Pieza. MostrarTodo() actúa como interfaz pública para Iterar(), proporcionando la información correcta, pero evitando que las clases cliente llamen a Iterar() en forma directa. Aunque ListaPiezas podría permitir que se pasaran otras funciones a Iterar(), CatalogoPiezas no lo permitiría.

La función `Insertar()` también ha cambiado. Observe que en la línea 257 ahora se llama directamente a `Encontrar()` debido a que se hereda de la clase base. La llamada a `Insertar()` de la línea 258 debe estar identificada completamente, o de lo contrario terminaría realizando una recursión sobre el método `Insertar()` de `CatalogoPiezas`.

En resumen, cuando los métodos de `CatalogoPiezas` necesitan llamar a los métodos de `ListaPiezas`, pueden hacerlo en forma directa. La única excepción es cuando `CatalogoPiezas` haya redefinido el método y se necesite la versión de `ListaPiezas`, en cuyo caso se debe identificar completamente el nombre de la función.

La herencia privada permite que `CatalogoPiezas` herede lo que puede utilizar, pero aún proporciona un acceso controlado al método `Insertar()` (de `ListaPiezas`) y a otros métodos a los que las clases cliente no deben tener acceso directo.

| DEBE                                                                                                                                                                                                                                                                                                                                                                                                                                   | NO DEBE                                                                                                                                                                                                                                                                                                                                                                                                                       |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>DEBE</b> utilizar herencia pública cuando los objetos de la clase derivada sean del tipo de la clase base.</p> <p><b>DEBE</b> utilizar la contención cuando quiera delegar funcionalidad a otra clase, pero no necesite acceso a sus miembros protegidos.</p> <p><b>DEBE</b> utilizar herencia privada cuando necesite implementar una clase con base en otra, y necesite acceso a los miembros protegidos de la clase base.</p> | <p><b>NO DEBE</b> utilizar herencia privada cuando necesite utilizar más de una instancia de la clase base. Debe usar la contención. Por ejemplo, si <code>CatalogoPiezas</code> necesitara dos instancias de <code>ListaPiezas</code>, no tendría que usar herencia privada.</p> <p><b>NO DEBE</b> usar herencia pública cuando los clientes de la clase derivada no deban tener acceso a los miembros de la clase base.</p> |

## Clases amigas

Algunas veces se crean varias clases, como un conjunto. Por ejemplo, `NodoPieza` y `ListaPiezas` estaban estrechamente acopladas, y hubiera sido conveniente que `ListaPiezas` leyera de manera directa el apuntador a una Pieza de `NodoPieza` llamado `suPieza`.

No sería conveniente hacer que `suPieza` fuera público, ni siquiera protegido, ya que éste es un detalle de implementación de `NodoPieza` y es mejor mantenerlo privado. Sin embargo, es conveniente exponerlo a `ListaPiezas`.

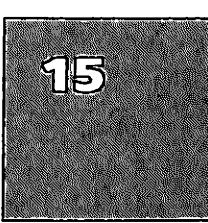
Si quiere exponer sus funciones o datos miembro privados a otra clase, debe declarar esa clase como amiga. Esto extiende la interfaz de su clase para incluir a la clase amiga.

Después de que `NodoPieza` declara a `ListaPiezas` como amiga, todos los datos y funciones miembro de `NodoPieza` son públicos, en lo que a `ListaPiezas` respecta.

Es importante observar que la amistad no se puede transferir. El hecho de que usted sea mi amigo y que Juan sea su amigo, no significa que Juan sea mi amigo. La amistad tampoco se hereda. De nuevo, el hecho de que usted sea mi amigo y yo comparta mis secretos con usted, no significa que esté dispuesto a compartir mis secretos con sus hijos.

Por último, la amistad no es commutativa. Asignar la Clase Uno como amiga de la Clase Dos, no hace que la Clase Dos sea amiga de la Clase Uno. Tal vez usted quiera decirme sus secretos, pero eso no significa que yo quiera decirle los míos.

El listado 15.7 muestra un ejemplo de la amistad modificando el ejemplo del listado 15.6, convirtiendo a `ListaPiezas` en amiga de `NodoPieza`. Observe que esto no hace que `NodoPieza` sea amiga de `ListaPiezas`.



#### ENTRADA LISTADO 15.7 Ejemplo de clases amigas

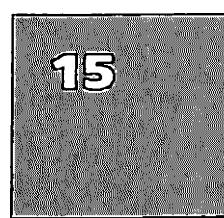
```
1: // Listado 15.7 - Ejemplo de clases amigas
2:
3: #include <iostream.h>
4:
5:
6: // ***** Pieza *****
7: // Clase base abstracta de piezas
8: class Pieza
9: {
10: public:
11:     Pieza() : suNumeroPieza(1) {}
12:     Pieza(int NumeroPieza):
13:         suNumeroPieza(NumeroPieza) {}
14:     virtual ~Pieza() {}
15:     int ObtenerNumeroPieza() const
16:         { return suNumeroPieza; }
17:     virtual void Desplegar() const = 0;
18: private:
19:     int suNumeroPieza;
20: };
21:
22: // implementación de la función virtual pura para que
23: // las clases derivadas se puedan encadenar
24: void Pieza::Desplegar() const
25: {
26:     cout << "\nNúmero de pieza: ";
27:     cout << suNumeroPieza << endl;
28: }
```

continúa

**LISTADO 15.7** CONTINUACIÓN

```
29:  
30:    // ***** Pieza de Auto *****  
31:    class PiezaAuto : public Pieza  
32:    {  
33:        public:  
34:            PiezaAuto() : suAnioModelo(94) {}  
35:            PiezaAuto(int anio, int numeroPieza);  
36:            virtual void Desplegar() const  
37:            {  
38:                Pieza::Desplegar();  
39:                cout << "Año del modelo: ";  
40:                cout << suAnioModelo << endl;  
41:            }  
42:        private:  
43:            int suAnioModelo;  
44:        };  
45:  
46:        PiezaAuto::PiezaAuto(int anio, int numeroPieza):  
47:            suAnioModelo(anio),  
48:            Pieza(numeroPieza)  
49:        {}  
50:  
51:    // ***** Pieza de AeroPlano *****  
52:    class PiezaAeroPlano : public Pieza  
53:    {  
54:        public:  
55:            PiezaAeroPlano() : suNumeroMotor(1) {};  
56:            PiezaAeroPlano(int NumeroMotor, int NumeroPieza);  
57:            virtual void Desplegar() const  
58:            {  
59:                Pieza::Desplegar();  
60:                cout << "Motor número: ";  
61:                cout << suNumeroMotor << endl;  
62:            }  
63:        private:  
64:            int suNumeroMotor;  
65:        };  
66:  
67:        PiezaAeroPlano::PiezaAeroPlano(int NumeroMotor, int NumeroPieza):  
68:            suNumeroMotor(NumeroMotor),  
69:            Pieza(NumeroPieza)  
70:        {}  
71:  
72:    // ***** Nodo de Pieza *****  
73:    class NodoPieza  
74:    {
```

```
75:     public:
76:         friend class ListaPiezas;
77:         NodoPieza (Pieza *);
78:         ~NodoPieza();
79:         void AsignarSiguiente(NodoPieza * nodo)
80:             { suSiguiente = nodo; }
81:         NodoPieza * ObtenerSiguiente() const;
82:         Pieza * ObtenerPieza() const;
83:     private:
84:         Pieza * suPieza;
85:         NodoPieza * suSiguiente;
86:     };
87:
88:     NodoPieza::NodoPieza(Pieza * apPieza):
89:         suPieza(apPieza),
90:         suSiguiente(0)
91:     {}
92:
93:     NodoPieza::~NodoPieza()
94:     {
95:         delete suPieza;
96:         suPieza = NULL;
97:         delete suSiguiente;
98:         suSiguiente = NULL;
99:     }
100:
101:    // Regresa NULL si no hay NodoPieza siguiente
102:    NodoPieza * NodoPieza::ObtenerSiguiente() const
103:    {
104:        return suSiguiente;
105:    }
106:
107:    Pieza * NodoPieza::ObtenerPieza() const
108:    {
109:        if (suPieza)
110:            return suPieza;
111:        else
112:            return NULL; //error
113:    }
114:
115:    // ***** Lista de Piezas *****
116:    class ListaPiezas
117:    {
118:        public:
119:            ListaPiezas();
120:            ~ListaPiezas();
```

*continúa*

**LISTADO 15.7** CONTINUACIÓN

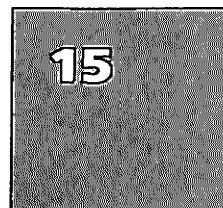
```
121:      // inecesita constructor de copia y operador igual a!
122:      void Iterar(void (Pieza::*f)() const) const;
123:      Pieza * Encontrar(int & posicion, int NumeroPieza) const;
124:      Pieza * ObtenerPrimero() const;
125:      void Insertar(Pieza *);
126:      Pieza * operator[](int) const;
127:      int ObtenerCuenta() const
128:          { return suCuenta; }
129:      static ListaPiezas & ObtenerListaPiezasGlobal()
130:          { return ListaPiezasGlobal; }
131: private:
132:     NodoPieza * apCabeza;
133:     int suCuenta;
134:     static ListaPiezas ListaPiezasGlobal;
135: };
136:
137:     ListaPiezas ListaPiezas::ListaPiezasGlobal;
138:     // Implementaciones para listas...
139:     ListaPiezas::ListaPiezas():
140:         apCabeza(0),
141:         suCuenta(0)
142:     {}
143:
144:     ListaPiezas::~ListaPiezas()
145:     {
146:         delete apCabeza;
147:     }
148:
149:     Pieza* ListaPiezas::ObtenerPrimero() const
150:     {
151:         if (apCabeza)
152:             return apCabeza->suPieza;
153:         else
154:             return NULL; // atrapar error aquí
155:     }
156:
157:     Pieza * ListaPiezas::operator[](int desplazamiento) const
158:     {
159:         NodoPieza * apNodo = apCabeza;
160:
161:         if (!apCabeza)
162:             return NULL; // atrapar error aquí
163:         if (desplazamiento > suCuenta)
164:             return NULL; // error
165:         for (int i = 0; i < desplazamiento; i++)
166:             apNodo = apNodo->suSiguiente;
```

```
167:         return apNodo->suPieza;
168:     }
169:
170:     Pieza * ListaPiezas::Encontrar(int & posicion, int NumeroPieza) const
171:     {
172:         NodoPieza * apNodo = NULL;
173:
174:         for (apNodo = apCabeza, posicion = 0;
175:              apNodo != NULL;
176:              apNodo = apNodo->suSiguiente, posicion++)
177:         {
178:             if (apNodo->suPieza->ObtenerNumeroPieza() == NumeroPieza)
179:                 break;
180:         }
181:         if (apNodo == NULL)
182:             return NULL;
183:         else
184:             return apNodo->suPieza;
185:     }
186:
187:     void ListaPiezas::Iterar(void (Pieza::*func)() const) const
188:     {
189:         if (!apCabeza)
190:             return;
191:         NodoPieza* apNodo = apCabeza;
192:         do
193:             (apNodo->suPieza->*func)();
194:             while (apNodo = apNodo->suSiguiente);
195:     }
196:
197:     void ListaPiezas::Insertar(Pieza * apPieza)
198:     {
199:         NodoPieza * apNodo = new NodoPieza(apPieza);
200:         NodoPieza * apActual = apCabeza;
201:         NodoPieza * apSiguiente = NULL;
202:         int Nuevo = apPieza->ObtenerNumeroPieza();
203:         int Siguiente = 0;
204:
205:         suCuenta++;
206:         if (!apCabeza)
207:         {
208:             apCabeza = apNodo;
209:             return;
210:         }
211:         // si éste es más pequeño que el nodo cabeza
212:         // se convierte en el nuevo nodo cabeza
```

**LISTADO 15.7** CONTINUACIÓN

```
213:         if (apCabeza->suPieza->ObtenerNumeroPieza() > Nuevo)
214:         {
215:             apNodo->suSiguiente = apCabeza;
216:             apCabeza = apNodo;
217:             return;
218:         }
219:         for (;;)
220:         {
221:             // si no hay siguiente, agregar éste
222:             if (!apActual->suSiguiente)
223:             {
224:                 apActual->suSiguiente = apNodo;
225:                 return;
226:             }
227:             // si va después de éste y antes del siguiente
228:             // entonces insertarlo aquí, de no ser así, obtener el siguiente
229:             apSiguiente = apActual->suSiguiente;
230:             Siguiente = apSiguiente->suPieza->ObtenerNumeroPieza();
231:             if (Siguiente > Nuevo)
232:             {
233:                 apActual->suSiguiente = apNodo;
234:                 apNodo->suSiguiente = apSiguiente;
235:                 return;
236:             }
237:             apActual = apSiguiente;
238:         }
239:     }
240:
241: class CatalogoPiezas : private ListaPiezas
242: {
243: public:
244:     void Insertar(Pieza *);
245:     int Existe(int NumeroPieza);
246:     Pieza * Obtener(int NumeroPieza);
247:     operator+(const CatalogoPiezas &);
248:     void MostrarTodo()
249:     { Iterar(&Pieza::Desplegar); }
250: private:
251: };
252:
253: void CatalogoPiezas::Insertar(Pieza * nuevaPieza)
254: {
255:     int numeroPieza = nuevaPieza->ObtenerNumeroPieza();
256:     int desplazamiento;
257:     if (!Encontrar(desplazamiento, numeroPieza))
258:         ListaPiezas::Insertar(nuevaPieza);
```

```
259:     else
260:     {
261:         cout << numeroPieza << " fue la ";
262:         switch (desplazamiento)
263:         {
264:             case 0: cout << "primera ";
265:                     break;
266:             case 1: cout << "segunda ";
267:                     break;
268:             case 2: cout << "tercera ";
269:                     break;
270:             default: cout << desplazamiento+1 << "a ";
271:         }
272:         cout << "entrada. ¡Rechazada!\n";
273:     }
274: }
275:
276: int CatalogoPiezas::Existe(int NumeroPieza)
277: {
278:     int desplazamiento;
279:
280:     Encontrar(desplazamiento, NumeroPieza);
281:     return desplazamiento;
282: }
283:
284: Pieza * CatalogoPiezas::Obtener(int NumeroPieza)
285: {
286:     int desplazamiento;
287:
288:     return (Encontrar(desplazamiento, NumeroPieza));
289:
290: }
291:
292: int main()
293: {
294:     CatalogoPiezas cp;
295:     Pieza * apPieza = NULL;
296:     int NumeroPieza;
297:     int valor;
298:     int opcion;
299:
300:     while (1)
301:     {
302:         cout << "(0)Salir (1)Auto (2)Avión: ";
303:         cin >> opcion;
304:         if (!opcion)
```



**LISTADO 15.7** CONTINUACIÓN

```
305:         break;
306:     cout << "¿Nuevo NumeroPieza?: ";
307:     cin >> NumeroPieza;
308:     if (opcion == 1)
309:     {
310:         cout << "¿Año del modelo?: ";
311:         cin >> valor;
312:         apPieza = new PiezaAuto(valor, NumeroPieza);
313:     }
314:     else
315:     {
316:         cout << "¿Número de motor?: ";
317:         cin >> valor;
318:         apPieza = new PiezaAeroPlano(valor, NumeroPieza);
319:     }
320:     cp.Insertar(apPieza);
321: }
322: cp.MostrarTodo();
323: return 0;
324: }
325:
```

**SALIDA**

```
(0)Salir (1)Auto (2)Avión: 1
¿Nuevo NumeroPieza?: 1234
¿Año del modelo?: 94
(0)Salir (1)Auto (2)Avión: 1
¿Nuevo NumeroPieza?: 4434
¿Año del modelo?: 93
(0)Salir (1)Auto (2)Avión: 1
¿Nuevo NumeroPieza?: 1234
¿Año del modelo?: 94
1234 fue la primera entrada. ¡Rechazada!
(0)Salir (1)Auto (2)Avión: 1
¿Nuevo NumeroPieza?: 2345
¿Año del modelo?: 93
(0)Salir (1)Auto (2)Avión: 0

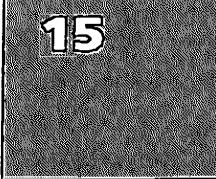
Número de pieza: 1234
Año del modelo: 94

Número de pieza: 2345
Año del modelo: 93

Número de pieza: 4434
Año del modelo: 93
```

**ANÁLISIS** En la línea 76 se declara la clase `ListaPiezas` como amiga de la clase `NodoPieza`.

En este listado se coloca la declaración `friend` en la sección pública, pero esto no es necesario; se puede colocar en cualquier parte de la declaración de la clase sin cambiar el significado de la instrucción. Debido a esta instrucción, todos los datos y funciones miembro privados están disponibles para cualquier función miembro de la clase `ListaPiezas`.



En la línea 149, la implementación de la función miembro `ObtenerPrimero()` refleja este cambio. En lugar de regresar `apCabeza->ObtenerPieza`, esta función ahora puede regresar el que de otra manera sería dato miembro privado escribiendo `apCabeza->suPieza`. De manera similar, la función `Insertar()` ahora puede escribir `apNodo->suSiguiente = apCabeza`, en lugar de escribir `apNodo->AsignarSiguiente(apCabeza)`.

Es cierto que estos son cambios triviales, y no existe un buen motivo para hacer que `ListaPiezas` sea amiga de `NodoPieza`, pero esto sí sirve para mostrar la forma en que funciona la palabra reservada `friend`.

Las declaraciones de clases `friend` se deben usar con extrema precaución. Si dos clases están entrelazadas de manera que es difícil separarlas, y una debe tener acceso a los datos de la otra con frecuencia, puede existir un buen motivo para usar esta declaración. Pero úsela con moderación; por lo general es igual de sencillo utilizar los métodos de acceso público, y esto le permite cambiar una clase sin tener que volver a compilar la otra.

**Nota**

A menudo escuchará a los programadores de C++ novatos quejarse de que las declaraciones `friend` "minan" la encapsulación, que es tan importante para la programación orientada a objetos. Francamente, esto es mentira. La declaración `friend` hace que la clase declarada como amiga sea parte de la interfaz de la clase y no mina la encapsulación más que la derivación pública.

**Clase amiga**

Para declarar una clase como amiga de otra, se coloca la palabra `friend` en la clase que va a otorgar los derechos de acceso. Es decir, yo puedo declararlo a usted como mi amigo, pero usted no puede declararse a usted mismo como mi amigo.

He aquí un ejemplo:

```
class NodoPieza
{
public:
    // declara a ListaPiezas como amiga de NodoPieza
    friend class ListaPiezas;
};
```

## Funciones amigas

Algunas veces necesitará otorgar este nivel de acceso no a toda la clase, sino sólo a una o dos funciones de esa clase. Puede hacer esto declarando a las funciones miembro de la otra clase como amigas, en lugar de declarar a toda la clase como amiga. De hecho, puede declarar a cualquier función, sea o no una función miembro de otra clase, como una función amiga.

## Funciones amigas y sobrecarga de operadores

El listado 15.1 proporciona una clase Cadena que redefine a `operator+`. También proporciona un constructor que toma un apuntador a un carácter constante, para que se puedan crear objetos de tipo cadena a partir de cadenas estilo C. Esto le permite crear una cadena y agregarle una cadena estilo C.


**Tip**

Las cadenas estilo C son arreglos de caracteres con terminador nulo, como  
`char myString [] = "¡Hola, mundo!".`

Lo que no puede hacer es crear una cadena estilo C (una cadena de caracteres) y concatenarle un objeto de tipo cadena, como se muestra en el siguiente ejemplo:

```
char cCadena[] = {"¡Hola"};
Cadena sCadena(", mundo");
Cadena sCadenaDos = cCadena + sCadena; //error!
```

Las cadenas estilo C no tienen un `operator+` sobrecargado. Como se dijo en el día 10, “Funciones avanzadas”, al decir `cCadena + sCadena`; lo que realmente está llamando es a `cCadena.operator+(sCadena)`. Como no puede llamar a `operator+()` en una cadena estilo C, esto produce un error en tiempo de compilación.

Puede solucionar este problema declarando una función `friend` en `Cadena`, lo cual sobrecarga al `operator+` pero toma dos objetos de tipo cadena. El constructor apropiado convertirá a la cadena estilo C en un objeto de tipo cadena, y luego se llamará a `operator+` usando los dos objetos de tipo cadena. El listado 15.8 muestra el uso de un operador `friend`.

**ENTRADA LISTADO 15.8 `operator+` amigable**

```
1: // Listado 15.8 - Operadores amigables
2:
3: #include <iostream.h>
4: #include <string.h>
```

```
5:  
6:  
7: // Clase cadena rudimentaria  
8: class Cadena  
9: {  
10:    public:  
11:        // constructores  
12:        Cadena();  
13:        Cadena(const char *const);  
14:        Cadena(const Cadena &);  
15:        ~Cadena();  
16:        // operadores sobrecargados  
17:        char & operator[](int desplazamiento);  
18:        char operator[](int desplazamiento) const;  
19:        Cadena operator+(const Cadena &);  
20:        friend Cadena operator+(const Cadena &, const Cadena &);  
21:        void operator+=(const Cadena &);  
22:        Cadena & operator=(const Cadena &);  
23:        // Métodos generales de acceso  
24:        int ObtenerLongitud() const  
25:            { return suLongitud; }  
26:        const char * ObtenerCadena() const  
27:            { return suCadena; }  
28:    private:  
29:        Cadena (int); // constructor privado  
30:        char * suCadena;  
31:        unsigned short suLongitud;  
32:    };  
33:  
34: // constructor predeterminado, crea una cadena de 0 bytes  
35: Cadena::Cadena()  
36: {  
37:     suCadena = new char[ 1 ];  
38:     suCadena[ 0 ] = '\0';  
39:     suLongitud = 0;  
40:     // cout << "\tConstructor de cadena predeterminado\n";  
41:     // ConstructorCuenta++;  
42: }  
43:  
44: // constructor privado (auxiliar), lo utilizan sólo  
45: // los métodos de la clase para crear una nueva cadena del  
46: // tamaño requerido. Se llena con caracteres nulos.  
47: Cadena::Cadena(int longitud)  
48: {  
49:     suCadena = new char[ longitud + 1 ];  
50:  
51:     for (int i = 0; i <= longitud; i++)  
52:         suCadena[ i ] = '\0';  
53:     suLongitud = longitud;  
54:     // cout << "\tConstructor de Cadena(int)\n";  
55:     // ConstructorCuenta++;  
56: }
```

**LISTADO 15.8** CONTINUACIÓN

---

```
57:
58:     // Convierte un arreglo de caracteres en una Cadena
59:     Cadena::Cadena(const char * const cCadena)
60:     {
61:         suLongitud = strlen(cCadena);
62:         suCadena = new char[ suLongitud + 1 ];
63:         for (int i = 0; i < suLongitud; i++)
64:             suCadena[ i ] = cCadena[ i ];
65:         suCadena[ suLongitud ] = '\0';
66:         // cout << "\tConstructor de Cadena(char *)\n";
67:         // ConstructorCuenta++;
68:     }
69:
70:     // constructor de copia
71:     Cadena::Cadena (const Cadena & rhs)
72:     {
73:         suLongitud = rhs.ObtenerLongitud();
74:         suCadena = new char[ suLongitud + 1 ];
75:         for (int i = 0; i < suLongitud; i++)
76:             suCadena[ i ] = rhs[ i ];
77:         suCadena[ suLongitud ] = '\0';
78:         // cout << "\tConstructor de Cadena(Cadena&)\n";
79:         // ConstructorCuenta++;
80:     }
81:
82:     // destructor, libera la memoria asignada
83:     Cadena::~Cadena ()
84:     {
85:         delete [] suCadena;
86:         suLongitud = 0;
87:         // cout << "\tDestructor de Cadena\n";
88:     }
89:
90:     // operador igual a, libera la memoria existente
91:     // luego copia la cadena y el tamaño
92:     Cadena& Cadena::operator=(const Cadena & rhs)
93:     {
94:         if (this == &rhs)
95:             return *this;
96:         delete [] suCadena;
97:         suLongitud = rhs.ObtenerLongitud();
98:         suCadena = new char[ suLongitud + 1 ];
99:         for (int i = 0; i < suLongitud; i++)
100:             suCadena[ i ] = rhs[ i ];
101:         suCadena[ suLongitud ] = '\0';
102:         return *this;
103:         // cout << "\tOperador = de Cadena\n";
104:     }
105:
```

```
106: //operador de desplazamiento que no es constante, i regresa
107: // la referencia a un carácter para que se pueda
108: // cambiar!
109: char & Cadena::operator[](int desplazamiento)
110: {
111:     if (desplazamiento > suLongitud)
112:         return suCadena[ suLongitud - 1 ];
113:     else
114:         return suCadena[ desplazamiento ];
115: }
116:
117: // operador de desplazamiento constante para utilizar
118: // en objetos const (vea el constructor de copia)
119: char Cadena::operator[](int desplazamiento) const
120: {
121:     if (desplazamiento > suLongitud)
122:         return suCadena[ suLongitud - 1 ];
123:     else
124:         return suCadena[ desplazamiento ];
125: }
126:
127: // crea una nueva cadena al agregar la cadena
128: // actual a rhs
129: Cadena Cadena::operator+(const Cadena & rhs)
130: {
131:     int longitudTotal = suLongitud + rhs.ObtenerLongitud();
132:     Cadena temp(longitudTotal);
133:     int i, j;
134:
135:     for (i = 0; i < suLongitud; i++)
136:         temp[ i ] = suCadena[ i ];
137:     for (j = 0, i = suLongitud; j<rhs.ObtenerLongitud(); j++, i++)
138:         temp[ i ] = rhs[ j ];
139:     temp[ longitudTotal ]='\'0';
140:     return temp;
141: }
142:
143: // crea una nueva cadena sumando
144: // una cadena a otra
145: Cadena operator+(const Cadena & lhs, const Cadena & rhs)
146: {
147:     int longitudTotal = lhs.ObtenerLongitud() + rhs.ObtenerLongitud();
148:     Cadena temp(longitudTotal);
149:     int i, j;
150:
151:     for (i = 0; i < lhs.ObtenerLongitud(); i++)
152:         temp[ i ] = lhs[ i ];
153:     for (j = 0, i = lhs.ObtenerLongitud();
154:          j < rhs.ObtenerLongitud(); j++, i++)
155:         temp[ i ] = rhs[ j ];
```



**LISTADO 15.8** CONTINUACIÓN

```

156:     temp[ longitudTotal ] = '\0';
157:     return temp;
158: }
159:
160: int main()
161: {
162:     Cadena s1("Cadena Uno ");
163:     Cadena s2("Cadena Dos ");
164:     char *c1 = { "C-Cadena Uno " } ;
165:     Cadena s3;
166:     Cadena s4;
167:     Cadena s5;
168:
169:     cout << "s1: " << s1.ObtenerCadena() << endl;
170:     cout << "s2: " << s2.ObtenerCadena() << endl;
171:     cout << "c1: " << c1 << endl;
172:     s3 = s1 + s2;
173:     cout << "s3: " << s3.ObtenerCadena() << endl;
174:     s4 = s1 + c1;
175:     cout << "s4: " << s4.ObtenerCadena() << endl;
176:     s5 = c1 + s2;
177:     cout << "s5: " << s5.ObtenerCadena() << endl;
178:     return 0;
179: }
```

**SALIDA**

s1: Cadena Uno  
 s2: Cadena Dos  
 c1: C-Cadena Uno  
 s3: Cadena Uno Cadena Dos  
 s4: Cadena Uno C-Cadena Uno  
 s5: C-Cadena Uno Cadena Dos

**ANÁLISIS**

La implementación de todos los métodos de cadena, exceptuando a `operator+`, permanece sin cambio, quedando igual que en el listado 15.1. En la línea 20 se sobrecarga un nuevo `operator+` para tomar dos referencias constantes a una cadena y para regresar una cadena, y esta función se declara como amiga.

Observe que este `operator+` no es una función miembro de ésta ni de ninguna otra clase. Se declara dentro de la declaración de la clase `Cadena` sólo para que se pueda hacer amiga, pero como se declara, no se necesita otro prototipo de función.

La implementación de este `operator+` se encuentra en las líneas 145 a 158. Observe que es similar al `operator+` anterior, excepto que toma dos cadenas y tiene acceso a ellas a través de sus métodos de acceso público.

El programa controlador muestra el uso de esta función en la línea 176, en donde `operator+` ahora se llama para actuar sobre ¡una cadena estilo C!

### Funciones amigas

Para declarar una función como amiga se usa la palabra reservada `friend` y luego la especificación completa de la función. Declarar una función como amiga no le da a la función `friend` acceso a su apuntador `this`, pero sí proporciona un acceso completo a todos los datos y funciones miembro protegidos y privados.

He aquí un ejemplo:

```
class NodoPieza
{
    // ...
    // hacer que otra función miembro de la clase sea una amiga
    friend void ListaPiezas::Insertar(Pieza *);
    // hacer que una función global sea amiga
    friend int UnaFuncion();
    // ...
};
```

## Sobrecarga del operador de inserción

Finalmente está listo para dar a su clase `Cadena` la capacidad de utilizar `cout` de la misma manera que cualquier otro tipo. Hasta ahora, cuando quería imprimir una cadena, estaba obligado a escribir lo siguiente:

```
cout << laCadena.ObtenerCadena();
```

Lo que puede hacer en vez de esto es escribir lo siguiente:

```
cout << laCadena;
```

Para lograr esto, debe redefinir a `operator<<()`. El día 16, "Flujos", presenta los detalles del trabajo con `iostream`; por ahora, el listado 15.9 muestra cómo se puede sobrecargar `operator<<` por medio de una función `friend`.

### ENTRADA LISTADO 15.9 Sobrecarga de operator<<()

```
1: // Listado 15.9 - Sobrecarga del operador <<
2:
3: #include <iostream.h>
4: #include <string.h>
5:
6:
7: class Cadena
8: {
9: public:
```

continúa

**LISTADO 15.9 CONTINUACIÓN**

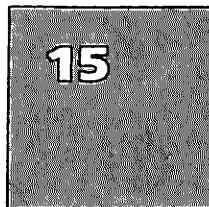
```
10:    // constructores
11:    Cadena();
12:    Cadena(const char * const);
13:    Cadena(const Cadena &);
14:    ~Cadena();
15:    // operadores sobrecargados
16:    char & operator[](int desplazamiento);
17:    char operator[](int desplazamiento) const;
18:    Cadena operator+(const Cadena &);
19:    void operator+=(const Cadena &);
20:    Cadena & operator=(const Cadena &);
21:    friend ostream & operator<<
22:        (ostream & elFlujo, Cadena & laCadena);
23:    // Métodos generales de acceso
24:    int ObtenerLongitud()const
25:    {
26:        return suLongitud;
27:    }
28:    const char * ObtenerCadena() const
29:    {
28:        return suCadena;
29:    }
30: private:
31:     Cadena (int); // constructor privado
32:     char * suCadena;
33:     unsigned short suLongitud;
34: };
35: // constructor predeterminado, crea una cadena de 0 bytes
36: Cadena::Cadena()
37: {
38:     suCadena = new char[ 1 ];
39:     suCadena[ 0 ] = '\0';
40:     suLongitud = 0;
41:     // cout << "\tConstructor de cadena predeterminado\n";
42:     // ConstructorCuenta++;
43: }
44: // constructor privado (auxiliar), lo utilizan sólo
45: // los métodos de la clase para crear una nueva cadena del
46: // tamaño requerido. Se llena con caracteres nulos.
47: Cadena::Cadena(int longitud)
48: {
49:     suCadena = new char[ longitud + 1 ];
50:     for (int i = 0; i <= longitud; i++)
51:         suCadena[ i ] = '\0';
52:     suLongitud = longitud;
53:     // cout << "\tConstructor de Cadena(int)\n";
54:     // ConstructorCuenta++;
55: }
56: // Convierte un arreglo de caracteres en una Cadena
```

```
58:     Cadena::Cadena(const char * const cCadena)
59:     {
60:         suLongitud = strlen(cCadena);
61:         suCadena = new char[ suLongitud + 1 ];
62:         for (int i = 0; i < suLongitud; i++)
63:             suCadena[ i ] = cCadena[ i ];
64:             suCadena[ suLongitud ] = '\0';
65:             // cout << "\tConstructor de Cadena(char*)\n";
66:             // ConstructorCuenta++;
67:     }
68:
69:     // constructor de copia
70:     Cadena::Cadena(const Cadena & rhs)
71:     {
72:         suLongitud = rhs.ObtenerLongitud();
73:         suCadena = new char[ suLongitud + 1 ];
74:         for (int i = 0; i < suLongitud; i++)
75:             suCadena[ i ] = rhs[ i ];
76:             suCadena[ suLongitud ] = '\0';
77:             // cout << "\tConstructor de Cadena(Cadena&)\n";
78:             // ConstructorCuenta++;
79:     }
80:
81:     // destructor, libera la memoria asignada
82:     Cadena::~Cadena()
83:     {
84:         delete [] suCadena;
85:         suLongitud = 0;
86:         // cout << "\tDestructor de Cadena\n";
87:     }
88:
89:     // operador igual a, libera la memoria existente
90:     // luego copia la cadena y el tamaño
91:     Cadena& Cadena::operator=(const Cadena & rhs)
92:     {
93:         if (this == &rhs)
94:             return *this;
95:         delete [] suCadena;
96:         suLongitud = rhs.ObtenerLongitud();
97:         suCadena = new char[ suLongitud + 1 ];
98:         for (int i = 0; i < suLongitud; i++)
99:             suCadena[ i ] = rhs[ i ];
100:            suCadena[ suLongitud ] = '\0';
101:            return *this;
102:            // cout << "\tOperador = de Cadena\n";
103:    }
104:
105:   // operador de desplazamiento que no es constante, i regresa
106:   // la referencia a un carácter para que se pueda
```

**LISTADO 15.9 CONTINUACIÓN**

```
107:    // cambiar!
108:    char & Cadena::operator[](int desplazamiento)
109:    {
110:        if (desplazamiento > suLongitud)
111:            return suCadena[ suLongitud - 1 ];
112:        else
113:            return suCadena[ desplazamiento ];
114:    }
115:
116:    // operador de desplazamiento constante para utilizar
117:    // en objetos const (vea el constructor de copia)
118:    char Cadena::operator[](int desplazamiento) const
119:    {
120:        if (desplazamiento > suLongitud)
121:            return suCadena[ suLongitud - 1 ];
122:        else
123:            return suCadena[ desplazamiento ];
124:    }
125:
126:    // crea una nueva cadena al agregar la cadena
127:    // actual a rhs
128:    Cadena Cadena::operator+(const Cadena & rhs)
129:    {
130:        int longitudTotal = suLongitud + rhs.ObtenerLongitud();
131:        Cadena temp(longitudTotal);
132:        int i, j;
133:
134:        for (i = 0; i < suLongitud; i++)
135:            temp[ i ] = suCadena[ i ];
136:        for (j = 0; j < rhs.ObtenerLongitud(); j++, i++)
137:            temp[ i ] = rhs[ j ];
138:        temp[ longitudTotal ] = '\0';
139:        return temp;
140:    }
141:
142:    // cambia la cadena actual, no regresa nada
143:    void Cadena::operator+=(const Cadena & rhs)
144:    {
145:        unsigned short rhsLong = rhs.ObtenerLongitud();
146:        unsigned short longitudTotal = suLongitud + rhsLong;
147:        Cadena temp(longitudTotal);
148:        int i, j;
149:
150:        for (i = 0; i < suLongitud; i++)
151:            temp[ i ] = suCadena[ i ];
152:        for (j = 0, i = 0; j < rhs.ObtenerLongitud(); j++, i++)
153:            temp[ i ] = rhs[ i - suLongitud ];
154:        temp[ longitudTotal ] = '\0';
155:        *this = temp;
156:    }
157:
158:    // int Cadena::ConstructorCuenta =
```

```
159:     ostream & operator<<(ostream & elFlujo, Cadena & laCadena)
160:     {
161:         elFlujo << laCadena.suCadena;
162:         return elFlujo;
163:     }
164:
165:     int main()
166:     {
167:         Cadena laCadena("¡Hola, mundo!");
168:         cout << laCadena;
169:         cout << endl;
170:         return 0;
171:     }
```

15

**SALIDA** ¡Hola, mundo!

**ANÁLISIS** En la línea 21 se declara a `operator<<` como función `friend` que toma una referencia a `ostream` y una referencia a `Cadena` y luego regresa una referencia a `ostream`. Observe que ésta no es una función miembro de `Cadena`. Regresa una referencia a un objeto `ostream` para que usted pueda concatenar las llamadas a `operator<<`, como en el siguiente ejemplo:

```
cout << "miEdad: " << suEdad << " años.;"
```

La implementación de esta función `friend` se encuentra en las líneas 159 a 163. Todo lo que esto hace en realidad es ocultar los detalles de implementación relacionados con la forma de proporcionar la cadena al `ostream`, y así es como debe ser. Verá más acerca de la sobrecarga de este operador y de `operator>>` en el día 16.

## Resumen

Hoy vio cómo delegar la funcionalidad a un objeto de una clase contenida. También vio cómo implementar una clase con base en otra mediante el uso de la contención o de la herencia privada. La contención está restringida en cuanto a que la nueva clase no tiene acceso a los miembros protegidos de la clase contenida, y no puede redefinir a las funciones miembro de la clase contenida. La contención es más simple de usar que la herencia privada, y se debe utilizar siempre que sea posible.

También vio cómo declarar funciones y clases amigas. Usando una función amiga, aprendió a sobrecargar el operador `<<`, para permitir que sus clases utilicen `cout` de la misma manera que lo hacen las clases integradas.

Recuerde que la herencia pública expresa una relación de tipo *es un*, la contención expresa una relación de tipo *tiene un*, y la herencia privada expresa *implementado con base en*. La relación *se delega a* se puede expresar ya sea por medio de la contención o de la herencia privada, aunque es más común la contención.

## Preguntas y respuestas

- P** ¿Por qué es tan importante distinguir entre relaciones de tipo *es un*, *tiene un*, e *implementado con base en*?
- R** El objetivo de C++ es implementar programas orientados a objetos bien diseñados. Mantener estas relaciones como debe ser asegura que su diseño corresponda a la realidad de lo que está modelando. Además, un diseño bien entendido tendrá una mayor probabilidad de reflejarse en un código bien diseñado.
- P** ¿Por qué se prefiere la contención a la herencia privada?
- R** El reto en la programación moderna es hacer frente a la complejidad. Entre más se puedan utilizar los objetos como si fueran cajas negras, habrá menos detalles de los cuales preocuparse y se podrá manejar una mayor complejidad. Las clases contenidas ocultan sus detalles; la herencia privada expone los detalles de implementación.
- P** ¿Por qué no hacer que todas las clases sean amigas de todas las clases que utilizan?
- R** Hacer que una clase sea amiga de otra expone los detalles de implementación y reduce la encapsulación. Lo ideal es mantener ocultos tantos detalles de cada clase como sea posible.
- P** Si una función está sobrecargada, ¿es necesario declarar cada forma de la función para que sea amiga?
- R** Sí, si sobrecarga una función y la declara como amiga de otra clase, debe declarar `friend` por cada forma en que quiera otorgar el acceso.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Cómo se establece una relación de tipo *es un*?
2. ¿Cómo se establece una relación de tipo *tiene un*?
3. ¿Cuál es la diferencia entre contención y delegación?
4. ¿Cuál es la diferencia entre delegación e *implementación con base en*?
5. ¿Qué es una función `friend`?

6. ¿Qué es una clase **friend**?
7. Si **Perro** es amigo de **Muchacho**, ¿**Muchacho** es amigo de **Perro**?
8. Si **Perro** es amigo de **Muchacho** y **Terrier** se deriva de **Perro**, ¿**Terrier** es amigo de **Muchacho**?
9. Si **Perro** es amigo de **Muchacho** y **Muchacho** es amigo de **Casa**, ¿**Perro** es amigo de **Casa**?
10. ¿Dónde debe aparecer la declaración de una función **friend**?

## Ejercicios

1. Muestre la declaración de una clase llamada **Animal**, que contenga un dato miembro que sea un objeto de tipo cadena.
2. Muestre la declaración de una clase llamada **ArregloLimitado**, que sea un arreglo.
3. Muestre la declaración de una clase llamada **Conjunto**, que se declare con base en un arreglo.
4. Modifique el listado 15.1 para proporcionar a la clase **Cadena** un operador de inserción (<<).
5. **CAZA ERRORES:** ¿Qué está mal en este programa?

```
1:      #include <iostream.h>
2:
3:      class Animal;
4:
5:      void asignarValor(Animal & , int);
6:
7:
8:      class Animal
9:      {
10:         public:
11:             int ObtenerPeso()const { return suPeso; }
12:             int ObtenerEdad() const { return suEdad; }
13:         private:
14:             int suPeso;
15:             int suEdad;
16:     };
17:
18:     void asignarValor(Animal & elAnimal, int elPeso)
19:     {
20:         friend class Animal;
21:         elAnimal.suPeso = elPeso;
22:     }
23:
24:     int main()
25:     {
26:         Animal peppy;
27:         asignarValor(peppy,5);
28:     }
```

6. Corrija el listado del ejercicio 5 para que se pueda compilar.

7. **CAZA ERRORES:** ¿Qué está mal en este código?

```
1:      #include <iostream.h>
2:
3:      class Animal;
4:
5:      void asignarValor(Animal & , int);
6:      void asignarValor(Animal & , int, int);
7:
8:      class Animal
9:      {
10:          friend void asignarValor(Animal & , int);
11:      private:
12:          int suPeso;
13:          int suEdad;
14:      };
15:
16:      void asignarValor(Animal& elAnimal, int elPeso)
17:      {
18:          elAnimal.suPeso = elPeso;
19:      }
20:
21:
22:      void asignarValor(Animal & elAnimal, int elPeso, int laEdad)
23:      {
24:          elAnimal.suPeso = elPeso;
25:          elAnimal.suEdad = laEdad;
26:      }
27:
28:      int main()
29:      {
30:          Animal peppy;
31:          asignarValor(peppy,5);
32:          asignarValor(peppy,7,9);
33:      }
```

8. Corrija el ejercicio 7 para que se pueda compilar.

# SEMANA 3

Día 16

## Flujos

Hasta ahora ha utilizado `cout` para escribir en la pantalla y `cin` para leer desde el teclado, sin comprender completamente cómo funcionan. Hoy aprenderá lo siguiente:

- Qué son los flujos y cómo se utilizan
- Cómo manejar la entrada y la salida por medio de flujos
- Cómo escribir en archivos y leerlos por medio de flujos

## Panorama general sobre los flujos

C++, como parte del lenguaje, no define cómo se escriben los datos en la pantalla o en un archivo, ni cómo se leen los datos en un programa. Sin embargo, éstas son, evidentemente, partes esenciales del trabajo con C++, y la biblioteca estándar de C++ incluye la biblioteca `iostream`, que facilita la entrada y la salida (E/S).

La ventaja de tener la entrada y salida separadas del lenguaje y manejarlas en bibliotecas es que es más fácil hacer que el lenguaje sea “independiente de la plataforma”. Es decir, puede escribir programas de C++ en una PC y volver a compilarlos y ejecutarlos en una estación de trabajo Sun. O puede compilar por medio del compilador GNU en la PC (por ejemplo, estando en su oficina) y llevarse ese código a su casa para utilizarlo en Linux. El creador del compilador provee la biblioteca apropiada, y todo funciona. Al menos ésa es la teoría general.

**Nota**

Una biblioteca es una colección de archivos objeto (.o en Linux, .obj en una PC) que se pueden enlazar con su programa para proporcionar una funcionalidad adicional. Ésta es la forma más básica de la reutilización de código y se ha estado utilizando desde que los primeros programadores manejaban las tarjetas perforadas para interpretar los 0s y 1s.

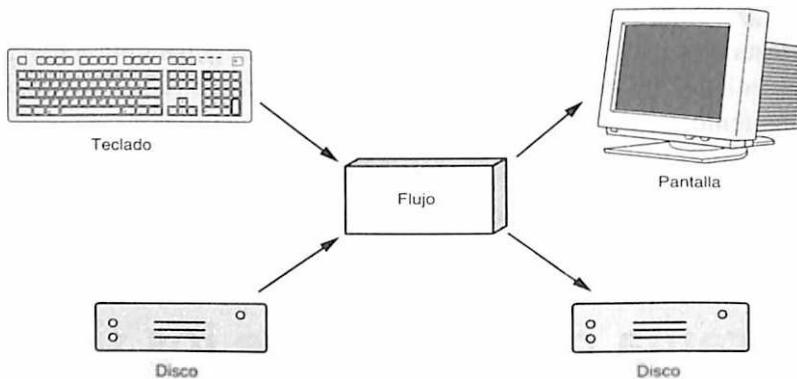
## Encapsulación

Las clases `iostream` ven el conjunto de datos que va desde su programa hasta la pantalla como un flujo de datos, un byte detrás de otro. Si el destino del flujo es un archivo o la pantalla, el origen por lo general es alguna parte de su programa. Si el flujo se invierte, los datos pueden venir desde el teclado o un archivo en disco y “verterse” en sus variables de datos.

Uno de los objetivos principales de los flujos es encapsular los problemas relacionados con el envío y recepción de los datos desde y hacia el disco o la pantalla. Después de crear un flujo, su programa trabaja con el flujo y éste se encarga de los detalles. La figura 16.1 ilustra esta idea elemental.

**FIGURA 16.1**

*Encapsulación por medio de flujos.*



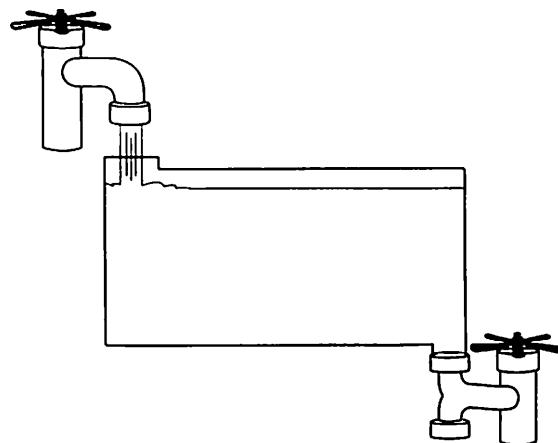
## Almacenamiento en búfer

Escribir en el disco (y en la pantalla, aunque en menor extensión) es muy “costoso”. Lleva mucho tiempo (relativamente hablando) escribir información en el disco o leer información del disco, y la ejecución del programa por lo general se bloquea debido a las lecturas y escrituras de disco. Para solucionar este problema, los flujos proporcionan el “almacenamiento en búfer”. La información se escribe en el flujo, pero no se escribe inmediatamente en el disco. En vez de esto, el búfer del flujo se va llenando, y cuando está lleno, escribe todo en el disco de una sola vez.

Imagine un tanque que se llena de agua por medio de una válvula que está en la parte superior del tanque, y el nivel de agua sube pero no sale agua por la válvula que se encuentra en la parte inferior del tanque. La figura 16.2 ilustra esto.

**FIGURA 16.2**

*Llenado del búfer.*

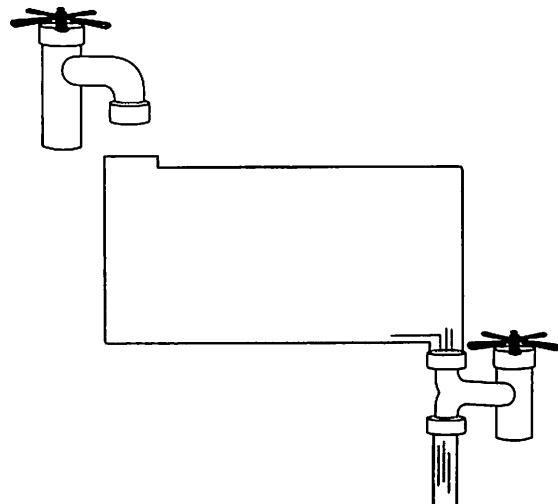


16

Cuando el agua (datos) llega hasta arriba, la válvula inferior se abre y el agua sale rápidamente. La figura 16.3 ilustra esto.

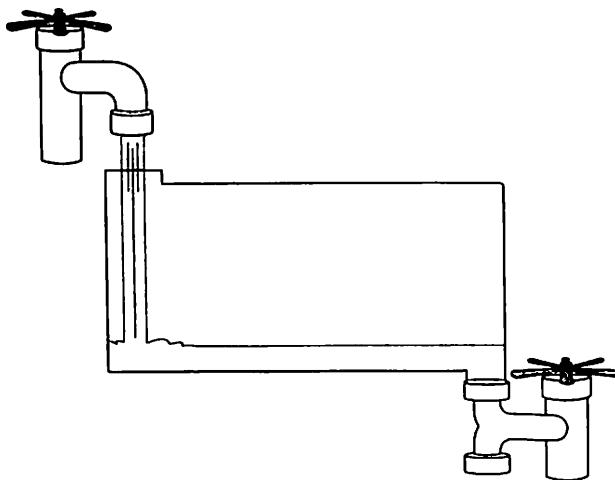
**FIGURA 16.3**

*Vaciado del búfer.*



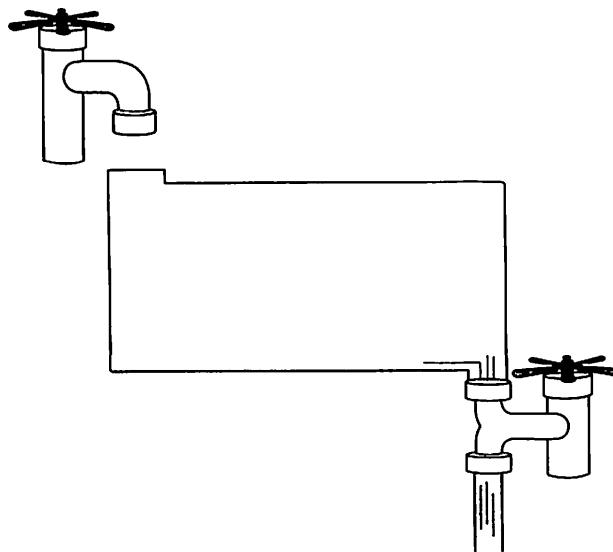
Cuando el búfer queda vacío, la válvula inferior se cierra, la válvula superior se abre y fluye más agua hacia el tanque búfer, como se muestra en la figura 16.4.

**FIGURA 16.4**  
*Rellenado del búfer.*



De vez en cuando se necesita sacar el agua del tanque incluso antes de que se llene por completo. Esto se conoce como “limpiar el búfer”. La figura 16.5 ilustra esto.

**FIGURA 16.5**  
*Limpieza del búfer.*



## Flujos y búferes

Como es de esperarse, C++ se basa en el método orientado a objetos para implementar los flujos y los búferes.

- La clase `streambuf` maneja el búfer, y sus métodos (funciones miembro) proporcionan la capacidad para llenar, vaciar, limpiar y manejar de cualquier otra forma el búfer.

- La clase `ios` es la clase base para las clases de flujos de entrada y salida. La clase `ios` tiene un objeto `streambuf` como variable miembro.
- Las clases `istream` y `ostream` se derivan de la clase `ios` y especializan el comportamiento de los flujos de entrada y salida, respectivamente.
- La clase `iostream` se deriva tanto de `istream` como de `ostream` y proporciona métodos de entrada y salida para escribir en la pantalla.
- Las clases `fstream` proporcionan entrada y salida desde archivos.

16

## Objetos de E/S estándar

Cuando inicia un programa de C++ que incluye la clase `iostream`, se crean e inicializan cuatro objetos:

- `cin` (se pronuncia "si-in") maneja la entrada desde el teclado, que viene siendo la entrada estándar.
- `cout` (se pronuncia "si-out") maneja la salida estándar, que viene siendo la salida a la pantalla.
- `cerr` (se pronuncia "si-err") maneja la salida que no se encuentra en el búfer al dispositivo de error estándar, la pantalla. Como esta salida no se encuentra en el búfer, todo lo que se envíe a `cerr` se escribe inmediatamente en el dispositivo de error estándar, sin esperar que el búfer se llene o que se reciba un comando de limpieza.
- `clog` (se pronuncia como "si-log") maneja los mensajes de error que se encuentran en el búfer y que se envían como salida al dispositivo de error estándar, la pantalla. Es común que esto se "redireccione" a un archivo de registro, como se describe en la siguiente sección.

### Nota

La biblioteca de clases `iostream` se agrega automáticamente al programa por medio del compilador. Todo lo que necesita hacer para utilizar estas funciones es colocar la instrucción `#include` apropiada al principio del listado de su programa.

## Redirección

Cada uno de los dispositivos estándar, entrada, salida y error, se pueden redireccionar a otros dispositivos. Por lo general, el error estándar se redirecciona a un archivo, y la entrada y salida estándar se pueden canalizar hacia archivos usando comandos del sistema operativo.

La *redirección* se refiere a enviar la salida (o entrada) hacia un lugar distinto al predeterminado. Los operadores de redirección para Linux (así como para otras versiones de UNIX y DOS) son (`<`) redirigir entrada y (`>`) redirigir salida.

La *canalización* se refiere al redireccionamiento de la salida de un programa como la entrada de otro.

Linux (y otras versiones de UNIX) proporciona capacidades de redirección avanzadas: redirigir salida (>), redirigir entrada (<), y redirigir salida hacia la entrada de un segundo programa (!). (La redirección en UNIX es muy particular, porque el sistema “ve” a todos los dispositivos como archivos; así, puede redireccionar la salida hacia la pantalla o hacia un archivo de texto, pero también la puede redireccionar hacia el módem, un disco duro o la tarjeta de sonido.)

DOS proporciona comandos de redirección rudimentarios, como redirigir salida (>) y redirigir entrada (<). También permite redirigir la salida de un programa hacia la entrada de otro con el carácter (!). La idea general es la misma en DOS, Linux y UNIX: tomar la salida dirigida a la pantalla y escribirla en un archivo, o canalizarla hacia otro programa. De manera alternativa, la entrada de un programa se puede extraer de un archivo en lugar del teclado.

La redirección es más una función del sistema operativo que de las bibliotecas `iostream`. C++ sólo proporciona acceso a los cuatro dispositivos estándar; queda a elección del usuario redireccionar los dispositivos a cualquier alternativa que sea necesaria.

## Entrada por medio de `cin`

EL objeto global llamado `cin` es responsable de la entrada y está disponible para su programa al incluir a `iostream.h`. En ejemplos anteriores utilizó el operador de inserción sobrecargado (`<<`) para manipular datos en las variables de su programa. ¿Cómo funciona esto? La sintaxis, como tal vez recuerde, es la siguiente:

```
int unaVariable;  
cout << "Escriba un número: ";  
cin >> unaVariable;
```

Hablaremos sobre el objeto global `cout` más adelante en este día; por ahora, enfoquémonos en la tercera línea, `cin >> unaVariable;`. ¿Puede adivinar lo que hace `cin`?

Evidentemente, debe ser un objeto global ya que no lo definió en su propio código. De la experiencia anterior con los operadores, sabe que `cin` ha sobrecargado el operador de extracción (`>>`) y que el efecto es escribir en la variable local llamada `unaVariable` cualquier información que `cin` tenga en su búfer.

Lo que tal vez no sea inmediatamente obvio es que `cin` ha sobrecargado el operador de extracción para una gran variedad de parámetros, entre los cuales se encuentran `int&`, `short&`, `long&`, `double&`, `float&`, `char&`, `char*`, etc. Al escribir `cin >> unaVariable;`, se valora el tipo de `unaVariable`. En el ejemplo anterior `unaVariable` es de tipo entero, por lo que se llama a la siguiente función:

```
istream & operator>> (int &)
```

Observe que como el parámetro se pasa por referencia, el operador de extracción puede actuar en la variable original. El listado 16.1 muestra el uso de `cin`.

**ENTRADA LISTADO 16.1 cin maneja tipos de datos distintos**

```

1: //Listado 16.1 - cadenas de caracteres y cin
2:
3: #include <iostream.h>
4:
5:
6: int main()
7: {
8:     int miInt;
9:     long miLong;
10:    double miDouble;
11:    float miFloat;
12:    unsigned int miUnsigned;
13:
14:    cout << "int: ";
15:    cin >> miInt;
16:    cout << "Long: ";
17:    cin >> miLong;
18:    cout << "Double: ";
19:    cin >> miDouble;
20:    cout << "Float: ";
21:    cin >> miFloat;
22:    cout << "Unsigned: ";
23:    cin >> miUnsigned;
24:
25:    cout << "\n\nInt:\t" << miInt << endl;
26:    cout << "Long:\t" << miLong << endl;
27:    cout << "Double:\t" << miDouble << endl;
28:    cout << "Float:\t" << miFloat << endl;
29:    cout << "Unsigned:\t" << miUnsigned << endl;
30:    return 0;
31: }
32:
```

**16****SALIDA**

```

int: 2
Long: 70000
Double: 987654321
Float: 3.33
Unsigned: 25

Int:      2
Long:    70000
Double:  9.87654e+08
Float:   3.33
Unsigned: 25

```

**ANÁLISIS** En las líneas 8 a 12 se declaran variables de varios tipos. En las líneas 14 a 23 se pide al usuario que escriba valores para estas variables, y los resultados se imprimen (por medio de cout) en las líneas 25 a 29.

La salida refleja que las variables se colocaron en los “tipos” correctos de variables, y el programa funciona como se espera.

## Cadenas

cin también puede manejar apuntadores a caracteres (char\*) como argumentos; por lo tanto, usted puede crear un búfer de caracteres y utilizar cin para llenarlo. Por ejemplo, puede escribir lo siguiente:

```
char SuNombre[ 50 ]
cout << "Escriba su nombre: ";
cin >> SuNombre;
```

Si escribe “Jesse”, la variable SuNombre se llenará con los caracteres J, e, s, s, e, \0. El último carácter es un *carácter nulo*; cin termina automáticamente la cadena con un carácter nulo, y usted debe tener suficiente espacio en el búfer para toda la cadena más el carácter nulo. Este carácter significa “fin de cadena” para las funciones de la biblioteca estándar, las cuales se explican en el día 21, “Qué sigue”.



### Nota

El carácter nulo ('\0') es distinto del apuntador NULL. Pueden contener el mismo valor en memoria (ceros binarios), pero sirven para distintos propósitos. Debido a eso, se deben tratar de distinta manera. No asigne el valor '\0' a un apuntador. No termine una cadena con la constante NULL.

## Problemas con cadenas

Después de todo este éxito con cin, podría sorprenderse al tratar de escribir un nombre completo en una cadena. cin cree que el espacio en blanco es un separador. Cuando ve un espacio o un carácter de nueva línea, da por hecho que la entrada para el parámetro está completa y, en el caso de las cadenas, agrega un carácter nulo justo ahí. El listado 16.2 muestra este problema.

### ENTRADA LISTADO 16.2 Muestra qué pasa al tratar de escribir más de una palabra con cin

```
1: //Listado 16.2 - Cadenas de caracteres y cin
2:
3: #include <iostream.h>
4:
5:
6: int main()
7: {
8:     char SuNombre[ 50 ];
9:
10:    cout << "Su primer nombre: ";
11:    cin >> SuNombre;
12:    cout << "Aquí está: " << SuNombre << endl;
13:    cout << "Su nombre completo: ";
14:    cin >> SuNombre;
15:    cout << "Aquí está: " << SuNombre << endl;
16:    return 0;
17: }
```

**SALIDA**

```
Su primer nombre: Jesse
Aqui está: Jesse
Su nombre completo: Jesse Liberty
Aqui está: Jesse
```

**ANÁLISIS**

En la línea 8 se crea un arreglo de caracteres para guardar la entrada del usuario. En la línea 10 se pide al usuario que escriba un nombre, y éste se guarda correctamente, como se muestra en la salida.

En la línea 13 se pide otra vez al usuario que escriba, esta vez un nombre completo. `cin` lee la entrada y, cuando ve el espacio entre los nombres, coloca un carácter nulo después de la primera palabra y termina la entrada. Esto no es exactamente lo que se espera que haga el programa.

Para entender por qué esto funciona así, examine el listado 16.3, el cual muestra la entrada para varios campos.

**16****ENTRADA** **LISTADO 16.3** Entrada múltiple

```
1: //Listado 16.3 - Cadenas de caracteres y cin
2:
3: #include <iostream.h>
4:
5:
6: int main()
7: {
8:     int miInt;
9:     long miLong;
10:    double miDouble;
11:    float miFloat;
12:    unsigned int miUnsigned;
13:    char miPalabra[ 50 ];
14:
15:    cout << "int: ";
16:    cin >> miInt;
17:    cout << "Long: ";
18:    cin >> miLong;
19:    cout << "Double: ";
20:    cin >> miDouble;
21:    cout << "Float: ";
22:    cin >> miFloat;
23:    cout << "Palabra: ";
24:    cin >> miPalabra;
25:    cout << "Unsigned: ";
26:    cin >> miUnsigned;
27:
28:    cout << "\n\nInt:\t" << miInt << endl;
29:    cout << "Long:\t" << miLong << endl;
30:    cout << "Double:\t" << miDouble << endl;
31:    cout << "Float:\t" << miFloat << endl;
```

*continúa*

**LISTADO 16.3** CONTINUACIÓN

---

```

32:     cout << "Palabra:\t" << miPalabra << endl;
33:     cout << "Unsigned:\t" << miUnsigned << endl;
34:     cout << "\n\nInt, Long, Double, Float, Palabra, Unsigned: ";
35:
36:     cin >> miInt >> miLong >> miDouble;
37:     cin >> miFloat >> miPalabra >> miUnsigned;
38:     cout << "\n\nInt:\t" << miInt << endl;
39:     cout << "Long:\t" << miLong << endl;
40:     cout << "Double:\t" << miDouble << endl;
41:     cout << "Float:\t" << miFloat << endl;
42:     cout << "Palabra:\t" << miPalabra << endl;
43:     cout << "Unsigned:\t" << miUnsigned << endl;
44:     return 0;
45: }
46:

```

---

**SALIDA**

```

int: 2
Long: 30303
Double: 393939397834
Float: 3.33
Palabra: Hola
Unsigned: 85

Int: 2
Long: 30303
Double: 3.93939e+11
Float: 3.33
Palabra: Hola
Unsigned: 85

Int, Long, Double, Float, Palabra, Unsigned: 3 304938 393847473 6.66
adiós -2

Int: 3
Long: 304938
Double: 3.93847e+08
Float: 6.66
Palabra: adiós
Unsigned: 4294967294

```

**ANÁLISIS**

Una vez más se crean varias variables, esta vez incluyendo un arreglo de tipo `char`. Se pide al usuario la entrada, y la salida se imprime fielmente.

En la línea 34 se pide al usuario toda la entrada a la vez, y luego cada “palabra” de entrada se asigna a la variable apropiada. Para facilitar este tipo de asignación múltiple, `cin` debe considerar cada palabra de la entrada como la entrada completa para cada variable. Si `cin` considerara toda la entrada como parte de la entrada de una variable, este tipo de entrada concatenada sería imposible.

Observe en la línea 34 que el último objeto requerido fue un entero sin signo, pero el usuario escribió -2. Como `cin` piensa que está escribiendo a un entero sin signo, el patrón de bits de -2 se evaluó como entero sin signo, y en la línea 43 `cout` despliega el valor 4294967294. El valor 4294967294 sin signo tiene el patrón de bits exacto del valor -2 con signo.

Más adelante en esta lección verá cómo escribir una cadena completa en un búfer, incluyendo varias palabras. Por ahora, surge la pregunta, “¿cómo maneja el operador de extracción este truco de concatenación?”

16

## **operator>> regresa una referencia a un objeto `istream`**

El valor de retorno de `cin` es una referencia a un objeto `istream`. Como `cin` es en sí un objeto `istream`, el valor de retorno de una operación de extracción puede ser la entrada a la siguiente extracción.

```
int VarUno, varDos, varTres;  
cout << "Escriba tres números: "  
cin >> VarUno >> varDos >> varTres;
```

Al escribir `cin >> VarUno >> varDos >> varTres;`, se evalúa la primera extracción (`cin >> VarUno`). El valor de retorno de esto es otro objeto `istream`, y el operador de extracción de ese objeto recibe la variable `varDos`. Es como si hubiera escrito lo siguiente:

```
((cin >> VarUno) >> varDos) >> varTres;
```

Verá esta técnica otra vez, más adelante cuando hablemos sobre `cout`.

## **Otras funciones miembro de `cin`**

Además de sobrecargar a `operator>>`, `cin` tiene otras funciones miembro. Éstas se utilizan cuando se requiere de un control más fino sobre la entrada.

### **Entrada de un solo carácter**

El `operator>>` que toma una referencia a un carácter se puede utilizar para recibir un solo carácter desde la entrada estándar. La función miembro `get()` también se puede utilizar para recibir un solo carácter, y lo puede hacer de dos formas: `get()` se puede utilizar sin parámetros, en cuyo caso se utiliza el valor de retorno, o se puede utilizar con una referencia a un carácter.

### **Uso de `get()` sin parámetros**

La primer forma de `get()` es sin parámetros. Ésta regresa el valor del carácter encontrado y regresará EOF (fin de archivo) si se llega al final del archivo. `get()` sin parámetros no se usa muy a menudo. No es posible concatenar diversas entradas con el método `get()`, ya que el valor de retorno no es un objeto `iostream`. Por lo tanto, lo siguiente no funcionará:

```
cin.get() >> miVarUno >> miVarDos; // illegal
```

El valor de retorno de `cin.get() >> miVarUno` es un entero, no un objeto `iostream`.

En el listado 16.4 se muestra un uso común de `get()` sin parámetros.

**ENTRADA****LISTADO 16.4** Uso de `get()` sin parámetros

```
1: // Listado 16.4 - Uso de get() sin parámetros
2: #include <iostream.h>
3:
4: int main()
5: {
6:     char ch;
7:     while ((ch = cin.get()) != EOF)
8:     {
9:         cout << "ch: " << ch << endl;
10:    }
11:    cout << "\nListo!\n";
12:    return 0;
13: }
```

**Nota**

En muchas plataformas (como DOS), necesita oprimir "Entrar" antes de que se muestre alguno de los caracteres. Algunas plataformas muestran cada carácter a medida que se escribe.

Para salir de este programa, debe enviar una señal de fin de archivo desde el teclado. En Linux se utiliza Ctrl+D; en equipos DOS utilice Ctrl+Z.

**SALIDA**

```
Hola
ch: H
ch: o
ch: l
ch: a
ch:

mundo
ch: m
ch: u
ch: n
ch: d
ch: o
ch:

(ctrl-d)
Listo!
```

**ANÁLISIS** En la línea 6 se declara una variable local de tipo carácter. El ciclo `while` le asigna a `ch` la entrada recibida de `cin.get()`, y si no es EOF, se imprime la cadena. Sin embargo, esta salida se envía a un búfer hasta que se lee un fin de línea. Al encontrarse con EOF (al oprimir Ctrl+D en Linux, o Ctrl+Z en un equipo DOS), el ciclo termina.

Hay que tener en cuenta que no todas las implementaciones de `istream` soportan esta versión de `get()`, aunque ahora es parte del estándar ANSI/ISO. Desde luego que los compiladores GNU sí la soportan.

16

### Uso de `get()` con una referencia a un carácter como parámetro

Cuando se pasa un carácter como parámetro a `get()`, ese carácter se llena con el siguiente carácter del flujo de entrada. El valor de retorno es un objeto `iostream`, por lo que esta forma de `get()` puede concatenarse, como se muestra en el listado 16.5.

#### ENTRADA

#### LISTADO 16.5 Uso de `get()` con parámetros

```

1: // Listado 16.5 - Uso de get() con parámetros
2: #include <iostream.h>
3:
4: int main()
5: {
6:     char a, b, c;
7:
8:     cout << "Escriba tres letras: ";
9:     cin.get(a).get(b).get(c);
10:    cout << "a: " << a << "\nb: " << b << "\nc: " << c << endl;
11:    return 0;
12: }
```

#### SALIDA

```

Escriba tres letras: uno
a: u
b: n
c: o
```

#### ANÁLISIS

En la línea 6 se crean tres variables de tipo carácter. En la línea 9, `cin.get()` se llama tres veces, concatenado. Primero se llama a `cin.get(a)`. Esto coloca la primera letra en `a` y regresa a `cin` para que al terminar se llame a `cin.get(b)`, lo que coloca la siguiente letra en `b`. El resultado final de esto es que se llama a `cin.get(c)` y la tercera letra se coloca en `c`.

Como `cin.get(a)` se evalúa en `cin`, usted hubiera podido escribir esto:

```
cin.get(a) >> b;
```

En esta forma, `cin.get(a)` se evalúa en `cin`, por lo que la segunda frase es `cin >> b;`.

| DEBE                                                                                                                                  | NO DEBE                                                                 |
|---------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------|
| <b>DEBE</b> utilizar el operador de extracción (>>) cuando necesite saltarse el espacio en blanco.                                    | Utilizar el operador de extracción (>>) para saltar espacios en blanco. |
| <b>DEBE</b> utilizar get() con un parámetro de tipo carácter cuando necesite examinar cada carácter, incluyendo el espacio en blanco. | Utilizar el operador de extracción (>>) para saltar espacios en blanco. |

## Entrada de cadenas desde el dispositivo de entrada estándar

El operador de extracción (>>) se puede utilizar para llenar un arreglo de caracteres, al igual que las funciones miembro `get()` y `getline()`.

La forma final de `get()` toma tres parámetros. El primer parámetro es un apuntador a un arreglo de caracteres, el segundo parámetro es el número máximo de caracteres a leer más uno, y el tercer parámetro es el carácter de terminación. Éste es un buen ejemplo de sobrecarga de funciones; dependiendo de los argumentos que se pasen en la llamada de la función, se elige la forma adecuada de la función.

Si escribe 20 como el segundo parámetro, `get()` leerá 19 y luego terminará la cadena con un carácter nulo, y guardará la cadena en el primer parámetro. El tercer parámetro, el carácter de terminación, tiene como carácter predeterminado el carácter de nueva línea ('\n'). Si se llega a un carácter de terminación antes de leer al número máximo de caracteres, se escribe un carácter nulo y el carácter de terminación se deja en el búfer.

El listado 16.6 muestra el uso de esta forma de `get()`.

### ENTRADA

### LISTADO 16.6 Uso de `get()` con un arreglo de caracteres

```

1: // Listado 16.6 - Uso de get() con un arreglo de caracteres
2: #include <iostream.h>
3:
4: int main()
5: {
6:     char cadenaUno[ 256 ];
7:     char cadenaDos[ 256 ];
8:
9:     cout << "Escriba la cadena uno: ";
10:    cin.get(cadenaUno, 256);
11:    cout << "cadenaUno: " << cadenaUno << endl;
12:    cout << "Escriba la cadena dos: ";
13:    cin >> cadenaDos;
14:    cout << "CadenaDos: " << cadenaDos << endl;
15:    return 0;
16: }
```

**SALIDA**

```
Escriba la cadena uno: Ahora es tiempo
cadenaUno: Ahora es tiempo
Escriba la cadena dos: Para la bondad
CadenaDos: Para
```

**ANÁLISIS**

En las líneas 6 y 7 se crean dos arreglos de caracteres. En la línea 9 se pide al usuario que escriba una cadena, y en la línea 10 se llama a `cin.get()`. El primer parámetro es el búfer a llenar, y el segundo es uno más que el número máximo que `get()` puede aceptar (la posición adicional se otorga al carácter nulo, `'\0'`). El tercer parámetro predeterminado es el carácter de nueva línea.

El usuario escribe "Ahora es tiempo". Como el usuario termina la frase con un carácter de nueva línea, esa frase se coloca en `cadenaUno`, seguida de un terminador nulo.

En la línea 12 se pide al usuario otra cadena, y esta vez se utiliza el operador de extracción. Como el operador de extracción toma todo hasta el primer espacio en blanco, la cadena Para, junto con un carácter nulo, se guarda en la segunda cadena que, desde luego, no es lo que se esperaba.

Otra forma de solucionar este problema es utilizar `getline()`, como se muestra en el listado 16.7.

16

**ENTRADA****LISTADO 16.7 Uso de getline()**

```
1: // Listado 16.7 - Uso de getline()
2: #include <iostream.h>
3:
4: int main()
5: {
6:     char cadenaUno[ 256 ];
7:     char cadenaDos[ 256 ];
8:     char cadenaTres[ 256 ];
9:
10:    cout << "Escriba la cadena uno: ";
11:    cin.getline(cadenaUno, 256);
12:    cout << "cadenaUno: " << cadenaUno << endl;
13:    cout << "Escriba la cadena dos: ";
14:    cin >> cadenaDos;
15:    cout << "cadenaDos: " << cadenaDos << endl;
16:    cout << "Escriba la cadena tres: ";
17:    cin.getline(cadenaTres, 256);
18:    cout << "cadenaTres: " << cadenaTres << endl;
19:    return 0;
20: }
```

**SALIDA**

```
Escriba la cadena uno: uno dos tres  
cadenaUno: uno dos tres  
Escriba la cadena dos: cuatro cinco seis  
cadenaDos: cuatro  
Escriba la cadena tres: cadenaTres: cinco seis
```

**ANÁLISIS**

Este ejemplo demanda un análisis cuidadoso; contiene algunas sorpresas potenciales. En las líneas 6 a 8 se declaran tres arreglos de caracteres.

En la línea 10 se pide al usuario que escriba una cadena, la cual es leída por `getline()`. Al igual que `get()`, `getline()` toma un búfer y un número máximo de caracteres. Sin embargo, a diferencia de `get()`, `getline()` lee el carácter de nueva línea y lo descarta. Con `get()` no se descarta el carácter de nueva línea. Se deja en el búfer de entrada.

En la línea 13 se vuelve a pedir al usuario que escriba, y esta vez se utiliza el operador de extracción. El usuario escribe `cuatro cinco seis` y la primera palabra, `cuatro`, se coloca en `cadenaDos`. Luego se despliega la cadena `Escriba cadena tres`, y se llama otra vez a `getline()`. Como `cinco seis` se encuentra todavía en el búfer de entrada, se lee inmediatamente hasta el carácter de nueva línea; termina `getline()` y la cadena contenida en `cadenaTres` se imprime en la línea 18.

El usuario no tiene oportunidad de escribir en `cadenaTres` porque la segunda llamada a `getline()` se llena con la cadena que queda en el búfer de entrada después de la llamada al operador de extracción en la línea 14. No hay forma de que el usuario pueda escribir la cadena siguiente ya que `cadenaDos` termina al oprimir “Entrar”, pero esta tecla permanece en el búfer de entrada y satisface inmediatamente la segunda llamada a `getline()`.

El operador de extracción (`>>`) lee hasta el primer espacio en blanco y coloca la palabra en el arreglo de caracteres.

La función miembro `get()` está sobrecargada. En la primera versión no lleva parámetros y regresa el valor del carácter que recibe. En la segunda versión toma una sola referencia a un carácter y regresa el objeto `istream` por referencia.

En la tercera y última versión, `get()` toma un arreglo de caracteres, un número de caracteres a obtener y un carácter de terminación (el cual tiene el carácter de nueva línea predeterminado). Esta versión de `get()` lee los caracteres que están en el arreglo hasta que llega a uno menos que su número máximo de caracteres, o hasta que se encuentra con el carácter de terminación, lo que ocurre primero. Si `get()` encuentra el carácter de terminación, deja ese carácter en el búfer de entrada y deja de leer caracteres.

La función miembro `getline()` también toma tres parámetros: el búfer a llenar, uno más que el número máximo de caracteres a obtener, y el carácter de terminación. `getline()` funciona de la misma manera que `get()` con estos parámetros, excepto que `getline()` descarta el carácter de terminación.

## cin.ignore() para limpieza de la entrada

Algunas veces necesita ignorar los caracteres restantes en una línea hasta que llegue ya sea al fin de la línea (EOL) o al fin del archivo (EOF). La función miembro `ignore()` sirve para este propósito. `ignore()` toma dos parámetros: el número máximo de caracteres a ignorar y el carácter de terminación. Si escribe `ignore(80, '\n')`, se descartarán máximo 80 caracteres hasta encontrar un carácter de nueva línea. Luego este carácter se descarta y termina la instrucción `ignore()`. El listado 16.8 muestra el uso de `ignore()`.

**ENTRADA****LISTADO 16.8** Uso de `ignore()`

```
1: // Listado 16.8 - Uso de ignore()
2: #include <iostream.h>
3:
4: int main()
5: {
6:     char cadenaUno[ 255 ];
7:     char cadenaDos[ 255 ];
8:
9:     cout << "Escriba la cadena uno:" ;
10:    cin.get(cadenaUno, 255);
11:    cout << "Cadena uno: " << cadenaUno << endl;
12:    cout << "Escriba la cadena dos: " ;
13:    cin.getline(cadenaDos, 255);
14:    cout << "Cadena dos: " << cadenaDos << endl;
15:    cout << "\n\nAhora intente de nuevo...\n";
16:    cout << "Escriba la cadena uno: " ;
17:    cin.get(cadenaUno, 255);
18:    cout << "Cadena uno: " << cadenaUno << endl;
19:    cin.ignore(255, '\n');
20:    cout << "Escriba la cadena dos: " ;
21:    cin.getline(cadenaDos, 255);
22:    cout << "Cadena dos: " << cadenaDos << endl;
23:    return 0;
24: }
```

**SALIDA**

```
Escriba la cadena uno: Habia una vez
Cadena uno: Habia una vez
Escriba la cadena dos: Cadena dos:

Ahora intente de nuevo...
Escriba la cadena uno: Habia una vez
Cadena uno: Habia una vez
Escriba la cadena dos: una princesa
Cadena dos: una princesa
```

En las líneas 6 y 7 se crean dos arreglos de caracteres. En la línea 9 se pide al usuario que escriba algo, y escribe “Había una vez” y oprime “Entrar”. En la línea 10 se utiliza `get()` para leer esta cadena. `get()` llena `cadenaUno` y termina con el carácter de nueva línea, pero deja este carácter en el búfer de entrada.

En la línea 12 otra vez se pide al usuario que escriba algo, pero la función `getline()` de la línea 13 lee el carácter de nueva línea que ya estaba en el búfer y termina inmediatamente, antes de que el usuario pueda escribir algo.

En la línea 16 se vuelve a pedir al usuario que escriba algo, y éste escribe la misma línea de entrada que la primera vez. Sin embargo, en la línea 19, esta vez se utiliza `ignore()` para “comerse” el carácter de nueva línea. Por lo tanto, cuando se llega a la llamada a `getline()` en la línea 21, el búfer está vacío y el usuario puede escribir la siguiente línea del cuento.

### **peek() y putback()**

El objeto de entrada `cin` tiene dos métodos adicionales que pueden ser bastante útiles: `peek()`, que se fija en el siguiente carácter pero no lo extrae, y `putback()`, que inserta un carácter en el flujo de entrada. El listado 16.9 muestra cómo se podrían utilizar estos dos métodos.

**ENTRADA**
**LISTADO 16.9 Uso de peek() y de putback()**

```

1: // Listado 16.9 - Uso de peek() y de putback()
2: #include <iostream.h>
3:
4: int main()
5: {
6:     char ch;
7:     cout << "Escriba una frase: ";
8:     while (cin.get(ch))
9:     {
10:         if (ch == '!')
11:             cin.putback('$');
12:         else
13:             cout << ch;
14:         while (cin.peek() == '#')
15:             cin.ignore(1, '#');
16:     }
17:     return 0;
18: }
```

**SALIDA**

Escriba una frase: Ahora!es#tiempo!para#la!diversión#!  
Ahora\$estimpo\$parala\$diversión\$

**ANÁLISIS**

En la línea 6 se declara una variable de tipo carácter llamada `ch` y en la línea 7 se pide al usuario que escriba una frase. El propósito de este programa es convertir cualquier signo de admiración (!) en signo de dólar (\$) y quitar cualquier signo de numeral (#).

El programa se ejecuta en un ciclo mientras esté recibiendo caracteres distintos del fin de archivo (Ctrl+D en Linux, Ctrl+Z o Ctrl+D en otros sistemas operativos). (Recuerde que `cin.get()` regresa un `\0` para el fin de archivo). Si el carácter actual es un signo de admiración, se descarta y el signo de dólar se regresa al búfer de entrada; se leerá la próxima vez. Si el elemento actual no es signo de admiración, se imprime. Entonces se “observa” el siguiente carácter, y si resulta ser un signo de numeral, se quita.

Este ejemplo no es la manera más eficiente de hacer estas cosas (y no encontrará un signo de numeral si es el primer carácter), pero ayuda a mostrar la forma en que estos métodos trabajan. Son relativamente complicados, así que no se preocupe demasiado pensando cuándo podría utilizarlos realmente. Déjelos en su bolsa de trucos; alguna vez le serán útiles.

16



`peek()` y `putback()` se utilizan comúnmente para analizar sintácticamente cadenas y otros datos, como cuando se escribe un compilador.

## Salida con `cout`

Ya ha utilizado a `cout` junto con el operador de inserción (`<<`) sobrecargado para escribir cadenas, enteros y otros datos numéricos en la pantalla. También es posible dar formato a los datos, alinear columnas y escribir los datos numéricos en forma decimal y hexadecimal. Esta sección le muestra cómo hacerlo.

### Limpieza de la salida

Anteriormente vio que si se utiliza `endl` se limpiará el búfer de salida. `endl` llama a la función miembro `flush()` de `cout`, la cual escribe todos los datos que está guardando en el búfer. Puede llamar al método `flush()` directamente o escribir lo siguiente:

```
cout << flush
```

Esto puede ser conveniente cuando necesite cerciorarse de que se vacíe el búfer de salida y que su contenido se escriba en la pantalla.

## Funciones relacionadas

Así como el operador de extracción se puede suplir con `get()` y `getline()`, el operador de inserción se puede suplir con `put()` y `write()`.

La función `put()` se utiliza para escribir un solo carácter en el dispositivo de salida. Como `put()` regresa una referencia a `ostream` y como `cout` es un objeto `ostream`, puede concatenar `put()` de la misma manera que como lo hace con el operador de inserción. El listado 16.10 ilustra esta idea.

**ENTRADA****LISTADO 16.10 Uso de put()**

```

1: // Listado 16.10 - Uso de put()
2: #include <iostream.h>
3:
4: int main()
5: {
6:     cout.put('H').put('o').put('l').put('a').put('\n');
7:     return 0;
8: }
```

**SALIDA**

Hola

**Nota**

Algunos compiladores (distintos de los de GNU) tienen problemas al imprimir utilizando este código. Si su compilador no quiere imprimir la palabra Hola, tal vez quiera saltarse este listado.

**ANÁLISIS**

La línea 6 se evalúa de esta forma: `cout.put('H')` escribe la letra H en la pantalla y regresa el objeto `cout`. Esto hace que quede lo siguiente:

```
cout.put('o').put('l').put('a').put('\n');
```

Se escribe la letra o, dejando a `cout.put('l')`. Este proceso se repite, se escribe cada letra y se escribe el objeto `cout` que se regresa hasta el carácter final ('`\n`'), y la función termina.

La función `write()` funciona igual que el operador de inserción (`<<`), excepto que toma un parámetro que indica a la función el número máximo de caracteres a escribir. El listado 16.11 muestra su uso.

**ENTRADA****LISTADO 16.11 Uso de write()**

```

1: // Listado 16.11 - Uso de write()
2: #include <iostream.h>
3: #include <string.h>
4:
5: int main()
6: {
7:     char Uno[] = "Uno, si por tierra";
8:     int longitudCompleta = strlen(Uno);
9:     int muyCorta = longitudCompleta - 6;
10:    int muyLarga = longitudCompleta + 6;
11:
12:    cout.write(Uno, longitudCompleta) << "\n";
13:    cout.write(Uno, muyCorta) << "\n";
```

```
14:         cout.write(Uno, muyLarga) << "\n";
15:     return 0;
16: }
```

SALIDA

Uno, si por tierra  
Uno, si por  
Uno, si por tierra

## **Nota**

La última linea de salida puede lucir diferente en su computadora; en realidad sólo es basura.

16

ANÁLISIS

En la línea 7 se crea la frase Uno. En la línea 8, la longitud de la frase se asigna al entero longitudCompleta, a muyCorta se le asigna esa longitud menos seis, y a larga se le asigna longitudCompleta más seis.

En la línea 12 se imprime la frase completa usando `write()`. La longitud se establece en la longitud actual de la frase, y se imprime la frase correcta.

En la línea 13 se vuelve a imprimir la frase, pero es seis caracteres más corta que la frase completa, y esto se refleja en la salida.

En la línea 14 se imprime de nuevo la frase, pero esta vez se indica a `write()` que escriba seis caracteres adicionales. Después de escribir la frase, se escriben los siguientes seis bytes de memoria contigua.

# **Manipuladores, indicadores e instrucciones para dar formato**

El flujo de salida mantiene varios indicadores de estado que determinan cuál base (decimal o hexadecimal) utilizar, el ancho de los campos, y qué carácter utilizar para llenar los campos. Un indicador de estado es un byte a cuyos bits individuales se les asigna un significado especial. Hablaremos sobre esta forma de manipular bits en el día 21, “Qué sigue”. Cada uno de los indicadores de `ostream` se puede establecer por medio de funciones miembro y manipuladores.

## Uso de cout.width()

El ancho predeterminado de la salida será sólo el espacio suficiente para imprimir el número, carácter o cadena en el búfer de salida. Puede cambiar este ancho utilizando `width()`. Como `width()` es una función miembro o método, se debe invocar con un objeto `cout`. Esta función sólo cambia el ancho del siguiente campo de salida y luego regresa inmediatamente al valor predeterminado. El listado 16.12 muestra su uso.

**ENTRADA****LISTADO 16.12** Ajuste del ancho de la salida

```

1: // Listado 16.12 - Ajuste del ancho de la salida
2: #include <iostream.h>
3:
4: int main()
5: {
6:     cout << "Inicio >";
7:     cout.width(25);
8:     cout << 123 << "< Fin\n";
9:
10:    cout << "Inicio >";
11:    cout.width(25);
12:    cout << 123 << "< Siguiente >";
13:    cout << 456 << "< Fin\n";
14:
15:    cout << "Inicio >";
16:    cout.width(4);
17:    cout << 123456 << "< Fin\n";
18:    return 0;
19: }
```

**SALIDA**

|                     |                          |
|---------------------|--------------------------|
| Inicio >            | 123< Fin                 |
| Inicio >            | 123< Siguiente >456< Fin |
| Inicio >123456< Fin |                          |

**ANÁLISIS**

La primera salida, que se encuentra en las líneas 6 a 8, imprime el número 123 dentro de un campo cuyo ancho se establece en 25 en la línea 7. Esto se refleja en la primera línea de la salida.

La segunda línea de la salida imprime primero el valor 123 en el mismo campo cuyo ancho se establece en 25, y luego imprime el valor 456. Observe que 456 se imprime en un campo cuyo ancho se restablece a sólo el suficiente; como se dijo, el efecto de `width()` dura sólo hasta la siguiente salida.

La salida final refleja que establecer un ancho menor que la salida es lo mismo que establecer un ancho que es lo suficientemente grande para que se imprima el valor completo.

## Cómo establecer los caracteres de llenado

Por medio de `width()`, `cout` llena con espacios el campo vacío que se crea, como se muestra en el listado 16.12. Algunas veces puede ser necesario que llene el área con otros caracteres, como por ejemplo asteriscos. Para hacer esto, llame a `fill()` y páselle como parámetro el carácter que quiere utilizar como carácter de llenado. El listado 16.13 muestra esto.

**ENTRADA** **LISTADO 16.13** Uso de fill()

```
1: // Listado 16.13 - Uso de fill()
2: #include <iostream.h>
3:
4: int main()
5: {
6:     cout << "Inicio >";
7:     cout.width(25);
8:     cout << 123 << "< Fin\n";
9:
10:    cout << "Inicio >";
11:    cout.width(25);
12:    cout.fill('*');
13:    cout << 123 << "< Fin\n";
14:    return 0;
15: }
```

**16****SALIDA** Inicio > 123< Fin  
Inicio >\*\*\*\*\*123< Fin**ANÁLISIS** Las líneas 6 a 8 repiten la funcionalidad del ejemplo anterior. Las líneas 10 a 14 repiten esto de nuevo, pero esta vez, en la línea 12 se establece un asterisco como carácter de llenado, como se refleja en la salida.

## Cómo establecer indicadores de iostream

Los objetos `iostream` llevan el registro de su estado por medio de indicadores. Puede establecer estos indicadores llamando a `setf()` y pasando cualquiera de las constantes enumeradas predefinidas.

Se dice que los objetos tienen estado cuando uno o todos sus datos representan una condición que puede cambiar durante el transcurso del programa.

Por ejemplo, puede establecer si se van o no a mostrar ceros a la derecha (para que 20.00 no se trunque a 20). Para activar los ceros a la derecha, llame a `setf(ios::showpoint)`.

Las constantes enumeradas tienen alcance fuera de la clase `iostream` (`ios`) y por consecuencia se llaman con su identificación completa del tipo `ios::nombreindicador`, como `ios::showpoint`.

Puede hacer que el signo de más (+) aparezca antes de los números positivos utilizando `ios::showpos`. Puede cambiar la alineación de la salida usando `ios::left`, `ios::right` o `ios::internal`.

Por último, puede establecer la base de los números a desplegar usando `ios::dec` (decimal), `ios::oct` (octal-base ocho), o `ios::hex` (hexadecimal-base diecisésis). Estos indicadores también se pueden concatenar en el operador de inserción. El listado 16.14 muestra estas

configuraciones. A manera de bono, el listado 16.14 también presenta al manipulador `setw`, el cual establece el ancho, pero también se puede concatenar con el operador de inserción.

**ENTRADA** **LISTADO 16.14** Uso de `setf`

```
1: // Listing 16.14 - Uso de setf()
2: #include <iostream.h>
3: #include <iomanip.h>
4:
5: int main()
6: {
7:     const int numero = 185;
8:
9:     cout << "El número es " << numero << endl;
10:
11:    cout << "El número es " << hex << numero << endl;
12:
13:    cout.setf(ios::showbase);
14:    cout << "El número es " << hex << numero << endl;
15:
16:    cout << "El número es " ;
17:    cout.width(10);
18:    cout << hex << numero << endl;
19:
20:    cout << "El número es " ;
21:    cout.width(10);
22:    cout.setf(ios::left);
23:    cout << hex << numero << endl;
24:
25:    cout << "El número es " ;
26:    cout.width(10);
27:    cout.setf(ios::internal);
28:    cout << hex << numero << endl;
29:
30:    cout << "El número es:" << setw(10) << hex << numero << endl;
31:    return 0;
32: }
```

**SALIDA**

```
El número es 185
El número es b9
El número es 0xb9
El número es      0xb9
El número es 0xb9
El número es      0xb9
El número es:    0xb9
```

**ANÁLISIS**

En la línea 7, la constante entera llamada `numero` se inicializa con el valor 185. Esto se despliega en la línea 9.

El valor se despliega de nuevo en la línea 11, pero esta vez se concatena el manipulador hex, lo que ocasiona que el valor se despliegue en forma hexadecimal como b9. (El valor b en forma hexadecimal representa al número 11. Once por 16 es igual a 176; sume el 9 para obtener un total de 185.)

En la línea 13 se establece el indicador showbase. Esto ocasiona que se agregue el prefijo 0x a todos los números hexadecimales, como se refleja en la salida.

En la línea 17 se establece el ancho en 10, y el valor se desplaza hacia el lado derecho. En la línea 21 el ancho se vuelve a establecer en 10, pero esta vez la alineación se establece a la izquierda, y el número se imprime alineado hacia la izquierda.

En la línea 26 se vuelve a establecer el ancho en 10, pero esta vez la alineación es interna. Por lo tanto, el 0x se imprime alineado hacia la izquierda, pero el valor b9 se imprime alineado hacia la derecha.

Por último, en la línea 30 se utiliza el operador de concatenación setw() para establecer el ancho en 10, y se imprime de nuevo el valor.

## Flujos en comparación con la función printf()

La mayoría de las implementaciones de C++ también proporcionan las bibliotecas de E/S estándar de C, incluyendo la función printf(). Aunque printf() es, de cierta forma, más fácil de utilizar que cout, es mucho menos deseable.

printf() no proporciona seguridad en los tipos, por lo que es fácil indicarle sin darse cuenta que despliegue un entero como si fuera un carácter, y viceversa. Además, printf() no soporta las clases, por lo que no es posible enseñarle cómo imprimir los datos de su clase; debe pasar los miembros de la clase a printf() uno por uno.

Por otro lado, printf() facilita el formateo ya que los caracteres de formato se pueden colocar directamente en la instrucción printf(). Debido a que printf() tiene sus usos y debido a que muchos programadores aún la utilizan bastante, en esta sección se repasará brevemente su uso.

Para utilizar printf(), asegúrese de incluir el archivo de encabezado stdio.h. En su forma más simple, printf() toma una cadena de formato como su primer parámetro y luego una serie de valores como sus parámetros restantes.

La cadena de formato es una cadena encerrada entre comillas que contiene texto y especificadores de conversión. Todos los especificadores de conversión deben empezar con el símbolo de porcentaje (%). Los especificadores de conversión comunes se presentan en la tabla 16.1.

**TABLA 16.1** Los especificadores de conversión comunes

| Especificador | Utilizado para |
|---------------|----------------|
| %s            | Cadenas        |
| %d            | Enteros        |
| %ld           | Entero largo   |
| %lf           | Doble          |
| %f            | Flotante       |

Cada uno de los especificadores de conversión también puede proporcionar una instrucción para el ancho y una instrucción para la precisión, expresado como tipo float, en el que los dígitos a la izquierda del decimal se utilizan para el ancho total, y los dígitos a la derecha del decimal proporcionan la precisión para los números de punto flotante. Por ejemplo, %5d es el especificador para un entero de 5 dígitos de ancho, y %15.5f es el especificador para un valor de tipo float de 15 dígitos de ancho, de los cuales se dedican los cinco dígitos finales para los decimales. El listado 16.15 muestra varios usos de printf().

**ENTRADA****LISTADO 16.15** Impresión por medio de printf()

```

1: // Listado 16.15 - Uso de printf()
2: #include <stdio.h>
3:
4: int main()
5: {
6:     printf("%s", "¡Hola, mundo!\n");
7:
8:     char * frase = "¡Hola de nuevo!\n";
9:     printf("%s", frase);
10:
11:    int x = 5;
12:    printf("%d\n",x);
13:
14:    char * fraseDos = "He aquí algunos valores: ";
15:    char * fraseTres = " y aquí están otros: ";
16:    int y = 7, z = 35;
17:    long longVar = 98456;
18:    float floatVar = 8.8f;
19:    printf("%s %d %d %s %ld %f\n",
20:           fraseDos, y, z, fraseTres, longVar, floatVar);
21:
22:    char * fraseCuatro = "Con formato: ";
23:    printf("%s %5d %10d %10.5f\n", fraseCuatro, y, z, floatVar);
24:    return 0;
25: }
```

**SALIDA**

¡Hola, mundo!  
 ¡Hola de nuevo!  
 5

```
He aqui algunos valores: 7 35 y aqui están otros: 98456 8.800000
Con formato:    7      35     8.80000
```

**ANÁLISIS**

La primera instrucción `printf()`, que se encuentra en la línea 6, utiliza la forma estándar: el término `printf`, seguido de una cadena encerrada entre comillas con un especificador de conversión (en este caso `%s`), seguido de un valor a insertar en el especificador de conversión.

El especificador `%s` indica que es una cadena, y el valor de la cadena es, en este caso, la cadena encerrada entre comillas “Hola, mundo!\n”

La segunda instrucción `printf()` es igual que la primera, pero esta vez se utiliza un apuntador a un tipo de datos `char`, en lugar de colocar la cadena dentro de la instrucción `printf()`.

La tercera instrucción `printf()`, que se encuentra en la línea 12, utiliza el especificador de conversión para enteros, y como valor la variable de tipo entero `x`. La cuarta instrucción `printf()`, que se encuentra en la línea 19, es más compleja. Aquí se concatenan 6 valores. Se proporciona cada uno de los especificadores de conversión, y luego se proporcionan los valores, separados con comas.

Por último, en la línea 23 se utilizan especificaciones de formato para especificar el ancho y la precisión. Como puede ver, todo esto es un poco más sencillo que el uso de manipuladores.

Sin embargo, como se dijo anteriormente, la limitación aquí es que no hay comprobación de tipos y `printf()` no se puede declarar como función amiga o método de una clase. Así que, si quiere imprimir todos los datos miembro de una clase, debe pasar de manera explícita cada método de acceso a la instrucción `printf()`.

**16****Preguntas frecuentes**

**FAQ:** ¿Puede resumir la forma en que se manipula la salida?

**Respuesta (con agradecimiento especial a Robert Francis):** En C++, para dar formato a la salida se utiliza una combinación de caracteres especiales, manipuladores de salida e indicadores.

Los siguientes caracteres especiales se incluyen en una cadena de salida que se envía por medio del operador de inserción:

- \\n—Carácter de nueva línea
- \\r—Retorno de carro
- \\t—Tabulador
- \\—Barra diagonal inversa
- \\ddd (número octal)—carácter ASCII
- \\a—Alarma (sonar campana)

**Ejemplo:**

```
cout << "\\Ocurrió un error\\t"
```

Timbra la campana, imprime un mensaje de error y avanza al siguiente tabulador. Los manipuladores se utilizan con el operador cout. Los manipuladores que toman argumentos requieren que se incluya iomanip.h en el archivo del programa.

La siguiente es una lista de manipuladores que no requieren de iomanip.h:

- flush—Limpia el búfer de salida
- endl—Inserta un carácter de nueva línea y limpia el búfer de salida
- oct—Establece la base de la salida en octal
- dec—Establece la base de la salida en decimal
- hex—Establece la base de la salida en hexadecimal

La siguiente es una lista de manipuladores que sí requieren de iomanip.h:

- setbase(base)—Establece la base de la salida (0 = decimal, 8 = octal, 10 = decimal, 16 = hexadecimal)
- setw (ancho)—Establece el ancho mínimo del campo
- setfill (ch)—Llena con el carácter especificado por ch cuando se define el ancho
- setprecision (p)—Establece la precisión para números de punto flotante
- setiosflags (f)—Establece uno o más indicadores de ios
- resetiosflags (f)—Restablece uno o más indicadores de ios

Ejemplo:

```
cout << setw(12) << setfill('#') << hex << x << endl;
```

Establece el ancho del campo en 12, establece el carácter de llenado en '#', especifica una salida hexadecimal, imprime el valor de 'x', coloca un carácter de nueva línea en el búfer y limpia el búfer. Todos los manipuladores, excepto flush, endl, y setw, permanecen vigentes hasta que se cambien o hasta que termina el programa. setw regresa a su valor predeterminado después del cout actual.

Los siguientes indicadores de ios se pueden utilizar con los manipuladores setiosflags y resetiosflags:

- ios::left—Justifica la salida a la izquierda en el ancho especificado
- ios::right—Justifica la salida a la derecha en el ancho especificado
- ios::internal—El signo se justifica a la izquierda y el valor se justifica a la derecha
- ios::dec—Salida decimal
- ios::oct—Salida octal
- ios::hex—Salida hexadecimal
- ios::showbase—Agrega un 0x a los números hexadecimales y un 0 a los números octales
- ios::showpoint—Agrega ceros a la derecha como lo requiera la precisión
- ios::uppercase—Los números hexadecimales y en notación científica se muestran en mayúsculas
- ios::showpos—Mostrar el signo + para números positivos
- ios::scientific—Mostrar números de punto flotante en notación científica
- ios::fixed—Mostrar números de punto flotante en notación decimal

Puede obtener información adicional en el archivo ios.h y en la documentación de la biblioteca GNU.

## Entrada y salida de archivos

Los flujos proporcionan una manera uniforme de manejar los datos que provienen del teclado o del disco duro y los datos que van a la pantalla o al disco duro. En cualquier caso, puede utilizar los operadores de inserción y de extracción o las demás funciones y manipuladores asociados. Para abrir y cerrar archivos se crean objetos `ifstream` y `ofstream` (para entrada y salida, respectivamente) como se describe en las siguientes secciones. Debido a que los objetos `ifstream` son similares a los objetos `ofstream`, el material cubierto es limitado. Sólo aplique las técnicas para `ofstream`.

16

### Uso de `ofstream`

Los objetos específicos utilizados para leer desde los archivos o escribir en ellos se llaman objetos `ofstream`. Éstos se derivan de los objetos `iostream` que ya ha utilizado.

Para empezar a escribir en un archivo, primero debe crear un objeto `ofstream` y luego asociar ese objeto con un archivo específico del disco. Para utilizar objetos `ofstream`, debe asegurarse de incluir `fstream.h` en su programa.

#### Nota

Debido a que `fstream.h` incluye a `iostream.h`, no necesita incluir `iostream` de forma explícita.

### Estados de condición

Los objetos `iostream` mantienen indicadores que informan sobre el estado de la salida y de la entrada. Puede revisar cada uno de estos indicadores por medio de las funciones booleanas `eof()`, `bad()`, `fail()` y `good()`. La función `eof()` regresa `true` si el objeto `iostream` se ha encontrado con EOF, fin del archivo. La función `bad()` regresa `true` si intenta realizar una operación no válida. La función `fail()` regresa `true` siempre que `bad()` sea `true` o que falle una operación. Por último, la función `good()` regresa `true` siempre que las otras tres funciones sean `false`.

### Apertura de archivos para entrada y salida

Para abrir el archivo `miarchivo.cpp` con un objeto `ofstream`, se declara una instancia de un objeto `ofstream` y se pasa el nombre del archivo como parámetro:

```
ofstream fout("miarchivo.cpp");
```

Para abrir este archivo como entrada se hace lo mismo, sólo que se utiliza un objeto:

```
ifstream fin("miarchivo.cpp");
```

Tenga en cuenta que `fout` y `fin` son nombres que usted puede asignar; aquí se ha utilizado `fout` para reflejar su similitud con `cout`, y se ha utilizado `fin` para reflejar su similitud con `cin`.

Una función importante para flujos de archivos que necesitará de inmediato es `close()`. Cada objeto de flujo de archivos que cree abrirá un archivo ya sea para lectura o para escritura (o ambas). Es importante utilizar `close()` en el archivo después de terminar de leer o escribir; esto asegura que el archivo no se dañe y que los datos que escribió se copien en el disco.

Después de que los objetos stream se asocian con archivos, se pueden utilizar igual que cualquier otro objeto stream. El listado 16.16 muestra esto.

**ENTRADA****LISTADO 16.16** Apertura de archivos para lectura y escritura

```

1: // Listado 16.16 - Lectura y escritura de archivos
2: #include <iostream.h>
3:
4: int main()
5: {
6:     char nombreArchivo[ 80 ];
7:     char bufer[ 255 ]; // para entrada del usuario
8:
9:     cout << "Nombre de archivo: ";
10:    cin >> nombreArchivo;
11:    ofstream fout(nombreArchivo); // abrir para escritura
12:    fout << "Esta linea se escribe directamente en el archivo...\n";
13:    cout << "Escriba el texto para el archivo: ";
14:    cin.ignore(1, '\n'); // elimina la nueva linea despues del nombre
   del archivo
15:    cin.getline(bufer, 255); // obtener la entrada del usuario
16:    fout << bufer << "\n"; // y escribirla en el archivo
17:    fout.close(); // cerrar el archivo, listo para volver a abrir
18:
19:    ifstream fin(nombreArchivo); // volver a abrir para lectura
20:    cout << "He aquí el contenido del archivo:\n";
21:    char ch;
22:    while (fin.get(ch))
23:        cout << ch;
24:
25:    cout << "\n***Fin del contenido del archivo.***\n";
26:
27:    fin.close(); // siempre reditúa ser ordenado
28:    return 0;
29: }
```

**SALIDA**

```

Nombre de archivo: prueba1
Escriba el texto para el archivo: ¡Este texto se escribirá en el archivo!
He aquí el contenido del archivo:
Esta linea se escribe directamente en el archivo...
¡Este texto se escribirá en el archivo!

***Fin del contenido del archivo.***
```

**ANÁLISIS**

En la línea 6 se reserva un búfer para el nombre del archivo, y en la línea 7 se reserva otro búfer para la entrada del usuario. En la línea 9 se pide al usuario que escriba

un nombre de archivo, y esta respuesta se escribe en el búfer llamado `nombreArchivo`. En la línea 11 se crea un objeto `ofstream` llamado `fout`, el cual se asocia con el nuevo nombre de archivo. Esto abre el archivo; si el archivo ya existe, su contenido se elimina.

En la línea 12 se escribe una cadena de texto directamente en el archivo. En la línea 13 se pide al usuario que escriba algo. El carácter de nueva línea que quedó al escribir el nombre del archivo se elimina en la línea 14, y la entrada del usuario se guarda en `bufer` en la línea 15. Esta entrada se escribe en el archivo, junto con un carácter de nueva línea en la línea 16, y luego el archivo se cierra en la línea 17.

En la línea 19 se vuelve a abrir el archivo, esta vez en modo de lectura, y se lee su contenido, un carácter a la vez, en las líneas 22 y 23.

## 16

### Cómo cambiar el comportamiento predeterminado de `ofstream` al abrir un archivo

El comportamiento predeterminado al momento de abrir un archivo es crear el archivo si todavía no existe, y truncarlo (es decir, eliminar todo su contenido) si ya existe. Si no quiere este comportamiento predeterminado, puede proporcionar explícitamente un segundo argumento al constructor de su objeto `ofstream`.

Entre los argumentos válidos se incluyen:

- `ios::app`—Agrega al final de los archivos existentes en lugar de eliminar el contenido.
- `ios::ate`—Lo lleva al final del archivo, pero puede escribir datos en cualquier lugar del archivo.
- `ios::trunc`—El predeterminado. Elimina los archivos existentes.
- `ios::nocreate`—Si el archivo no existe, la apertura falla.
- `ios::noreplace`—Si el archivo ya existe, la apertura falla.

Observe que `app` es abreviatura de *append* (agregar); `ate` es abreviatura de *at end* (al final), y `trunc` es abreviatura de *truncate* (truncar). El listado 16.17 muestra el uso de `app` abriendo de nuevo el archivo del listado 16.16 y agregándole contenido.

#### ENTRADA

#### LISTADO 16.17 Agrega contenido al final de un archivo

```
1: //Listado 16.17 - Concatena texto al final de un archivo
2: #include <iostream.h>
3:
4: int main() // regresa 1 en caso de error
5: {
6:     char nombreArchivo[ 80 ];
7:     char bufer[ 255 ];
8:
9:     cout << "Vuelva a escribir el nombre del archivo: ";
10:    cin >> nombreArchivo;
11:    ifstream fin(nombreArchivo);
12:    if (fin) // ¿ya existe?
```

continúa

**LISTADO 16.17** CONTINUACIÓN

```

13:      {
14:          cout << "Contenido actual del archivo:\n";
15:          char ch;
16:          while (fin.get(ch))
17:              cout << ch;
18:          cout << "\n***Fin del contenido del archivo.\n";
19:      }
20:      fin.close();
21:      cout << "\nAbriendo " << nombreArchivo;
22:      cout << " en modo agregar... \n";
23:      ofstream fout(nombreArchivo, ios::app);
24:      if (!fout)
25:      {
26:          cout << "No se puede abrir" << nombreArchivo;
27:          cout << " para agregar.\n";
28:          return(1);
29:      }
30:      cout << "\nEscriba el texto para el archivo: ";
31:      cin.ignore(1, '\n');
32:      cin.getline(bufer, 255);
33:      fout << bufer << "\n";
34:      fout.close();
35:      fin.open(nombreArchivo); // ireasignar objeto fin existente!
36:      if (!fin)
37:      {
38:          cout << "No se puede abrir" << nombreArchivo;
39:          cout << " para lectura.\n";
40:          return(1);
41:      }
42:      cout << "\nHe aquí el contenido del archivo:\n";
43:      char ch;
44:      while (fin.get(ch))
45:          cout << ch;
46:      cout << "\n***Fin del contenido del archivo.\n";
47:      fin.close();
48:      return 0;
49:  }

```

**SALIDA**

Vuelva a escribir el nombre del archivo: pruebal  
 Contenido actual del archivo:  
 Esta linea se escribe directamente en el archivo...  
 ¡Este texto se escribirá en el archivo!

\*\*\*Fin del contenido del archivo.\*\*\*

Abriendo pruebal en modo agregar...

Escriba el texto para el archivo: Más texto para el archivo  
 He aquí el contenido del archivo:  
 Esta linea se escribe directamente en el archivo...

¡Este texto se escribirá en el archivo!  
Más texto para el archivo

\*\*\*Fin del contenido del archivo.\*\*\*

### ANÁLISIS

De nuevo se pide al usuario que escriba el nombre del archivo. Esta vez se crea un objeto stream de archivo de lectura en la línea 11. Esa apertura se prueba en la línea 12, y si el archivo ya existe, se imprime su contenido en las líneas 14 a 18. Observe que `if(fin)` es sinónimo de `if (fin.good())`.

Luego se cierra el archivo de entrada y se vuelve a abrir, esta vez en modo agregar, en la línea 23. Después de esta apertura (y de cada apertura), el archivo se prueba para asegurar que se haya abierto adecuadamente. Observe que `if(!fout)` es lo mismo que probar `if (fout.fail())`. Luego se pide al usuario que escriba un texto, y el archivo se cierra de nuevo en la línea 34.

Por último, como en el listado 16.16, el archivo se vuelve a abrir en modo de lectura; sin embargo, esta vez no es necesario volver a declarar `fin`. Sólo se le vuelve a asignar el mismo nombre de archivo. De nuevo se prueba la apertura en la línea 36, y si todo está bien, se imprime en la pantalla el contenido del archivo, y se cierra el archivo por última vez.

16

| DEBE                                                                                         | NO DEBE                                                                                 |
|----------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| <b>DEBE</b> probar cada apertura de un archivo para cerciorarse de que fue exitosa.          | <b>NO DEBE</b> tratar de cerrar o reasignar a <code>cin</code> ni a <code>cout</code> . |
| <b>DEBE</b> reutilizar los objetos <code>ifstream</code> y <code>ofstream</code> existentes. |                                                                                         |
| <b>DEBE</b> cerrar todos los objetos <code>fstream</code> cuando termine de utilizarlos.     |                                                                                         |

## Archivos binarios en comparación con archivos de texto

Algunos sistemas operativos, como DOS, hacen distinción entre los archivos de texto y los archivos binarios. Los archivos de texto guardan todo como texto (como tal vez haya adivinado), por lo que los números grandes, como 54.325, se guardan como una cadena de números ('5', '4', '.', '3', '2', '5'). Esto puede ser ineficiente, pero tiene la ventaja de que puede leer y manipular el texto por medio de los comandos estándar de Linux, como `cat`, `more`, `vi`, `head`, `tail`, `grep`, etc., y por medio de programas simples, como el comando `type` de DOS.

Para ayudar a que la biblioteca en tiempo de ejecución distinga entre archivos de texto y binarios, C++ proporciona el indicador `ios::binary`. Este indicador se ignora en muchos sistemas debido a que todos los datos se guardan en formato binario. En algunos sistemas algo mojigatos, el indicador `ios::binary` no es válido y no compila, o lo que es peor, se ignora silenciosamente.



En Linux, el sistema operativo ve los archivos como un flujo de bytes (es responsabilidad de los programadores dictar la estructura de ese flujo en sus programas). En Linux, el compilador GNU maneja el indicador `ios:binary` en forma apropiada. En otros sistemas operativos, tal vez no sea así.

Los archivos binarios pueden guardar no sólo enteros y cadenas, sino también estructuras completas de datos. Puede escribir todos los datos a la vez usando el método `write()` de `fstream`. Uno de los usos más comunes de un archivo binario es guardar todos los datos miembro de un objeto.

Si utiliza `write()`, puede recuperar los datos por medio de `read()`. Sin embargo, cada una de estas funciones espera un apuntador a un carácter, por lo que debe convertir la dirección de su objeto en apuntador a un carácter.

El segundo argumento para estas funciones es el número de caracteres a escribir, el cual se puede determinar mediante `sizeof()`. Observe que lo que está escribiendo son los datos, no los métodos. Lo que recupera son sólo datos. El listado 16.18 muestra cómo escribir en un archivo el contenido de una clase.

**ENTRADA****LISTADO 16.18** Escritura de una clase en un archivo

```
1: // Listado 16.18 -
2: #include <fstream.h>
3:
4: class Animal
5: {
6: public:
7:     Animal(int peso, long dias) :
8:         suPeso(peso),
9:         suNumeroDiasVivo(dias)
10:    {}
11:   ~Animal() {}
12:   int ObtenerPeso() const
13:   { return suPeso; }
14:   void AsignarPeso(int peso)
15:   { suPeso = peso; }
16:   long ObtenerDiasVivo() const
17:   { return suNumeroDiasVivo; }
18:   void AsignarDiasVivo(long dias)
19:   { suNumeroDiasVivo = dias; }
20: private:
21:     int suPeso;
22:     long suNumeroDiasVivo;
23: };
24:
```

```

25:     int main() // regresa 1 en caso de error
26:     {
27:         char nombreArchivo[ 80 ];
28:
29:         cout << "Escriba el nombre del archivo: ";
30:         cin >> nombreArchivo;
31:         ofstream fout(nombreArchivo, ios::binary);
32:         if (!fout)
33:         {
34:             cout << "No se puede abrir" << nombreArchivo;
35:             cout << " para escritura.\n";
36:             return(1);
37:         }
38:
39:         Animal Oso(50, 100);
40:         fout.write((char *) &Oso, sizeof Oso);
41:         fout.close();
42:         ifstream fin(nombreArchivo, ios::binary);
43:         if (!fin)
44:         {
45:             cout << "No se puede abrir" << nombreArchivo;
46:             cout << " para lectura.\n";
47:             return(1);
48:         }
49:
50:         Animal OsoDos(1, 1);
51:         cout << "OsoDos peso: " << OsoDos.ObtenerPeso() << endl;
52:         cout << "OsoDos dias: " << OsoDos.ObtenerDiasVivo() << endl;
53:         fin.read((char*) &OsoDos, sizeof OsoDos);
54:         cout << "OsoDos peso: " << OsoDos.ObtenerPeso() << endl;
55:         cout << "OsoDos dias: " << OsoDos.ObtenerDiasVivo() << endl;
56:         fin.close();
57:         return 0;
58:     }

```

**SALIDA**

Escriba el nombre del archivo: Animales  
 OsoDos peso: 1  
 OsoDos dias: 1  
 OsoDos peso: 50  
 OsoDos dias: 100

**ANÁLISIS**

En las líneas 4 a 23 se declara una clase *Animal* simplificada. En las líneas 31 a 37 se crea un archivo y se abre para escritura en modo binario. En la línea 39 se crea un animal cuyo peso es 50, y tiene 100 días de estar vivo, y en la línea 40 se escriben sus datos en el archivo.

El archivo se cierra en la línea 41 y se vuelve a abrir para lectura en modo binario en la línea 42. En la línea 50 se crea un segundo animal cuyo peso es 1, y tiene sólo un día de estar vivo. En la línea 53 se leen los datos del archivo en el nuevo objeto animal, borrando los datos existentes y reemplazándolos con los datos del archivo.

## Procesamiento de la línea de comandos

Muchos sistemas operativos, como DOS y UNIX, permiten que el usuario pase parámetros al programa cuando éste inicia. Estos parámetros se conocen como opciones de la línea de comandos, y por lo general se separan con espacios en la línea de comandos, por ejemplo:

```
UnPrograma Param1 Param2 Param3
```

Estos parámetros no se pasan directamente a `main()`. En vez de eso, se pasan dos parámetros a la función `main()` de cualquier programa. El primer parámetro es un entero que contiene el número de argumentos de la línea de comandos. También se cuenta el nombre del programa, así que todos los programas tienen por lo menos un parámetro. El ejemplo de la línea de comandos que se mostró anteriormente tiene cuatro. (El nombre `UnPrograma` más los tres parámetros forman un total de cuatro argumentos de línea de comandos.)

El segundo parámetro que se pasa a `main()` es un arreglo de apuntadores a cadenas de caracteres. Ya que el nombre de un arreglo es un apuntador constante al primer elemento del arreglo, puede declarar este argumento como un apuntador a un apuntador a un `char`, un apuntador a un arreglo de valores de tipo `char`, o como un arreglo de arreglos de valores de tipo `char`.

Por lo general, el primer argumento se llama `argc` (conteo de los argumentos), pero puede ponerle el nombre que usted quiera. El segundo argumento por lo general se llama `argv` (vector de argumentos), pero, como en el anterior, esto es sólo una convención.

Es común evaluar a `argc` para asegurarse de haber recibido el número esperado de argumentos, y utilizar `argv` para tener acceso a los argumentos en sí. Observe que `argv[0]` es el nombre del programa, y `argv[1]` es el primer parámetro para el programa, que se representa como una cadena. Si su programa toma dos números como argumentos, tendrá que convertir estas cadenas en números. En el día 21 verá cómo utilizar las conversiones de la biblioteca estándar. El listado 16.19 muestra cómo utilizar los argumentos de la línea de comandos.

### ENTRADA

### LISTADO 16.19 Uso de los argumentos de la línea de comandos

```

1: // Listado 16.19 - Uso de los argumentos de la linea de comandos
2: #include <iostream.h>
3:
4: int main(int argc, char **argv)
5: {
6:     cout << "Se recibieron " << argc << " argumentos...\n";
7:     for (int i = 0; i < argc; i++)
8:         cout << "argumento " << i << ":" << argv[i] << endl;
9:     return 0;
10: }
```

### SALIDA

```
lst16-19 Aprenda C++ en 21 días
Se recibieron 6 argumentos...
argumento 0: lst16-19
```

```
argumento 1: Aprenda
argumento 2: C++
argumento 3: en
argumento 4: 21
argumento 5: dias
```

**Nota**

Esto funciona mejor cuando se ejecuta desde una línea de comandos. Si está utilizando una interfaz GUI como KDE (K Desktop Environment), tendrá que abrir una sesión de terminal. Si está utilizando un sistema operativo distinto de Linux, tal vez tenga que ejecutar este código desde la línea de comandos (es decir, desde una ventana DOS) o configurar los parámetros de línea de comandos en su compilador (para ello, vea la documentación de su compilador).

**16****ANÁLISIS**

La función `main()` declara dos argumentos: `argc` es un entero que contiene la cuenta de los argumentos de la línea de comandos, y `argv` es un apuntador al arreglo de cadenas. Cada cadena del arreglo a la que apunta `argv` es un argumento de la línea de comandos. Observe que `argv` se hubiera podido declarar fácilmente como `char *argv[]` o `char argv[][]`. La forma en que declare a `argv` es cuestión de su estilo de programación; aún cuando este programa lo declaró como un apuntador a un apuntador, se utilizaron desplazamientos de arreglos para tener acceso a las cadenas individuales.

En la línea 6 se utiliza `argc` para imprimir el número de argumentos de la línea de comandos: 6 en total, contando el nombre del programa.

En las líneas 7 y 8 se imprime cada uno de los argumentos de la línea de comandos, pasando las cadenas con terminación nula a `cout` mediante la indexación del arreglo de cadenas.

En el listado 16.20 se muestra un uso más común para los argumentos de la línea de comandos. El código del listado 16.18 se modificó para tomar el nombre de archivo como argumento de la línea de comandos.

**ENTRADA****LISTADO 16.20** Uso de argumentos de la línea de comandos

```
1: // Listado 16.20 - Ejemplo del manejo de los argumentos
2: #include <iostream.h>
3:
4: class Animal
5: {
6: public:
7:     Animal(int peso, long dias):
8:         suPeso(peso),
9:         suNumeroDiasVivo(dias)
10:    {}
```

*continúa*

**LISTADO 16.20** CONTINUACIÓN

---

```
11:     ~Animal(){}
12:     int ObtenerPeso() const
13:     { return suPeso; }
14:     void AsignarPeso(int peso)
15:         { suPeso = peso; }
16:     long ObtenerDiasVivo() const
17:     { return suNumeroDiasVivo; }
18:     void AsignarDiasVivo(long dias)
19:     { suNumeroDiasVivo = dias; }
20: private:
21:     int suPeso;
22:     long suNumeroDiasVivo;
23: };
24:
25: int main(int argc, char *argv[]) // regresa 1 en caso de error
26: {
27:     if (argc != 2)
28:     {
29:         cout << "Uso: " << argv[ 0 ];
30:         cout << " <nombreachivo>" << endl;
31:         return(1);
32:     }
33:     ofstream fout(argv[ 1 ], ios::binary);
34:     if (!fout)
35:     {
36:         cout << "No se puede abrir" << argv[1];
37:         cout << " para escritura.\n";
38:         return(1);
39:     }
40:
41:     Animal Oso(50, 100);
42:     fout.write((char*) &Oso, sizeof Oso);
43:     fout.close();
44:     ifstream fin(argv[ 1 ], ios::binary);
45:     if (!fin)
46:     {
47:         cout << "No se puede abrir" << argv[ 1 ];
48:         cout << " para lectura.\n";
49:         return(1);
50:     }
51:
52:     Animal OsoDos(1, 1);
53:     cout << "OsoDos peso: " << OsoDos.ObtenerPeso() << endl;
54:     cout << "OsoDos días: " << OsoDos.ObtenerDiasVivo() << endl;
55:     fin.read((char*) &OsoDos, sizeof OsoDos);
56:     cout << "OsoDos peso: " << OsoDos.ObtenerPeso() << endl;
57:     cout << "OsoDos días: " << OsoDos.ObtenerDiasVivo() << endl;
58:     fin.close();
59:     return 0;
60: }
```

---

**SALIDA**

```
OsoDos peso: 1
OsoDos dias: 1
OsoDos peso: 50
OsoDos dias: 100
```

**ANÁLISIS**

La declaración de la clase `Animal` es la misma que la del listado 16.18. Sin embargo, esta vez, en lugar de pedir al usuario el nombre del archivo, se utilizan argumentos de la línea de comandos. En la línea 25 se declara a `main()` para tomar dos parámetros: la cuenta de los argumentos de la línea de comandos y un apuntador al arreglo de cadenas de argumentos de la línea de comandos.

En las líneas 27 a 32 el programa se asegura de recibir el número esperado de argumentos (exactamente dos). Si el usuario no proporciona un solo nombre de archivo, se imprime un mensaje de error:

```
Uso 1st16-20 <nombrearchivo>
```

Entonces el programa termina. Observe que al usar `argv[0]` en lugar de codificar directamente el nombre de un programa, puede compilar este programa para que tenga cualquier nombre, y esta instrucción de uso funcionará automáticamente.

En la línea 33, el programa intenta abrir el nombre de archivo proporcionado para salida binaria. No hay razón para copiar el nombre de archivo en un búfer local temporal. Se puede utilizar directamente al tener acceso a `argv[1]`.

Esta técnica se repite en la línea 44 en donde el mismo archivo se vuelve a abrir para lectura, y se utiliza en las instrucciones de condición de error cuando los archivos no se pueden abrir, en las líneas 36 y 47.



## Resumen

Hoy se presentaron los flujos, y se describieron los objetos globales `cout` y `cin`. El objetivo de los objetos `istream` y `ostream` es encapsular el trabajo de escribir en los controladores de dispositivos y usar búferes para la entrada y la salida.

En cualquier programa se crean cuatro objetos stream estándar: `cout`, `cin`, `cerr` y `clog`. Cada uno de éstos se puede “redireccionar” en muchos sistemas operativos.

El objeto `cin` de `istream` se puede utilizar para entrada, y su uso más común es con el operador de extracción (`>>`) sobrecargado. El objeto `cout` de `ostream` se utiliza para la salida, y su uso más común es con el operador de inserción (`<<`) sobrecargado.

Cada uno de estos objetos tienen una variedad de métodos, o funciones miembro, como `get()` y `put()`. Debido a que las formas comunes de cada uno de estos métodos regresan una referencia a un objeto `stream`, es fácil concatenar cada uno de estos operadores y funciones.

El estado de los objetos `stream` se puede cambiar mediante el uso de manipuladores. Éstos pueden establecer las características de formato y despliegue y varios atributos más de los objetos `stream`.

La E/S de archivos se puede lograr mediante el uso de clases `fstream`, las cuales se derivan de las clases `stream`. Además de soportar los operadores normales de inserción y de extracción, estos objetos también soportan `read()` y `write()` para guardar y recuperar objetos binarios grandes.

## Preguntas y respuestas

- P ¿Cómo se sabe cuándo utilizar los operadores de inserción y de extracción, y cuándo utilizar los otros métodos de las clases stream?**
- R** En general, es más fácil utilizar los operadores de inserción y de extracción, y se prefieren cuando su comportamiento es lo que se necesita. En aquellas circunstancias inusuales en las que estos operadores no puedan hacer el trabajo (como leer en una cadena de palabras), se pueden utilizar los otros métodos.
- P ¿Cuál es la diferencia entre `cerr` y `clog`?**
- R** `cerr` no utiliza búfer. Todo lo que se escribe en `cerr` se escribe inmediatamente hacia la salida estándar (por lo regular, la pantalla). Esto está bien para errores que se escriben en pantalla, pero puede tener un costo demasiado alto en cuanto a rendimiento al escribir archivos de registro en el disco. `clog` usa búfer para su salida, y por ende puede ser más eficiente.
- P ¿Por qué se crearon los flujos si `printf()` funciona bien?**
- R** `printf()` no soporta el poderoso sistema de tipos de C++, y no soporta clases definidas por el usuario.
- P ¿Cuándo se utilizaría `putback()`?**
- R** Cuando se utilice una operación de lectura para determinar si un carácter es válido, y otra operación de lectura diferente (tal vez un objeto diferente) necesite que el carácter esté en el búfer. Esto se utiliza más comúnmente cuando se analiza sintácticamente un archivo; por ejemplo, el compilador de C++ podría utilizar `putback()`.
- P ¿Cuándo se utilizaría `ignore()`?**
- R** Un uso común es después de utilizar `get()`. Como `get()` deja el carácter de terminación en el búfer, es común llamar a `ignore(1, '\n')`; inmediatamente después de una llamada a `get()`. Esto se utiliza con frecuencia en el análisis sintáctico.
- P Mis amigos utilizan `printf()` en sus programas de C++. ¿Puedo hacerlo yo?**
- R** Claro. Ganará algo de conveniencia, pero sacrificará la seguridad de los tipos.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

16

### Cuestionario

1. ¿Qué es el operador de inserción, y qué hace?
2. ¿Qué es el operador de extracción, y qué hace?
3. ¿Cuáles son las tres formas de utilizar `cin.get()`, y cuáles son sus diferencias?
4. ¿Cuál es la diferencia entre `cin.read()` y `cin.getline()`?
5. ¿Cuál es el ancho predeterminado para enviar como salida un entero largo mediante el operador de inserción?
6. ¿Cuál es el valor de retorno del operador de inserción?
7. ¿Qué parámetro lleva el constructor para un objeto `ofstream`?
8. ¿Qué hace el argumento `ios::ate`?

### Ejercicios

1. Escriba un programa que escriba en los cuatro objetos `iostream` estándar: `cin`, `cout`, `cerr` y `clog`.
2. Escriba un programa que pida al usuario que escriba su nombre completo y luego lo despliegue en pantalla.
3. Modifique el listado 16.9 para que haga lo mismo, pero sin utilizar `putback()` ni `ignore()`.
4. Escriba un programa que tome un nombre de archivo como parámetro y que abra el archivo para lectura. Lea todos los caracteres del archivo y despliegue en la pantalla sólo las letras y los signos de puntuación. (Ignore todos los caracteres no imprimibles.) Luego el programa deberá cerrar el archivo y terminar.
5. Escriba un programa que despliegue sus argumentos de la línea de comandos en orden inverso, y que no despliegue el nombre del programa.



# SEMANA 3

Día 17

## Espacios de nombres

Una nueva adición para el ANSI de C++ es el uso de espacios de nombres para ayudar a que los programadores eviten conflictos de nombres al utilizar más de una biblioteca. Hoy aprenderá lo siguiente:

- Cómo se resuelven por medio del nombre las funciones y las clases
- Cómo crear un espacio de nombres
- Cómo utilizar un espacio de nombres
- Cómo utilizar el espacio de nombres estándar std

## Comencemos con los espacios de nombres

Los conflictos de nombres han sido una fuente de irritación para los desarrolladores de C y de C++. La estandarización ANSI ofrece una oportunidad de resolver este problema mediante el uso de *espacios de nombres*, pero tome en cuenta esto: no todos los compiladores soportan esta característica. Los compiladores g++ versiones 2.9.5 y posteriores sí soportan esta característica. El compilador g++ versión 2.7.2 es uno de esos que no la soporta y emitirá el siguiente mensaje:

```
warning: namespaces are mostly broken in this version of g++
```

Un conflicto de nombres ocurre cuando se encuentra un nombre duplicado con el mismo alcance en dos partes del programa. Esto ocurre con más frecuencia en paquetes de bibliotecas distintos. Por ejemplo, una biblioteca de clases contenedoras muy probablemente declarará e implementará una clase llamada `List`. (Aprenderá más sobre las clases contenedoras en el día 19, “Plantillas”.)

No es sorprendente encontrar también una clase `List` en una biblioteca de manejo de ventanas. Suponga que necesita mantener una lista de ventanas para su aplicación, o que está utilizando la clase `List` que se encuentra en la biblioteca de clases contenedoras. Al declarar una instancia de la clase `List` de la biblioteca de ventanas para guardar sus ventanas, descubre que las funciones miembro que quiere llamar no están disponibles. El compilador ha igualado su declaración de `List` con la clase `List` contenedora de la biblioteca estándar, pero la que realmente quiere es la clase `List` que se encuentra en la biblioteca de ventanas específica del fabricante.

Los espacios de nombres se utilizan para particionar el espacio de nombres global y para eliminar, o por lo menos reducir, los conflictos de nombres. Los espacios de nombres son parecidos en cierta forma a las clases, y la sintaxis es muy similar.

Los elementos declarados dentro del espacio de nombres son propiedad del mismo. Todos los elementos dentro de un espacio de nombres tienen visibilidad pública. Los espacios de nombres se pueden anidar dentro de otros espacios de nombres. Las funciones se pueden definir dentro o fuera del cuerpo del espacio de nombres. Si una función se define fuera del cuerpo del espacio de nombres, debe ser identificada por el nombre del espacio de nombres.

## Cómo se resuelven por medio del nombre las funciones y las clases

A medida que analiza sintácticamente el código fuente y construye una lista de nombres de funciones y variables, el compilador revisa si existen conflictos de nombres. Los conflictos que no puede resolver el compilador, tal vez pueda resolverlos el enlazador.

El compilador no puede revisar conflictos de nombres entre unidades de traducción (por ejemplo, archivos objeto); ése es el propósito del enlazador. Por lo tanto, el compilador ni siquiera emitirá una advertencia.

Es muy común que el enlazador (`ld` en Linux) falle con el mensaje `Identifier multiply defined` (`identifier` es algún tipo con nombre). Verá este mensaje si ha definido el mismo nombre con el mismo alcance en diferentes unidades de traducción. Si redefines un nombre dentro de un solo archivo que tenga el mismo alcance, obtendrá un error de compilación.

Al compilar y enlazar el siguiente ejemplo, el enlazador producirá un mensaje de error:

```
// archivo primero.cxx
int valorEntero = 0 ;
int main( ) {
    int valorEntero = 0 ;
    // . . .
```

```
    return 0 ;
}

// archivo segundo.cxx
int valorEntero = 0 ;
// fin de segundo.cxx
```

El enlazador GNU anuncia los siguientes mensajes de diagnóstico:

```
segundo.cc: multiple definition of `valorEntero'
primero.cc: primero defined here
```

Si estos nombres tuvieran un alcance distinto, ni el compilador ni el enlazador tendrían problemas.

También es posible recibir una advertencia del compilador con respecto al *ocultamiento de identificadores*. El compilador debe advertir en *primero.cxx* que la variable *valorEntero* de *main()* está ocultando la variable global que tiene el mismo nombre.

Para usar la variable *valorEntero* declarada fuera de *main()*, debe establecer en forma explícita que desea utilizar la variable global. Considere este ejemplo, el cual asigna el valor 10 a la variable *valorEntero* que está fuera de *main()*, y no a la variable *valorEntero* declarada dentro de *main()*:

```
// archivo primero.cxx
int valorEntero = 0 ;
int main( )
{
    int valorEntero = 0 ;
    ::valorEntero = 10 ; //asignar a "valorEntero" global
    // . .
    return 0 ;
}

// archivo segundo.cxx
int valorEntero = 0 ;
// fin de segundo.cxx
```



### Nota

Observe el uso del operador de resolución de ámbito :: el cual indica que se hace referencia a *valorEntero* como global, no como local.

El problema con los dos enteros globales definidos fuera de cualquier función es que tienen el mismo nombre y visibilidad, y esto producirá un error de enlace.

 Nota

El término *visibilidad* se utiliza para designar el alcance de un objeto definido, ya sea una variable, una clase o una función. Por ejemplo, una variable declarada y definida fuera de cualquier función tiene un alcance de *archivo*, o *global*. La visibilidad de esta variable va desde el punto de su definición hasta el fin del archivo. Una variable que tenga un alcance de *bloque*, o *local*, se encuentra dentro de una estructura de bloque. Los ejemplos más comunes son las variables definidas dentro de funciones. El siguiente ejemplo muestra el alcance de las variables.

```
int enteroConAlcanceGlobal = 5 ;
void f( )
{
    int enteroConAlcanceLocal = 10 ;
}
int main( )
{
    int enteroConAlcanceLocal = 15 ;
    {
        int otroLocal = 20 ;
        int enteroConAlcanceLocal = 30 ;
    }
    return 0 ;
}
```

La primera definición `int`, `enteroConAlcanceGlobal`, tiene visibilidad dentro de las funciones `f()` y `main()`. La siguiente definición se encuentra dentro de la función `f()` y se llama `enteroConAlcanceLocal`. Esta variable tiene alcance local, lo que significa que sólo es visible dentro del bloque que la define.

La función `main()` no puede tener acceso a la variable `enteroConAlcanceLocal` de la función `f()`. Cuando la función termina, `enteroConAlcanceLocal` queda fuera de alcance. La tercera definición, también llamada `enteroConAlcanceLocal`, se encuentra en la función `main()`. Esta variable tiene alcance de bloque.

Observe que la variable `enteroConAlcanceLocal` de `main()` no tiene conflicto con la variable `enteroConAlcanceLocal` de `f()`. Las dos definiciones que siguen, `otroLocal` y `enteroConAlcanceLocal`, tienen alcance de bloque. Tan pronto como la ejecución del programa llega a la llave de cierre, estas dos variables pierden su visibilidad.

Observe que esta variable `enteroConAlcanceLocal` está ocultando a la variable `enteroConAlcanceLocal` definida justo antes de la llave de apertura (la segunda variable `enteroConAlcanceLocal` definida en el programa). Cuando el programa pasa más allá de la llave de cierre, la segunda variable `enteroConAlcanceLocal` definida recupera su visibilidad. Ningún cambio realizado a la variable `enteroConAlcanceLocal` definida dentro de las llaves afecta el contenido de la variable `enteroConAlcanceLocal` externa.

**Nota**

Los nombres pueden tener *enlace interno* y *enlace externo*. Estos dos términos se refieren al uso o disponibilidad de un nombre entre múltiples unidades de traducción o dentro de una sola unidad de traducción. Cualquier nombre que tenga enlace interno puede ser referido sólo dentro de la unidad de traducción en la que está definido. Por ejemplo, una variable definida para tener enlace interno puede ser compartida por funciones que estén dentro de la misma unidad de traducción. Los nombres que tengan enlace externo están disponibles para otras unidades de traducción. El siguiente ejemplo demuestra el funcionamiento de los enlaces interno y externo.

```
// archivo: primero.cxx
int intExterno = 5 ;
const int j = 10 ;
int main()
{
    return 0 ;
}

// archivo: segundo.cxx
extern int intExterno ;
int unIntExterno = 10 ;
const int j = 10 ;
```

La variable `intExterno` definida en `primero.cxx` tiene enlace externo. Aunque se define en `primero.cxx`, `segundo.cxx` también puede tener acceso a ella. Las dos “`j`” de ambos archivos son `const` y por lo tanto tienen enlace interno de manera predeterminada. Puede evitar el `const` predeterminado si proporciona una declaración explícita, como se muestra a continuación:

```
// archivo: primero.cxx
extern const int j = 10 ;

// archivo: segundo.cxx
extern const int j ;
#include <iostream>
int main()
{
    std::cout << "j vale " << j << std::endl ;
    return 0 ;
}
```

Observe que este ejemplo llama a `cout` con la designación de espacio de nombres `std`; esto le permite hacer referencia a todos los objetos “estándar” de la biblioteca ANSI estándar. Al crearlo, este ejemplo despliega lo siguiente:

```
j vale 10
```

El comité de estándares desaprueba el siguiente uso:

```
static int staticInt = 10 ;
int main()
{
    // ...
}
```

El uso de `static` para limitar el alcance de las variables externas ya no se recomienda y eventualmente podría convertirse en algo ilegal. Ahora se deben usar espacios de nombres en lugar de `static`.

### DEBE

**DEBE** utilizar espacios de nombres en lugar de la palabra reservada `static`.

### NO DEBE

**NO DEBE** aplicar la palabra reservada `static` a una variable definida con alcance de archivo.

## Creación de un espacio de nombres

La sintaxis para la declaración de un espacio de nombres es similar a la sintaxis para una declaración de un tipo `struct` o de una clase: primero, aplique la palabra reservada `namespace` seguida de un nombre opcional para el espacio de nombres, y luego una llave de apertura. El espacio de nombres se concluye con una llave de cierre, sin utilizar punto y coma al final.

He aquí un ejemplo:

```
namespace Ventana
{
    void mover(int x, int y) ;
}
```

El nombre `Ventana` identifica de forma única al espacio de nombres. Puede tener muchas ocurrencias del nombre que identifica a un espacio de nombres. Estas múltiples ocurrencias pueden estar dentro de un solo archivo o entre múltiples unidades de traducción. El espacio de nombre `std` de la biblioteca estándar de C++ es un excelente ejemplo de esta característica. Esto tiene sentido ya que la biblioteca estándar es un agrupamiento lógico de funcionalidad.

El principal concepto detrás de los espacios de nombres es agrupar elementos relacionados en un área especificada (todos bajo un mismo nombre). El siguiente es un breve ejemplo de un espacio de nombres que abarca varios archivos de encabezado:

```
// encabezado1.h
namespace Ventana
{
    void mover(int x, int y) ;
}

// encabezado2.h
```

```
namespace Ventana
{
    void cambiarTamaño(int x, int y) ;
}
```

## Declaración y definición de tipos

Puede declarar y definir tipos y funciones dentro de espacios de nombres. Claro que ésta es una cuestión de diseño y de mantenimiento. Un buen diseño implica que debe separar la interfaz de la implementación. Debe seguir este principio no sólo con las clases, sino también con los espacios de nombres. El siguiente ejemplo muestra un espacio de nombres desordenado y mal definido:

```
namespace Ventana {
    // . . . otras declaraciones y definiciones de variables.
    void mover(int x, int y) ; // declaraciones
    void cambiarTamaño(int x, int y) ;
    // . . . otras declaraciones y definiciones de variables.

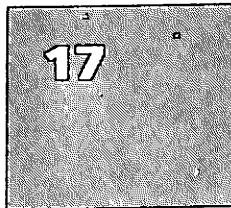
    void mover(int x, int y)
    {
        if(x < MAX_PANTALLA_X && x > 0)
            if(y < MAX_PANTALLA_Y && y > 0)
                plataforma.mover(x, y) ; // rutina específica
    }

    void cambiarTamaño(int x, int y)
    {
        if(x < MAX_TAMANIO_X && x > 0)
            if(y < MAX_TAMANIO_Y && y > 0)
                plataforma.cambiarTamaño(x, y) ; // rutina específica
    }
    // . . . continúan las definiciones
}
```

¡Puede ver qué tan rápido se llenan los espacios de nombres! El ejemplo anterior tiene aproximadamente 20 líneas; imagínese si este espacio de nombres fuera cuatro veces más largo.

## Cómo definir funciones fuera de un espacio de nombres

Debe definir las funciones para los espacios de nombres fuera del cuerpo del espacio de nombres. Esto muestra una clara separación de la declaración de la función y su definición, y mantiene ordenado el espacio de nombres. Separar del espacio de nombres la definición de la función también le permite colocar dentro de un archivo de encabezado el espacio de nombres y sus declaraciones encarnadas; las definiciones se pueden colocar en un archivo de implementación.



He aquí un ejemplo:

```
// archivo encabezado.h
namespace Ventana {
    void mover(int x, int y) ;
    // otras declaraciones ...
}

// archivo impl.cxx
void Ventana::mover(int x, int y)
{
    // código para mover la ventana
}
```

## Cómo agregar nuevos miembros

Puede agregar nuevos miembros a un espacio de nombres sólo dentro de su cuerpo. No puede definir nuevos miembros utilizando sintaxis de identificación completa. Lo más que puede esperar de este estilo de definición es una queja del compilador. El siguiente ejemplo demuestra este error:

```
namespace Ventana {
    // muchas declaraciones
}
//...algo de código
int Ventana::nuevoEnteroEnEspacionombre ; // lo siento, no puedo hacer esto
```

La línea de código anterior no es válida. Su compilador (apegado al ANSI de C++) emitirá un diagnóstico que refleje el error. Para corregir el error (o evitarlo por completo) coloque la declaración dentro del cuerpo del espacio de nombres.

Todos los miembros que están dentro de un espacio de nombres son públicos. El siguiente código no compilará:

```
namespace Ventana {
    private:
        void mover(int x, int y) ;
}
```

## Cómo anidar espacios de nombres

Es posible anidar un espacio de nombres dentro de otro espacio de nombres. Se pueden anidar debido a que la definición de un espacio de nombres también es una declaración. Como con cualquier otro espacio de nombres, debe identificar cada elemento o función utilizando el nombre de cada espacio de nombres que lo contiene. Por ejemplo, a continuación se muestra la declaración de un espacio de nombres anidado dentro de otro espacio de nombres:

```
namespace Ventana {
    namespace Vidrio {
        void tamano(int x, int y) ;
    }
}
```

Para tener acceso a la función `tamanio()` fuera de `Ventana`, debe identificar a la función con ambos nombres de los espacios de nombres que la incluyen. El siguiente código muestra la identificación:

```
int main( )
{
    Ventana::Vidrio::tamanio(10, 20) ;
    return 0 ;
}
```

17

## Uso de un espacio de nombres

Ahora veamos un ejemplo del uso de un espacio de nombres y del operador de resolución de ámbito. El código declara primero todos los tipos y funciones a utilizar dentro del espacio de nombres llamado `Ventana`. Después de definir todo lo requerido, el ejemplo define todas las funciones miembro declaradas. Estas funciones miembro se definen fuera del espacio de nombres; los nombres se identifican en forma explícita por medio del operador de resolución de ámbito. El listado 17.1 muestra el uso de un espacio de nombres.

### ENTRADA LISTADO 17.1 Uso de un espacio de nombres

```
1: // Listado 17.1 Uso de un espacio de nombres
2:
3: #include <iostream.h>
4:
5:
6: namespace Ventana
7: {
8:     const int MAX_X = 30 ;
9:     const int MAX_Y = 40 ;
10:    class Vidrio
11:    {
12:        public:
13:            Vidrio();
14:            ~Vidrio();
15:            void tamanio(int x, int y);
16:            void mover(int x, int y);
17:            void mostrar();
18:        private:
19:            static int cnt;
20:            int x;
21:            int y;
22:    };
23: }
24:
25: int Ventana::Vidrio::cnt = 0;
26:
27: Ventana::Vidrio():
```

continúa

**LISTADO 17.1** CONTINUACIÓN

```
28: x(0),
29: y(0)
30: {}
31:
32: Ventana::Vidrio::~Vidrio() {}
33:
34: void Ventana::Vidrio::tamanio(int x, int y)
35: {
36:     if(x < Ventana::MAX_X && x > 0)
37:         Vidrio::x = x;
38:     if(y < Ventana::MAX_Y && y > 0)
39:         Vidrio::y = y;
40: }
41:
42: void Ventana::Vidrio::mover(int x, int y)
43: {
44:     if(x < Ventana::MAX_X && x > 0)
45:         Vidrio::x = x;
46:     if(y < Ventana::MAX_Y && y > 0)
47:         Vidrio::y = y;
48: }
49:
50: void Ventana::Vidrio::mostrar()
51: {
52:     std::cout << " x " << Vidrio::x;
53:     std::cout << " y " << Vidrio::y << std::endl;
54: }
55:
56: int main( )
57: {
58:     Ventana::Vidrio vidrio;
59:
60:     vidrio.mover(20, 20);
61:     vidrio.mostrar( );
62:     return 0 ;
63: }
```

**SALIDA**

x 20 y 20

**ANÁLISIS**

Observe que la clase Vidrio está anidada dentro del espacio de nombres Ventana. Ésta es la razón por la que se tiene que identificar el nombre Vidrio con Ventana::.

La variable estática cnt, la cual se declara en Vidrio en la línea 19, se define como siempre. Observe que, en las líneas 34 a 40, MAX\_X y MAX\_Y están identificadas completamente dentro de la función Vidrio::tamanio(). Esto se debe a que Vidrio está dentro del alcance; de no ser así, el compilador emitiría un diagnóstico de error. Esto también es cierto para la función Vidrio::mover().

Si compila este código con la versión 2.7.2 de g++, recibirá el siguiente mensaje:

warning: namespaces are mostly broken in this version of g++

La única solución es utilizar una versión más reciente que tenga soporte para los espacios de nombres.

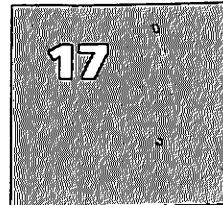
Lo que también es interesante es la identificación de `Vidrio::x` y de `Vidrio::y` dentro de ambas definiciones de funciones. ¿A qué se debe esto? Bueno, si usted hubiera escrito la función `Vidrio::mover()` de la siguiente manera, tendría un problema:

```
void Ventana::Vidrio::mover(int x, int y)
{
    if(x < Ventana::MAX_X && x > 0)
        x = x ;
    if(y < Ventana::MAX_Y && y > 0)
        y = y ;
    Platform::mover(x, y) ;
}
```

¿Ya vio cuál es la cuestión? Probablemente no obtendrá mucha información de su compilador; algunos no emiten ningún tipo de mensaje de diagnóstico.

La causa del problema son los argumentos de la función. Los argumentos `x` y `y` ocultan las instancias privadas de las variables `x` y `y` declaradas dentro de la clase `Vidrio`. En efecto, las instrucciones asignan `x` y `y` a sí mismas:

```
x = x ;
y = y ;
```



## Presentación de la palabra reservada `using`

La palabra reservada `using` se utiliza tanto para la directiva `using` como para la declaración `using`. La sintaxis de la palabra reservada `using` determina si el contexto es una directiva o una declaración.

### La directiva `using`

La directiva `using` expone efectivamente todos los nombres declarados en un espacio de nombres para que estén en el alcance actual. Puede hacer referencia a los nombres sin necesidad de identificarlos con su respectivo nombre de espacio de nombres. El siguiente ejemplo muestra el uso de la directiva `using`:

```
namespace Ventana {
    int valor1 = 20 ;
    int valor2 = 40 ;
}
...
Ventana::valor1 = 10 ;

using namespace Ventana ;
valor2 = 30 ;
```

El alcance de la directiva `using` empieza en su declaración y llega hasta el final del alcance actual. Observe que debe identificar a `valor1` para poder referenciarlo. La variable `valor2`

no requiere de identificación ya que la directiva introduce en el alcance actual todos los nombres que se encuentren en un espacio de nombres Ventana.

La directiva `using` se puede utilizar en cualquier nivel de alcance. Esto le permite utilizar la directiva dentro del alcance de bloque; cuando ese bloque quede fuera de alcance, también quedarán fuera de alcance todos los nombres que se encuentren dentro del espacio de nombres. El siguiente ejemplo muestra este comportamiento:

```
namespace Ventana {  
    int valor1 = 20 ;  
    int valor2 = 40 ;  
}  
//. . .  
void f()  
{  
    {  
        using namespace Ventana ;  
        valor2 = 30 ;  
    }  
    valor2 = 20 ; //error!  
}
```

La última línea de código de `f()`, `valor2 = 20 ;` es un error, ya que `valor2` no está definido. El nombre es accesible en el bloque anterior debido a que la directiva lleva el nombre dentro de ese bloque. Cuando ese bloque queda fuera de alcance, ocurre lo mismo con los nombres que se encuentran en el espacio de nombres llamado `Ventana`.

Los nombres de variables declaradas dentro de un alcance local ocultan cualquier nombre de un espacio de nombres que se introduzca en ese alcance. Este comportamiento es similar a la forma en que una variable local oculta a una variable global. Incluso si usted introduce un espacio de nombres después de una variable local, esa variable local ocultará el nombre del espacio de nombres. El siguiente ejemplo muestra esto:

```
namespace Ventana {  
    int valor1 = 20 ;  
    int valor2 = 40 ;  
}  
//. . .  
void f()  
{  
    int valor2 = 10 ;  
    using namespace Ventana ;  
    std::cout << valor2 << std::endl ;  
}
```

La salida de esta función es 10, no 40. Esta salida confirma el hecho de que el `valor2` que se encuentra en `f()` oculta al `valor2` que se encuentra en el espacio de nombres llamado `Ventana`. Si necesita utilizar un nombre dentro de un espacio de nombres, debe identificar ese nombre con el nombre del espacio de nombres.

Puede surgir una ambigüedad si se utiliza un nombre que esté definido tanto globalmente como dentro de un espacio de nombres. La ambigüedad surge sólo si se utiliza el nombre, y no cuando se introduce un espacio de nombres. Esto se demuestra en el siguiente fragmento de código:

```
namespace Ventana {  
    int valor1 = 20 ;  
}  
//...  
using namespace Ventana ;  
int valor1 = 10 ;  
void f( )  
{  
    valor1 = 10 ;  
}
```

La ambigüedad ocurre dentro de la función `f()`. La directiva lleva efectivamente a `Ventana::valor1` al espacio de nombres global; como ya hay un `valor1` definido en forma global, el uso de `valor1` en `f()` es un error. Observe que si se quitara la línea de código de `f()`, no existiría ningún error.

17

## La declaración using

La declaración `using` es similar a la directiva `using` con la excepción de que la declaración proporciona un nivel más fino de control. Dicho de manera más específica, la declaración `using` se utiliza para declarar un nombre específico (que pertenece a un espacio de nombres) para que esté en el alcance actual. Así puede referirse al objeto especificado sólo por su nombre. El siguiente ejemplo muestra el uso de la declaración `using`:

```
namespace Ventana {  
    int valor1 = 20 ;  
    int valor2 = 40 ;  
    int valor3 = 60 ;  
}  
//...  
using Ventana::valor2 ; //llevar a valor2 hacia el alcance actual  
Ventana::valor1 = 10 ; //se debe identificar a valor1  
valor2 = 30 ;  
Ventana::valor3 = 10 ; // se debe identificar a valor3
```

La declaración `using` agrega el nombre especificado al alcance actual. La declaración no afecta los otros nombres que están dentro del espacio de nombres. En el ejemplo anterior se hace referencia a `valor2` sin identificación, pero `valor1` y `valor3` necesitan ser identificados. La declaración `using` proporciona un mayor control sobre los nombres de los espacios de nombres que se llevan hacia un alcance determinado. Esto contrasta con la directiva que lleva hacia un alcance determinado a todos los nombres que se encuentran dentro de un espacio de nombres.

Después de llevar un nombre a un alcance, éste permanece visible hasta el fin de ese alcance. Este comportamiento es igual que con cualquier otra declaración. Una declaración `using` se puede utilizar en el espacio de nombres global o dentro de cualquier alcance local.

Sería un error introducir un nombre en un alcance local en el que se haya declarado un nombre de un espacio de nombres. Lo opuesto también es un error. El siguiente ejemplo muestra esto:

```
namespace Ventana {  
    int valor1 = 20 ;  
    int valor2 = 40 ;  
}  
/. . .  
void f()  
{  
    int valor2 = 10 ;  
    using Ventana::valor2 ; // declaración múltiple  
    std::cout << valor2 << std::endl ;  
}
```

La segunda línea de la función `f()` producirá un error de compilación debido a que el nombre `valor2` ya está definido. El mismo error ocurre si la declaración `using` se introduce antes de la definición del `valor2` local.

Cualquier nombre introducido en el alcance local con una declaración `using` oculta a cualquier nombre que esté fuera de ese alcance. El siguiente código demuestra este comportamiento:

```
namespace Ventana {  
    int valor1 = 20 ;  
    int valor2 = 40 ;  
}  
int valor2 = 10 ;  
/. . .  
void f()  
{  
    using Ventana::valor2 ;  
    std::cout << valor2 << std::endl ;  
}
```

La declaración `using` de `f()` oculta al `valor2` definido en el espacio de nombres global.

Como mencioné anteriormente, una declaración `using` le brinda un control más fino sobre los nombres introducidos desde un espacio de nombres. Una directiva `using` lleva a todos los nombres desde un espacio de nombres hasta el alcance actual. Es preferible utilizar una declaración en lugar de una directiva, ya que una directiva anula el propósito del mecanismo del espacio de nombres. Una declaración es más definitiva ya que se están identificando explícitamente los nombres que se quieren introducir en un alcance. Una declaración `using` no contaminará el espacio de nombres global, como ocurre con una directiva `using` (a menos que se declaren todos los nombres encontrados en el espacio de nombres). El ocultamiento de nombres, la contaminación del espacio de nombres global y la ambigüedad se reducen a un nivel más manejable por medio de la declaración `using`.

## Uso del alias de un espacio de nombres

El *alias* de un espacio de nombres está diseñado para dar a un espacio de nombres un nombre distinto del que tiene. Un alias proporciona un término abreviado para que usted lo utilice para referirse a un espacio de nombres. Esto es cierto sobre todo cuando el nombre de un espacio de nombres es muy largo; crear un alias puede ayudar a no escribir mucho y en forma repetitiva. Analice este ejemplo:

```
namespace la_compania_de_software {  
    int valor ;  
    // . . .  
}  
la_compania_de_software::valor = 10 ;  
. . .  
namespace LCS = la_compania_de_software ;  
LCS::valor = 20 ;
```

17

Una desventaja es que el alias que usted elija puede tener conflicto con un nombre existente. De ser así, el compilador se dará cuenta del conflicto y podrá resolverlo cambiando el nombre del alias.

## Uso del espacio de nombres sin nombre

Un espacio de nombres sin nombre es simplemente eso: un espacio de nombres que no tiene un nombre. Un uso común de los espacios sin nombre es proteger los datos globales de los potenciales conflictos de nombre entre unidades de traducción. Cada unidad de traducción tiene su propio y único espacio de nombres sin nombre. Puede hacer referencia a todos los nombres definidos dentro del espacio de nombres sin nombre (dentro de cada unidad de traducción) sin necesidad de identificarlos explícitamente. Lo siguiente es un ejemplo de dos espacios de nombre sin nombre que se encuentran en dos archivos separados:

```
// archivo: uno.cxx  
namespace {  
    int valor ;  
    char ap(char *ap) ;  
    // . . .  
}  
  
// archivo: dos.cxx  
namespace {  
    int valor ;  
    char ap(char *ap) ;  
    // . . .  
}  
int main( )  
{  
    char c = ap(aptr) ;  
}
```

Cada uno de los nombres, `valor` y las funciones `ap()`, son distintos entre cada archivo. Para hacer referencia a un nombre (espacio de nombre sin nombre) que esté dentro de una unidad de traducción, se utiliza el nombre sin identificación. Este uso se demuestra en el ejemplo anterior con la llamada a la función `ap()`. Esta sintaxis implica el uso de una directiva `using` para los objetos a los que se hace referencia desde un espacio de nombres sin nombre. Debido a esto, no se puede tener acceso a los miembros de un espacio de nombres sin nombre en otra unidad de traducción. El comportamiento de un espacio de nombres sin nombre es el mismo que el de un objeto `static` que tenga enlace externo. Considere este ejemplo:

```
static int valor = 10 ;
```

Recuerde que el comité de estándares desaprueba este uso de la palabra reservada `static`. Los espacios de nombres existen ahora para reemplazar código, como se mostró anteriormente. Otra forma de ver a los espacios de nombres sin nombre es que son variables globales con enlace interno.

## Uso del espacio de nombres estándar `std`

El mejor ejemplo de los espacios de nombres se encuentra en la biblioteca estándar de C++. La biblioteca estándar está completamente encerrada dentro del espacio de nombres llamado `std`. Todas las funciones, clases, objetos y plantillas se declaran dentro del espacio de nombres `std`.

Sin duda, verá código como el siguiente:

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

Recuerde que la directiva `using` extrae todo del espacio de nombres con nombre. Es malo emplear la directiva `using` al utilizar la biblioteca estándar. ¿Por qué? Porque esto anula el propósito de utilizar un espacio de nombres; el espacio de nombres global se contaminará con todos los nombres encontrados en el encabezado. Tome en cuenta que todos los archivos de encabezado utilizan la característica de los espacios de nombres, así que si incluye múltiples archivos de encabezado y especifica la directiva `using`, todo lo que esté declarado en los encabezados estará en el espacio de nombres global. Considere que la mayoría de los ejemplos que vienen en este libro violan esta regla; esta acción no es un intento por defender la violación de la regla, sólo se hace para asegurar la brevedad de los ejemplos. Lo que usted debe utilizar es la declaración `using`, como en el siguiente ejemplo:

```
#include <iostream>
using std::cin ;
using std::cout ;
using std::endl ;
int main( )
{
    int valor = 0 ;
    cout << "Así que, ¿cuántos huevos dijo que quería?" << endl ;
```

```
    cin >> valor ;
    cout << valor << " huevos estrellados!" << endl ;
    return(0) ;
}
```

A continuación se muestra un ejemplo de la ejecución del programa:

Así que, ¿cuántos huevos dijo que quería?

```
4
14 huevos estrellados!
```

Como alternativa, podría identificar completamente los nombres que utiliza, como en el siguiente código de muestra:

```
#include <iostream>
int main( )
{
    int valor = 0 ;
    std::cout << "¿Cuántos huevos quería?" << std::endl ;
    std::cin >> valor ;
    std::cout << valor << " huevos estrellados!" << std::endl ;
    return(0) ;
}
```



La salida del programa se muestra a continuación:

```
¿Cuántos huevos quería?
4
14 huevos estrellados!
```

Esto podría ser apropiado para programas cortos, pero se puede volver algo incómodo para cualquier cantidad considerable de código. ¡Imagine tener que escribir el prefijo `std::` para todos los nombres que utilice que se encuentren en la biblioteca estándar!

## Resumen

La creación de un espacio de nombres es muy similar a la declaración de clases. Hay un par de diferencias que vale la pena mencionar. En primer lugar, después de la llave de cierre de un espacio de nombres no se pone punto y coma. En segundo lugar, un espacio de nombres es abierto, mientras que una clase es cerrada. Esto significa que puede seguir definiendo el espacio de nombres en otros archivos o en secciones separadas de un solo archivo.

Cualquier cosa que se pueda declarar se puede insertar en un espacio de nombres. Si diseña clases para una biblioteca reutilizable, debe utilizar la característica de espacio de nombres. Las funciones declaradas dentro de un espacio de nombres se deben definir fuera del cuerpo de ese espacio de nombres. Esto provoca que la interfaz se separe de la implementación y también evita que el espacio de nombres se llene excesivamente.

Es posible anidar los espacios de nombres. Un espacio de nombres es una declaración; este hecho le permite anidar los espacios de nombres. No olvide que debe identificar completamente los nombres que estén anidados.

La directiva `using` se utiliza para exponer en el alcance actual a todos los nombres que se encuentran en un espacio de nombres. Esto contamina el espacio de nombres global con todos los nombres que se encuentran en el espacio de nombres con nombre. Por lo general, es una mala práctica utilizar la directiva `using`, especialmente con respecto a la biblioteca estándar. Es mejor utilizar declaraciones `using`.

La declaración `using` se utiliza para exponer en el alcance actual a un espacio de nombres específico. Esto le permite hacer referencia al objeto sólo con su nombre.

Un alias de un espacio de nombres es similar en naturaleza a un `typedef`. Un alias de un espacio de nombres le permite crear otro nombre para un espacio de nombres que ya tenga nombre. Esto puede ser bastante útil al utilizar un espacio de nombres que tenga un nombre largo.

Todos los archivos pueden contener un espacio de nombres sin nombre. Éste, como su nombre lo implica, es un espacio de nombres que no tiene nombre, y le permite utilizar los nombres que están dentro del espacio de nombres, sin necesidad de identificación.

Hace que los nombres del espacio de nombres sean locales para la unidad de traducción. Los espacios de nombres sin nombre son lo mismo que declarar una variable global con la palabra reservada `static`.

La biblioteca estándar de C++ está encerrada en un espacio de nombres llamado `std`. Evite utilizar la directiva `using` al usar la biblioteca estándar; mejor utilice la declaración `using`.

## Preguntas y respuestas

**P ¿Tengo que utilizar espacios de nombres?**

**R** No, puede escribir programas simples e ignorar por completo los espacios de nombres. Asegúrese de utilizar las viejas bibliotecas estándar (por ejemplo, `#include <string.h>`) en lugar de las nuevas (por ejemplo `#include <cstring>`).

**P ¿Cuáles son los dos tipos de instrucciones que hay con la palabra reservada `using`? ¿Qué diferencias hay entre esos dos tipos de instrucciones?**

**R** La palabra reservada `using` se puede utilizar para las directivas `using` y para las declaraciones `using`. Una directiva `using` permite que se utilicen todos los nombres que se encuentran en un espacio de nombres como si fueran nombres normales. Una declaración `using`, por el contrario, permite que el programa utilice un nombre individual de un espacio de nombres sin tener que identificarlo con el identificador de espacio de nombres.

**P ¿Qué son los espacios de nombres sin nombre? ¿Para qué los necesitamos?**

**R** Los espacios de nombres sin nombre son espacios de nombres que no tienen nombre. Se utilizan para “envolver” una colección de declaraciones y protegerla contra posibles conflictos entre nombres. Los nombres que se encuentran en un espacio de nombres sin nombre no se pueden utilizar fuera de la unidad de traducción en la que está declarado el espacio de nombres.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, "Respuestas a los cuestionarios y ejercicios", y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Puedo utilizar nombres definidos en un espacio de nombres sin utilizar la palabra reservada `using`?
2. ¿Cuáles son las principales diferencias entre espacios de nombres normales y espacios de nombres sin nombre?
3. ¿Cuál es el espacio de nombres estándar?

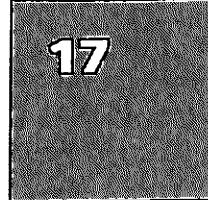
### Ejercicios

1. CAZA ERRORES: ¿Qué está mal en este programa?

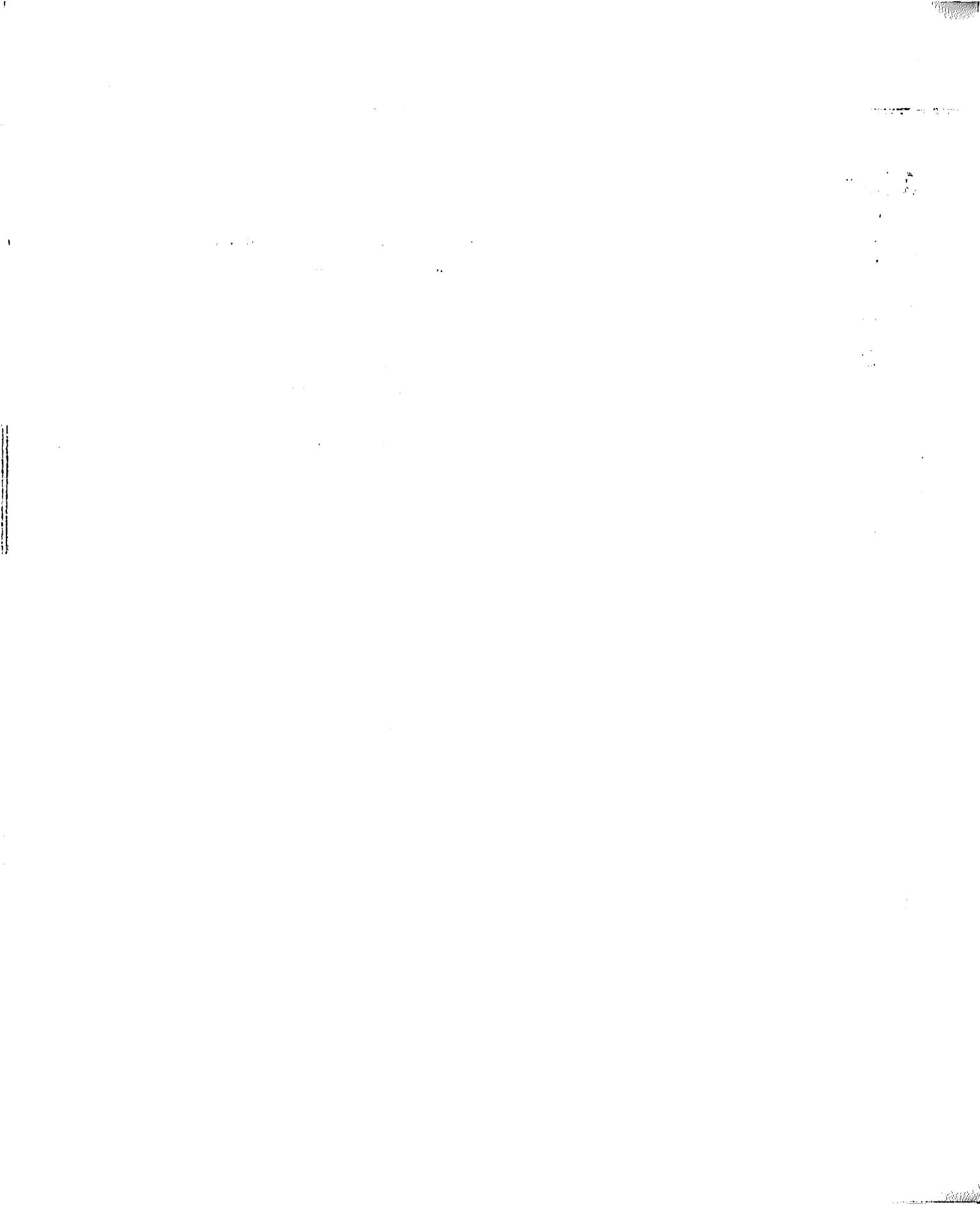
```
#include <iostream>

int main()
{
    cout << "¡Hola, mundo!" << endl;
    return 0;
}
```

2. Mencione tres formas de solucionar el problema del ejercicio 1.



17



# SEMANA 3

Día 18

## Análisis y diseño orientados a objetos

Es fácil enfocarse en la sintaxis de C++ y olvidarse de cómo y por qué se utilizan estas técnicas para construir programas. Hoy aprenderá lo siguiente:

- Cómo utilizar el análisis orientado a objetos para modelar el problema que está tratando de resolver
- Cómo utilizar el diseño orientado a objetos para crear una solución convincente, extensiva y confiable
- Cómo utilizar el UML (Lenguaje de Modelado Unificado) para documentar su análisis y su diseño

### ¿Es C++ un lenguaje orientado a objetos?

C++ se creó como un puente entre la programación orientada a objetos y el lenguaje C, el lenguaje de programación más popular en el mundo para desarrollo de software comercial. El objetivo era proporcionar un diseño orientado a objetos para una plataforma veloz de desarrollo de software comercial.

C se desarrolló como intermediario entre los lenguajes de aplicaciones de negocios de alto nivel, como COBOL, y el lenguaje ensamblador, que es de alto rendimiento pero difícil de utilizar. C debía implementar la programación “estructurada”, en la que los problemas se “descomponían” en unidades más pequeñas de actividades repetitivas llamadas procedimientos.

Los programas que estamos escribiendo en el comienzo de este nuevo siglo son mucho más complejos que los que se escribieron en el principio de la última década. Los programas creados en lenguajes procedurales tienden a ser difíciles de manejar, difíciles de mantener e imposibles de extender. Las interfaces gráficas de usuario, Internet, la telefonía digital y una gran variedad de nuevas tecnologías han incrementado en forma dramática la complejidad de los proyectos de programación. A su vez, las expectativas del consumidor en cuanto a la calidad de la interfaz de usuario se están elevando.

Ante esta complejidad creciente, los desarrolladores analizaron profundamente el estado de la industria. Lo que descubrieron fue, en el mejor de los casos, desalentador. El software se terminaba demasiado tarde, no estaba completo, era defectuoso, nada confiable y costoso. A menudo los proyectos se pasaban del presupuesto y salían muy tarde al mercado. El costo de mantener y construir estos proyectos era prohibitivo, y se desperdiciaba una tremenda cantidad de dinero.

El desarrollo de software orientado a objetos ofrece una salida para este abismo. Los lenguajes de programación orientada a objetos crean un sólido enlace entre las estructuras de datos y los métodos que manipulan esos datos. Lo que es más importante, en la programación orientada a objetos ya no se piensa en las estructuras de datos y las funciones manipuladoras; en vez de esto, se piensa en los objetos (cosas).

El mundo está lleno de cosas: autos, perros, árboles, nubes, flores, etc. Cada cosa tiene características (veloz, amigable, café, esponjoso, bonito). Casi todas las cosas tienen un comportamiento (mover, ladrar, crecer, llover, marchitarse). No pensamos en los datos acerca de un perro y cómo los podríamos manipular; pensamos en un perro como una cosa de este mundo, qué apariencia tiene y qué hace.

## Qué son los modelos

Si vamos a manejar la complejidad, debemos crear un modelo del universo. El objetivo del modelo es crear una abstracción significativa del mundo real. Dicha abstracción debe ser más simple que el mundo real, pero también debe reflejar al mundo real en forma precisa, para que podamos utilizar el modelo para predecir el comportamiento de las cosas del mundo real.

El globo terráqueo de un niño es un modelo clásico. El modelo no es la cosa en sí; nunca podríamos confundir el globo terráqueo de un niño con el planeta Tierra, pero uno proyecta al otro lo suficientemente bien como para que podamos aprender sobre la Tierra estudiando el globo terráqueo.

Hay, desde luego, simplificaciones considerables. El globo terráqueo de un niño nunca tiene lluvia, inundaciones, terremotos, etcétera, pero puedo utilizar ese globo para predecir cuánto tiempo me tomará volar desde mi hogar hasta Indianápolis en caso de que alguna vez necesite ir a explicarle a la administración superior por qué mi manuscrito llegó tarde (“verán, iba muy bien, pero luego me perdí en una metáfora y me tardé horas en salir”).

Un modelo que no es más simple que la cosa que se está modelando no sirve de mucho. Steven Wright bromea acerca de esto: “Tengo un mapa en el que una pulgada equivale a una pulgada. Vivo en E5”.

El diseño de software orientado a objetos trata acerca de la construcción de buenos modelos. Consta de dos componentes importantes: un lenguaje de modelado y un proceso.

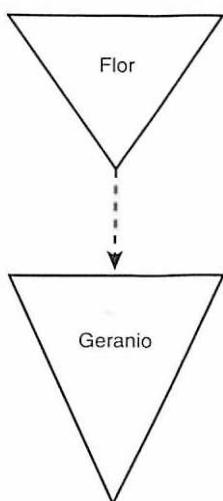
## Diseño de software: el lenguaje de modelado

El *lenguaje de modelado* es el aspecto menos importante del análisis y diseño orientados a objetos; por desgracia, tiende a recibir la mayor atención. Un lenguaje de modelado es sólo una convención sobre la forma de dibujar un modelo en papel. Podemos decidir fácilmente dibujar nuestras clases como triángulos y la relación de herencia como una línea punteada. De ser así, podríamos modelar un geranio como se muestra en la figura 18.1.

18

**FIGURA 18.1**

Generalización/  
especialización.



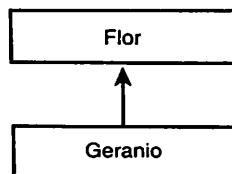
Puede ver en la figura que un geranio es un tipo especial de flor. Si usted y yo acordamos dibujar nuestros diagramas de herencia (generalización/especialización) de esta manera, nos entenderemos a la perfección. Con el tiempo, probablemente nos gustaría modelar muchas relaciones complejas para desarrollar nuestro propio conjunto complicado de convenciones y reglas para dibujar diagramas.

Claro que necesitaremos explicar nuestras convenciones a las demás personas con las que trabajemos, así como a cada nuevo empleado o colaborador. Tal vez interactuemos con otras compañías que tengan sus propias convenciones, y necesitemos tiempo para negociar una convención común y la forma de compensar los inevitables malos entendidos.

Sería más conveniente que todos en la industria acordaran usar un lenguaje común de modelado. (De hecho, sería conveniente que todos en el mundo acordaran usar un lenguaje hablado, pero hagamos una cosa a la vez.) El lenguaje unificado de desarrollo de software es el UML (Lenguaje de Modelado Unificado). El trabajo de UML es contestar preguntas como la siguiente: “¿Cómo se dibuja una relación de herencia?” El dibujo del geranio mostrado en la figura 18.1 se dibujaría en UML como se muestra en la figura 18.2.

**FIGURA 18.2**

Dibujo UML de especialización.



En UML, las clases se dibujan como rectángulos, y la herencia se dibuja como una línea con punta de flecha. Curiosamente, la flecha apunta de la clase más especializada a la clase más general. La dirección de la flecha es contraintuitiva para muchas personas, pero esto no es muy importante; cuando todos estamos de acuerdo, el sistema funciona perfectamente.

Los detalles de UML son bastante claros. Los diagramas no son difíciles de utilizar ni de entender, y los explicaré a medida que vayamos avanzando en esta lección, en lugar de tratar de enseñar UML fuera del contexto. Aunque es posible escribir un libro completo que trate sobre UML, la verdad es que el 90 % del tiempo sólo se utilizará un pequeño subconjunto de la notación de UML, y ese subconjunto se puede aprender fácilmente.

## Diseño de software: el proceso

El *proceso* del análisis y el diseño orientados a objetos es mucho más complejo e importante que el lenguaje de modelado. Así que, desde luego, es de lo que menos se oye hablar. Esto se debe a que el debate sobre los lenguajes de modelado está más arraigado; como industria, decidimos utilizar UML. El debate en cuanto al proceso continúa.

Un *metodologista* es alguien que desarrolla o estudia uno o más métodos. Por lo general, los metodologistas desarrollan y publican sus propios métodos. Un *método* es un lenguaje de modelado y un proceso. Tres de los principales metodologistas y sus métodos son Grady Booch, quien desarrolló el método Booch; Ivar Jacobson, quien desarrolló la ingeniería de software orientada a objetos; y James Rumbaugh, quien desarrolló la OMT (Tecnología de Modelado de Objetos). Juntos, estos tres hombres han creado *Objectory*, un método y

producto comercial de Rational Software, Inc. Los tres trabajan en Rational Software, en donde se les conoce afectuosamente como *los tres amigos*.

Esta lección sigue el método *Objectory*, pero no tan al pie de la letra. No lo sigo así porque no creo en la necesidad de apegarme estrictamente a la teoría académica (me interesa mucho más el mercadeo del producto que apegarme a un método). Otros métodos tienen algo que ofrecer, y tiendo a ser ecléctico; recojo piezas y pedazos a medida que avanzo y las uno todas en un marco de trabajo funcional.

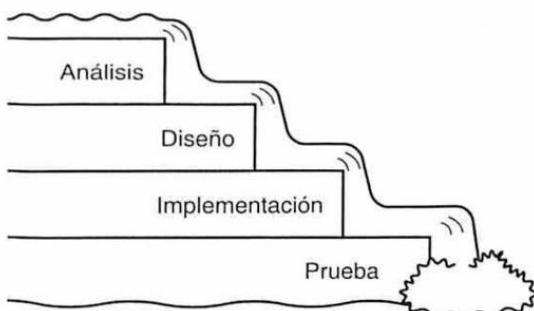
El proceso del diseño de software es *iterativo*. Esto significa que a medida que desarrollamos software, pasamos por todo el proceso en forma repetida esforzándonos por mejorar la comprensión de los requerimientos. El diseño dirige la implementación, pero los detalles descubiertos durante la implementación retroalimentan el diseño. Lo que es más importante, no tratamos de desarrollar ningún proyecto de tamaño ajustable en una sola dirección y con un solo orden; lo que hacemos es iterar sobre algunas piezas del proyecto, mejorando constantemente nuestro diseño y refinando nuestra implementación.

El desarrollo iterativo se puede distinguir del desarrollo en cascada. En el desarrollo en cascada, la salida de una etapa se convierte en la entrada de otra, y no hay regreso (vea la figura 18.3). En un proceso de desarrollo en cascada, los requerimientos se expresan con detalle, y los clientes autorizan ("Sí, esto es lo que yo quiero"); a continuación, los requerimientos se envían al diseñador, ya fijos. El diseñador crea el diseño (y qué maravilla para contemplar) y lo envía al programador, quien implementa el diseño. El programador a su vez entrega el código a una persona de control de calidad, quien prueba el código y luego lo libera al cliente. Suena perfecto en teoría, pero es un desastre en la práctica.

18

FIGURA 18.3

El método en cascada.



En el diseño iterativo, el visionario idea un concepto, y luego empezamos a trabajar en él para desarrollar los requerimientos. A medida que examinamos los detalles, la visión puede crecer y evolucionar. Cuando llevamos un buen avance en los requerimientos, empezamos el diseño, sabiendo perfectamente que las preguntas que surgen durante el diseño pueden ocasionar modificaciones a los requerimientos. A medida que trabajamos sobre el diseño, empezamos a crear prototipos y luego a implementar el producto. Las cuestiones que surgen en el desarrollo retroalimentan el diseño e incluso pueden influenciar nuestra comprensión de los requerimientos. Lo que es más importante, diseñamos e implementamos sólo piezas del producto completo, iterando repetidamente en las fases de diseño e implementación.

Aunque los pasos del proceso se repiten en forma iterativa, es casi imposible describirlos de manera tan cíclica. Por lo tanto, los describiré en secuencia: conceptualización, análisis, diseño, implementación, prueba y distribución. No me malentienda, en realidad pasamos por cada uno de estos pasos muchas veces durante el desarrollo de un solo producto. El proceso de diseño iterativo sólo es difícil de presentar y de comprender si avanzamos por cada paso en forma cíclica, por lo que los describo uno tras otro.

A continuación se muestran los pasos del proceso de diseño iterativo:

1. Conceptualización: cuando se desarrollan la visión y el propósito general del proyecto.
2. Análisis: determinar las necesidades de la organización (entender lo que debe hacer el software); aquí es donde se modelan las clases.
3. Diseño: crear el plano para solucionar el problema establecido.
4. Implementación: crear el sistema a partir del plano de diseño; aquí es donde se desarrolla el código (por ejemplo, en C++).
5. Prueba: asegurarse de que el sistema haga lo que se supone que debe de hacer.
6. Distribución: proporcionar el sistema a los usuarios.

Estos pasos son muy sencillos. Todo lo que resta son detalles.



Esta lección sólo cubre los tres primeros pasos: conceptualización, análisis y diseño. Los demás pasos están mucho más allá del alcance de esta lección introductoria. Puede encontrar muchos libros que traten estos temas con mucho detalle.

### Controversias

Existen interminables controversias sobre lo que ocurre en cada paso del proceso de diseño iterativo, e incluso sobre el nombre que se dé a esos pasos. He aquí el secreto: los pasos esenciales son los mismos en casi todos los procesos: encontrar lo que necesita crear, diseñar una solución e implementar ese diseño.

Aunque los grupos de noticias y las listas de correo de tecnología de objetos se desarrollan por separado, los fundamentos del análisis y el diseño orientados a objetos son bastante claros. En esta lección se muestra un método práctico para el proceso, el cual puede ser la base para que usted cree la arquitectura de su aplicación.

El objetivo de todo este trabajo es producir código que cumpla con los requerimientos establecidos, que sea confiable y que se pueda extender y mantener. Lo que es más importante, el objetivo es producir código de alta calidad que esté a tiempo y dentro del presupuesto establecido.

## Conceptualización: la visión

Todo buen software empieza con una visión. Un individuo tiene una idea sobre un producto y piensa que sería bueno crearlo. Raras veces los comités crean visiones apremiantes. La primera fase del análisis y el diseño orientados a objetos es capturar esta visión en una sola oración (o cuando mucho, en un párrafo corto). La visión se convierte en la guía fundamental del desarrollo, y el equipo que se reúne para implementar dicha visión debe regresar varias veces (y hacer las actualizaciones necesarias) a medida que avanza.

18

Incluso si la oración de la visión sale de un comité que se encuentre en el departamento de comercialización, se debe designar a una persona para que sea el “visionario”. El trabajo de esa persona es custodiar la luz sagrada. A medida que se progrese, los requerimientos evolucionarán. Las demandas del tiempo programado para la producción y del tiempo en el que el producto llegará al mercado pueden modificar lo que usted trata de lograr en la primera iteración del programa. Pero el visionario debe vigilar la idea esencial, para asegurar que lo que se produzca refleje la visión original con alta fidelidad. Esta dedicación implacable y este compromiso apasionado son los que conducen al buen término del proyecto. Si pierde el sentido de la visión, su producto está perdido.

## Análisis de los requerimientos

La fase de conceptualización, en la que se articula la visión, es muy breve. Puede ser sólo un destello de una idea, seguido del tiempo que lleva escribir lo que el visionario tiene en mente. A menudo, como experto orientado a objetos, usted se unirá al proyecto después de que la visión se haya articulado.

Algunas compañías confunden el enunciado de la visión con los requerimientos. Es necesaria una visión sólida, pero no es suficiente. Para pasar al análisis, debe entender la forma en que se va a utilizar el producto y cómo debe funcionar. El objetivo de la fase de análisis es articular y capturar estos requerimientos. El resultado de la fase de análisis es la producción de un documento de requerimientos. La primera sección del documento de requerimientos es el análisis de los casos de uso.

## Casos de uso

La fuerza impulsora en el análisis, diseño e implementación son los casos de uso. Un *caso de uso* es simplemente una descripción de alto nivel de la forma en que se va a utilizar el producto. Los casos de uso no sólo dirigen el análisis, también dirigen el diseño, ayudan a encontrar las clases y son muy importantes cuando se va a probar el producto.

Crear un conjunto resistente e integral de casos de uso puede ser la tarea individual más importante del análisis. Aquí es en donde se depende más de los expertos de dominio; estos expertos tienen la mayor parte de la información acerca de los requerimientos comerciales que se tratan de capturar.

Los casos de uso le dan muy poca importancia a la interfaz de usuario, y no le dan ninguna importancia al funcionamiento interno del sistema que se está creando. Cualquier sistema o persona que interactúa con el sistema se conoce como *actor*.

Para resumir, a continuación se muestran algunas definiciones:

- Caso de uso: Una descripción de cómo se va a utilizar el software.
- Expertos de dominio: Personas con experiencia en el *dominio* (área) de negocios para el que se está creando el producto.
- Actor: Cualquier persona o sistema que interactúa con el sistema que se está desarrollando.

Un caso de uso es una descripción de la interacción entre un actor y el sistema mismo. Para los propósitos de análisis del caso de uso, el sistema se trata como una “caja negra”. Un actor “envía un mensaje” al sistema, y sucede algo: se regresa información, se cambia el estado del sistema, la nave espacial cambia su dirección, o sucede cualquier otra cosa.

## Cómo identificar a los actores

Es importante observar que no todos los actores son personas. Los sistemas que interactúan con el sistema que se está creando también son actores. Por lo tanto, si fuéramos a crear un cajero automático, tanto el cliente como el empleado del banco pueden ser actores (así como otros sistemas con los que interactúe nuestro nuevo sistema, como un sistema de rastreo de hipotecas o de préstamos a estudiantes). Las características esenciales de los actores son las siguientes:

- Son externos al sistema
- Interactúan con el sistema

Por lo general, empezar es la parte más difícil del análisis de los casos de uso. A menudo, la mejor forma de avanzar es con una sesión de “lluvia de ideas”. Simplemente escriba la lista de personas y de sistemas que interactuarán con su nuevo sistema. Recuerde que cuando hablamos de *personas*, en realidad nos referimos a los papeles que juegan (el empleado del banco, el gerente, el cliente, etcétera). Una persona puede jugar más de un papel.

Para el ejemplo del cajero automático que acabamos de mencionar, podemos esperar que dicha lista incluya los siguientes papeles:

- El cliente
- El personal del banco
- Un sistema de soporte para la oficina
- La persona que llena con dinero el cajero automático

Al principio, no necesita ir más allá de la lista obvia. Generar hasta tres o cuatro actores puede ser suficiente para que pueda empezar a generar casos de uso. Cada uno de estos actores interactúa de distintas formas con el sistema. Debe capturar estas interacciones en sus casos de uso.

### Cómo determinar los primeros casos de uso

Debemos empezar con el papel del cliente. Podríamos hacer una lluvia de ideas para encontrar los siguientes casos de uso para un *cliente*:

- El cliente revisa los balances de una cuenta
- El cliente deposita dinero en su cuenta
- El cliente retira dinero de su cuenta
- El cliente transfiere dinero entre cuentas
- El cliente abre una cuenta
- El cliente cierra una cuenta

¿Deberíamos distinguir entre “El cliente deposita dinero en su cuenta de cheques” y “El cliente deposita dinero en su cuenta de ahorro”, o deberíamos combinar estas acciones (como lo hicimos en la lista anterior) en “El cliente deposita dinero en su cuenta”? La respuesta a esta pregunta depende de si esta distinción es significativa en el dominio.

Para determinar si estas acciones son uno o dos casos de uso, debe preguntarse si los *mecanismos* son distintos (el cliente tiene que hacer algo muy diferente con estos depósitos) y si los *resultados* son diferentes (el sistema contesta de forma distinta). La respuesta a ambas preguntas para la cuestión relacionada con el depósito es “no”: en esencia, el cliente deposita dinero de la misma forma en cualquier cuenta, y los resultados son casi iguales; el cajero automático responde incrementando el balance en la cuenta apropiada.

Dado que el actor y el sistema se comportan y responden más o menos de manera similar, sin importar si el depósito se hace en la cuenta de cheques o en la de ahorro, estos dos casos de uso son en realidad un solo caso de uso. Más adelante, cuando desarrollemos escenarios de los casos de uso, podremos probar las dos variaciones para ver si habría alguna diferencia.

Al ir pensando en cada actor, puede descubrir casos de uso adicionales al hacer estas preguntas:

- ¿Por qué el actor está usando este sistema?

*El cliente está usando el sistema para obtener efectivo, para hacer un depósito o para revisar el balance de una cuenta.*

- ¿Qué resultados quiere de cada petición el actor?

*Agregar efectivo a una cuenta u obtener efectivo para hacer una compra.*

- ¿Qué ocasionó que ahora el actor utilizara este sistema?

*Tal vez el actor haya recibido un ingreso o esté pensando en hacer una compra.*

- ¿Qué debe hacer el actor para utilizar el sistema?

*Meter su tarjeta en el cajero automático.*

*¡Ahá! Necesitamos un caso de uso para que el cliente inicie una sesión en el sistema.*

- ¿Qué información debe proporcionar al sistema el actor?

*Escribir un número de identificación personal.*

*¡Ahá! Necesitamos casos de uso para obtener y modificar el número de identificación personal.*

- ¿Qué información espera el actor obtener del sistema?

*Balances, entre otros.*

A menudo puede encontrar casos de uso adicionales si se enfoca en los atributos de los objetos que están en el dominio. El cliente tiene un nombre, un NIP (número de identificación personal) y un número de cuenta; ¿tenemos casos de uso para manejar estos objetos? Una cuenta tiene un número, un balance y un historial de transacciones; ¿hemos capturado estos elementos en los casos de uso?

Después de explorar detalladamente los casos de uso para los clientes, el siguiente paso para desarrollar la lista de casos de uso es desarrollar los casos de uso para cada uno de los demás actores. La siguiente lista muestra un primer conjunto razonable de casos de uso para el ejemplo del cajero automático:

- El cliente revisa los balances de la cuenta
- El cliente deposita dinero en su cuenta
- El cliente retira dinero de su cuenta
- El cliente transfiere dinero entre cuentas

- El cliente abre una cuenta
- El cliente cierra una cuenta
- El cliente entra en su cuenta
- El cliente revisa las transacciones recientes
- El empleado del banco entra a un cuenta especial de administración
- El empleado del banco hace un ajuste a la cuenta de un cliente
- Un sistema de soporte para la oficina actualiza la cuenta de un usuario con base en la actividad externa
- Los cambios en la cuenta de un usuario se reflejan en un sistema de apoyo para la oficina
- El cajero automático avisa que no tiene suministro de efectivo
- El técnico del banco llena con efectivo el cajero automático

## Cómo crear el modelo del dominio

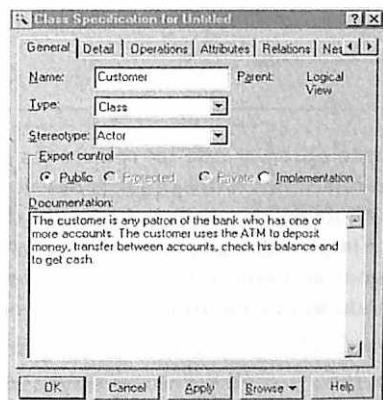
Después de un primer vistazo a sus casos de uso, puede empezar a desarrollar su documento de requerimientos con un modelo detallado del dominio. El *modelo del dominio* es un documento en el que captura todo lo que sabe acerca del dominio (el campo comercial en el que está trabajando). Como parte de su modelo de dominio, debe crear objetos que describan a todos los objetos mencionados en sus casos de uso. Hasta ahora, el ejemplo del cajero automático incluye estos objetos: cliente, personal del banco, sistemas de soporte para las oficinas, cuanta de cheques, cuenta de ahorros, etc.

Para cada uno de estos objetos del dominio, necesitamos capturar datos esenciales, como el nombre del objeto (por ejemplo, cliente, cuenta y así sucesivamente), si el objeto es un actor, el comportamiento y los atributos principales del objeto. Muchas herramientas de modelado soportan la captura de esta información en descripciones de "clases". La figura 18.4 muestra cómo se captura esta información con el programa Rational Rose de Rational Software.

18

**FIGURA 18.4**

*Captura de información con el programa Rational Rose.*



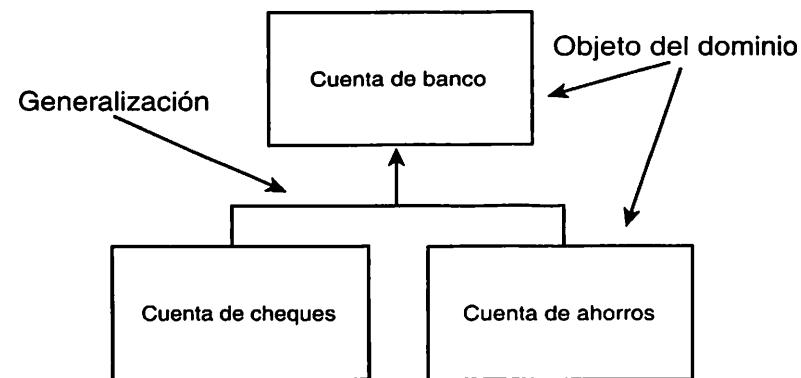
Es importante tener en cuenta que los elementos que estamos describiendo aquí *no son* objetos de diseño, sino objetos del dominio. Ésta es la documentación de la forma en que el mundo funciona, no la documentación de la forma en que nuestro sistema funcionará.

Podemos trazar en un diagrama la relación entre los objetos del dominio del ejemplo del cajero automático por medio del UML, con las mismas convenciones para diagramas que utilizaremos más adelante para describir las relaciones entre clases del dominio. Éste es uno de los puntos fuertes del UML: podemos utilizar las mismas herramientas en cada etapa del proyecto.

Por ejemplo, podemos capturar que las cuentas de cheques y las cuentas de ahorros son especializaciones del concepto más general de cuenta de banco, usando las convenciones del UML para clases y relaciones de generalización, como se muestra en la figura 18.5.

**FIGURA 18.5**

*Especialización.*



En el diagrama de la figura 18.5, las cajas representan los diversos objetos del dominio, y las líneas con punta de flecha indican la generalización. El UML especifica que estas líneas se deben dirigir de la clase *especializada* a la clase “base” más general. Por consiguiente, tanto Cuenta de cheques como Cuenta de ahorros apuntan hacia Cuenta de banco, lo cual indica que cada una es una forma especializada de Cuenta de banco.

### Nota

De nuevo, es importante observar que las relaciones que estamos mostrando en este momento son relaciones entre objetos del dominio. Más adelante, tal vez decida tener un objeto CuentaCheques en su diseño, así como un objeto CuentaBanco, e implementar esta relación por medio de la herencia; pero éstas son decisiones en tiempo de diseño. En tiempo de análisis, todo lo que estamos haciendo es documentar nuestra comprensión de estos objetos del dominio.

El UML es un lenguaje de modelado muy completo, y puede capturar cualquier cantidad de relaciones. Sin embargo, las principales relaciones que se capturan en el análisis son la generalización (o especialización), la contención y la asociación.

### Generalización

A menudo la generalización se confunde con la “herencia”, pero existe una clara y considerable distinción entre las dos. La generalización describe la relación; la herencia es la implementación de la generalización por medio de la programación (es la forma en que manifestamos la generalización en el código).

La generalización implica que el objeto derivado *es* un subtipo del objeto base. Por lo tanto, una cuenta de cheques *es una* cuenta de banco. La relación es simétrica: la cuenta de banco *generaliza* el comportamiento y los atributos comunes de las cuentas de cheques y de ahorros.

Durante el análisis del dominio se busca capturar estas relaciones *de la misma forma en que existen en el mundo real*.

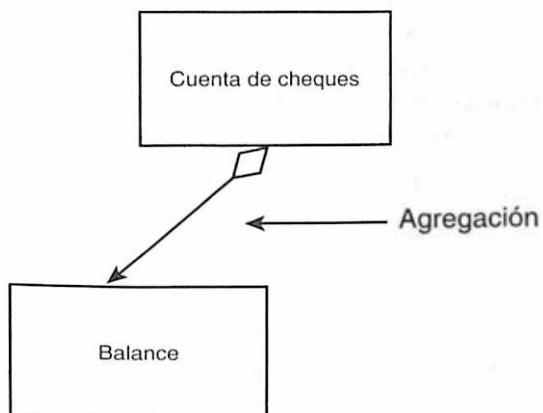
### Contención

Por lo general, un objeto se compone de muchos objetos de otro tipo. Por ejemplo, un auto está compuesto de un volante, llantas, puertas, radio, etc. Una cuenta de cheques se compone de un balance, un historial de transacciones, el número de identificación del cliente, etc. Decimos que la cuenta de cheques *tiene* estos elementos; los modelos de contención tienen la relación *tiene un*. El UML muestra la relación de contención por medio del dibujo de una línea con un rombo que va desde el objeto que contiene hasta el objeto contenido, como se muestra en la figura 18.6.

18

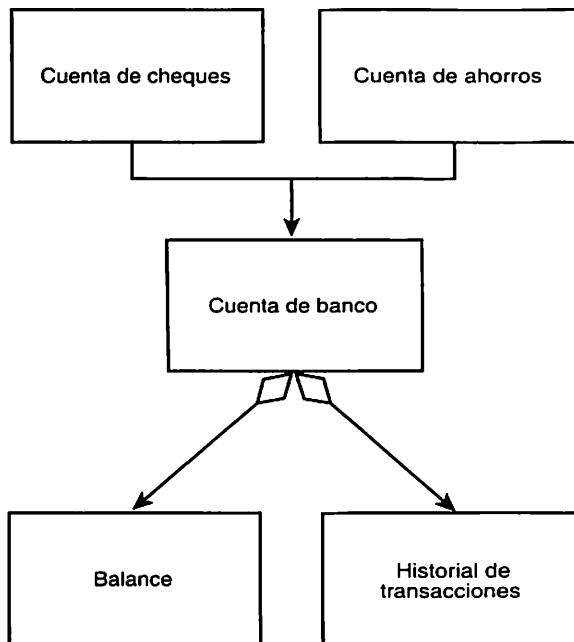
**FIGURA 18.6**

Contención.



El diagrama de la figura 18.6 indica que la Cuenta de cheques *tiene un* Balance. Puede combinar estos diagramas para mostrar un conjunto de relaciones bastante complejo (vea la figura 18.7).

**FIGURA 18.7**  
*Relaciones de objetos.*

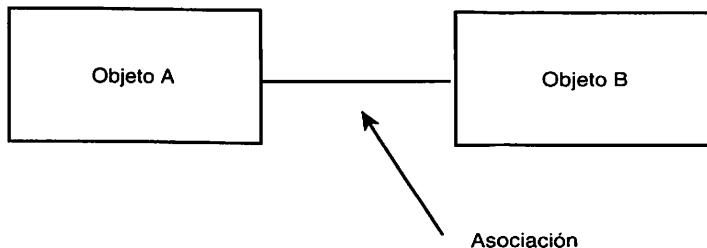


El diagrama de la figura 18.7 establece que una Cuenta de cheques y una Cuenta de ahorros son Cuentas de banco, y que todas las Cuentas de banco tienen un Balance y un Historial de transacciones.

### Asociación

La tercera relación que se captura comúnmente en el análisis del dominio es una asociación simple. Una *asociación* indica que dos objetos se conocen entre sí y que interactúan de alguna forma. Esta definición será mucho más precisa en la etapa de diseño. En la etapa de análisis sólo indicamos que el Objeto A y el Objeto B interactúan, pero que ninguno contiene al otro y ninguno es una especialización del otro. Mostramos esta asociación en el UML por medio de una línea recta simple entre los objetos, como se muestra en la figura 18.8.

**FIGURA 18.8**  
*Asociación.*



El diagrama de la figura 18.8 indica que el Objeto A se asocia de alguna forma con el Objeto B.

### Cómo establecer escenarios

Ahora que tenemos un conjunto preliminar de casos de uso y las herramientas con las que formaremos un diagrama de la relación entre los objetos del dominio, estamos listos para formalizar los casos de uso y darles más profundidad.

Cada caso de uso se puede dividir en una serie de escenarios. Un *escenario* es una descripción de un conjunto específico de circunstancias que se distinguen de entre los diversos elementos del contingente de casos de uso. Por ejemplo, el caso de uso “El cliente retira dinero de su cuenta” podría tener los siguientes escenarios:

- El cliente solicita un retiro de \$300 de la cuenta de cheques y el sistema coloca el dinero en la bandeja de disposición de efectivo e imprime un recibo.
- El cliente solicita un retiro de \$300 de su cuenta de cheques, pero su balance es de \$200. Se informa al cliente que no hay suficientes fondos en la cuenta de cheques para completar el retiro.
- El cliente solicita un retiro de \$300 de su cuenta de cheques, pero hoy ya ha retirado \$100 y el límite es de \$300 por día. Se informa el problema al cliente, y éste opta por retirar sólo \$200.
- El cliente solicita un retiro de \$300 de su cuenta de cheques, pero se acabó el papel para imprimir los recibos. Se informa el problema al cliente, y éste elige proceder sin obtener un recibo.

18

Y así por el estilo. Cada escenario explora una variación en el caso de uso original. A menudo, estas variaciones son condiciones de excepciones (no hay suficiente dinero en la cuenta, no hay suficiente dinero en el cajero, etcétera). Algunas veces, las variaciones exploran matices de decisiones en el caso de uso en sí (por ejemplo, ¿quería el cliente transferir dinero antes de hacer el retiro?).

No se pueden explorar todos los escenarios posibles. Se buscan los escenarios que pongan a prueba los requerimientos del sistema o detalles de la interacción con el actor.

### Cómo establecer lineamientos

Como parte de su metodología, debe crear lineamientos para documentar cada escenario. Estos lineamientos se capturan en el documento de requerimientos. Por lo general, debe asegurarse de que cada escenario incluya lo siguiente:

- Condiciones previas—Qué debe ser cierto para que el escenario comience
- Activadores—Qué hace que el escenario comience

- Qué acciones realizan los actores
- Qué resultados o cambios ocasiona el sistema
- Qué retroalimentación reciben los actores
- Si ocurren o no las actividades repetitivas, y qué ocasiona que concluyan
- Una descripción del flujo lógico del escenario
- Qué hace que el escenario termine
- Condiciones posteriores—Qué debe ser cierto cuando el escenario esté completo

Además, debe nombrar cada caso de uso y cada escenario. Por ejemplo, podría tener la siguiente situación:

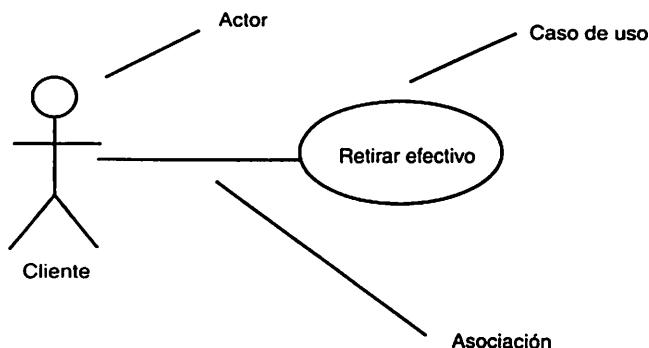
|                                 |                                                                                                                                                                                                                                                                                                                                                                                                                               |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Caso de uso:</b>             | El cliente retira efectivo.                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Escenario:</b>               | El retiro de efectivo de la cuenta de cheques se realiza con éxito.                                                                                                                                                                                                                                                                                                                                                           |
| <b>Condiciones previas:</b>     | El cliente ya está dentro del sistema.                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Activador:</b>               | El cliente solicita un “retiro”.                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Descripción:</b>             | El cliente elige retirar efectivo de una cuenta de cheques. Hay suficiente efectivo en la cuenta, suficiente papel para recibos en el cajero automático y la red está funcionando. El cajero automático pide al cliente que indique la cantidad del retiro, y el cliente pide \$300, una cantidad válida para retirar en este momento. La máquina entrega \$300 e imprime un recibo, y el cliente toma el dinero y el recibo. |
| <b>Condiciones posteriores:</b> | La cuenta del cliente tiene un débito de \$300, y el cliente tiene \$300 en efectivo.                                                                                                                                                                                                                                                                                                                                         |

Este caso de uso se puede mostrar con el diagrama tan increíblemente simple que se muestra en la figura 18.9.

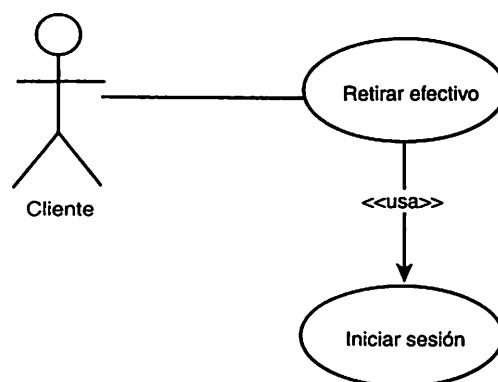
Aquí se captura poca información, exceptuando una abstracción de alto nivel de una interacción entre un actor (el cliente) y el sistema. Este diagrama se vuelve un poco más útil cuando se muestra la interacción entre los casos de uso. Digo que *un poco* más útil debido a que sólo son posibles dos interacciones: “usa” y “extiende”. El estereotipo “usa” indica que un caso de uso es un superconjunto de otro. Por ejemplo, no es posible *retirar efectivo* sin antes *iniciar una sesión*. Podemos mostrar esta relación con el diagrama que se muestra en la figura 18.10.

**FIGURA 18.9**

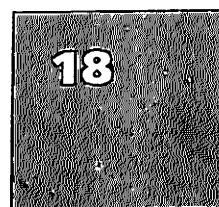
*Diagrama de un caso de uso.*

**FIGURA 18.10**

*El estereotipo "usa".*



La figura 18.10 indica que el caso de uso Retirar efectivo “usa” el caso de uso Iniciar sesión, y por lo tanto implementa completamente a Iniciar sesión como parte de Retirar efectivo.



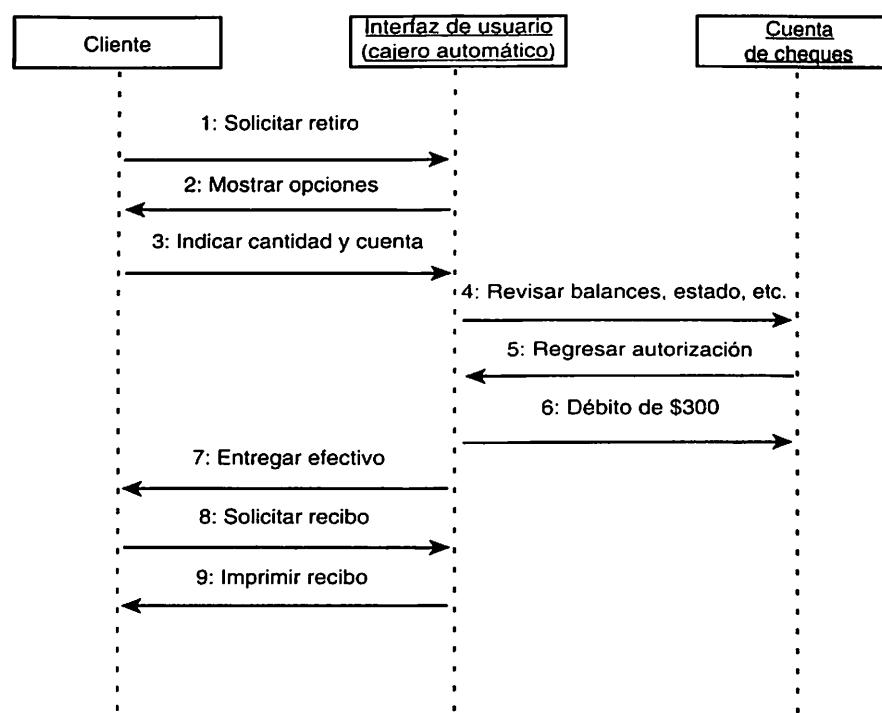
### Tip

El caso de uso “extiende” tiene el propósito de indicar relaciones condicionales y algo semejante a la herencia, pero existe tanta confusión en la comunidad modeladora de objetos en relación con la distinción entre “usa” y “extiende” que muchos desarrolladores simplemente han hecho a un lado a “extiende”, pues sienten que su significado no está muy bien comprendido. Yo utilizo “usa” cuando de otra forma tendría que copiar y pegar el caso de uso completo, y utilizo “extiende” cuando sólo uso el caso bajo ciertas condiciones definibles.

### Diagramas de interacción

Aunque el diagrama de un caso de uso en sí puede ser de valor limitado, puede asociar diagramas con los casos de uso que mejoren en forma dramática la documentación y la comprensión de las interacciones. Por ejemplo, sabemos que el escenario Retirar efectivo representa las interacciones entre los siguientes objetos del dominio: el cliente, la cuenta de cheques y la interfaz de usuario. Podemos documentar esta interacción con un diagrama de interacciones, como se muestra en la figura 18.11.

**FIGURA 18.11**  
*Diagrama de interacciones de UML.*



El diagrama de interacción de la figura 18.11 captura los detalles del escenario que tal vez no sean evidentes al leer el texto. Los objetos que están interactuando son objetos del *dominio*, y tanto el cajero automático como la interfaz de usuario son tratados como un solo objeto, y sólo se pide la cuenta de banco específica para los detalles.

Este ejemplo algo simple del cajero automático muestra sólo un imaginario conjunto de interacciones, pero descubrir todos los detalles específicos de estas interacciones puede ser una herramienta poderosa para comprender tanto el dominio del problema como los requerimientos de su nuevo sistema.

## Cómo crear paquetes

Debido a que se generan muchos casos de uso para cualquier problema que sea considerablemente complejo, el UML le permite agrupar los casos de uso en paquetes.

Un *paquete* es como un directorio o una carpeta; es una colección de objetos de modelado (clases, actores, etcétera). Para manejar la complejidad de los casos de uso, puede crear paquetes agregados por cualquier característica que tenga sentido para su problema. Por ejemplo, puede agregar sus casos de uso por tipo de cuenta (todo lo que afecte a las cuentas de cheques o a las de ahorros), por crédito o débito, por tipo de cliente, o por cualquier característica que tenga sentido para usted. Lo que es más importante, un solo caso de uso puede aparecer en distintos paquetes, lo que le permite una gran flexibilidad en el diseño.

## Análisis de la aplicación

Además de crear casos de uso, el documento de requerimientos capturará las suposiciones, restricciones y requerimientos acerca del hardware y de los sistemas operativos. Los requerimientos de la aplicación son los prerrequisitos *específicos* de su cliente (esas cosas que normalmente se determinan durante el diseño y la implementación, pero que su cliente ha decidido por usted).

Los requerimientos de la aplicación con frecuencia se conducen por la necesidad de tener una interfaz con los sistemas existentes (heredados). En este caso, comprender lo que hacen los sistemas y cómo funcionan es un componente esencial de su análisis.

Lo ideal es que analice el problema, diseñe la solución y luego decida qué plataforma y sistema operativo encajan mejor en el diseño. Ese escenario es tan ideal como raro. Muy a menudo, el cliente tiene una inversión fija en un sistema operativo o en una plataforma de hardware específicos. El plan de negocios del cliente depende de que su software se ejecute en el sistema existente, y usted debe capturar estos requerimientos oportunamente y realizar el diseño de acuerdo con ellos.

## Análisis de los sistemas

Hay software escrito para operar individualmente, que interactúa sólo con el usuario final. Sin embargo, a menudo necesitará tener una interfaz con un sistema operativo. El *análisis de los sistemas* es el proceso de recolectar todos los detalles de los sistemas con los cuales interactuará. ¿Su nuevo sistema será un servidor que proporcione servicios al sistema existente, o será un cliente? ¿Podrá negociar una interfaz entre los sistemas, o deberá adaptarse a un estándar existente? ¿El otro sistema será estable, o deberá estar tirando continuamente a un blanco en movimiento?

Debe contestar en la fase de análisis éstas y otras preguntas relacionadas, antes de que empiece a diseñar su nuevo sistema. Además, será conveniente que trate de capturar las restricciones y limitaciones implícitas en la interacción con los demás sistemas. ¿Disminuirán la sensibilidad de su sistema? ¿Exigirán demasiado de su nuevo sistema, consumiendo recursos y tiempo de cómputo?

## Documentos de planeación

Después de entender lo que debe hacer su sistema y cómo debe comportarse, es hora de empezar a crear un documento que establezca el tiempo y presupuesto para el proyecto. Por lo general, el plazo de tiempo lo establece el cliente: "Tiene 18 meses para terminar esto". Lo ideal es que examine los requerimientos y calcule el tiempo que llevará diseñar e implementar la solución. Eso es lo ideal. La realidad práctica es que la mayoría de los sistemas tiene un límite de tiempo y un límite de costo impuestos, y el verdadero truco es averiguar cuánta de la funcionalidad requerida puede construir en el tiempo y con el costo asignados.

He aquí dos lineamientos que debe considerar cuando va a crear el presupuesto y plazo de tiempo para un proyecto:

- Si se le da un rango, el número externo es probablemente optimista.
- Ley de Hofstadter: Siempre toma más tiempo de lo que usted espera, incluso si toma en cuenta la ley de Hofstadter.

Teniendo en cuenta estas realidades, es imperativo que realice su trabajo por prioridades. Sencillamente, no terminará a tiempo. Es importante que cuando se le agote el tiempo lo que tenga funcione y sea adecuado para una primera liberación. Si está construyendo un puente y se agota el tiempo, si no tuvo oportunidad de terminar el carril para las bicicletas, eso está muy mal; pero de todas formas puede abrir el puente y empezar a cobrar peaje. Si se le acaba el tiempo y apenas va a la mitad del río, eso no es muy bueno.

Una cosa esencial que debe saber acerca de los documentos de planeación es que están mal. En esta etapa tan temprana del proceso, es casi imposible ofrecer una estimación confiable de la duración del proyecto. Después de tener todos los requerimientos, puede darse una buena idea de cuánto tiempo llevará hacer el diseño, una estimación justa de cuánto tardará la implementación, y una estimación razonable del tiempo de prueba. Luego debe dejar por lo menos de un 20 a un 25 por ciento de tiempo de reserva, mismo que puede ir reduciendo a medida que avanza y aprende más.

### Nota

La inclusión de "tiempo de reserva" en su documento de planeación no es una excusa para evitar los documentos de planeación. Es sólo una advertencia para no confiar demasiado en ellos antes de tiempo. A medida que el proyecto avance, usted intensificará su comprensión de cómo funciona el sistema, y sus estimaciones serán cada vez más precisas.

## Visualizaciones

La pieza final del documento de requerimientos es la visualización. Éste es un nombre elegante para los diagramas, las imágenes, imágenes capturadas en pantalla, los prototipos y cualquier otra representación visual creada para ayudarlo a idear y diseñar la interfaz gráfica de usuario de su proyecto.

Para muchos proyectos grandes, puede desarrollar un prototipo completo para ayudarlo (y ayudar a sus clientes) a comprender cómo se comportará el sistema. En algunos equipos de trabajo, el prototipo se convierte en el documento de requerimientos viviente; el sistema “real” se diseña para implementar la funcionalidad diseñada en el prototipo.

## Artefactos

Al final de cada fase de análisis y diseño creará una serie de documentos o “artefactos”. La tabla 18.1 muestra algunos de los artefactos de la fase de análisis. El cliente utiliza estos documentos para asegurarse de que usted comprenda lo que él necesita; también los utilizan los usuarios finales para retroalimentar y orientar el proyecto, y finalmente los utiliza el equipo del proyecto para diseñar e implementar el código. Muchos de estos documentos también proporcionan material de vital importancia tanto para su equipo de documentación como para el equipo de control de calidad, para indicarles cómo debe comportarse el sistema.

18

**TABLA 18.1** Artefactos creados durante la etapa de análisis de un proyecto

| Artefacto                              | Descripción                                                                                                                          |
|----------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| Visualizaciones del caso de uso        | Un documento que detalla los casos de uso, escenarios, estereotipos, condiciones previas, condiciones posteriores y visualizaciones. |
| Análisis del dominio                   | Documento y diagramas que describen las relaciones entre los objetos del dominio.                                                    |
| Diagramas de colaboración de análisis  | Diagramas de colaboración que describen las interacciones entre los objetos del dominio del problema.                                |
| Diagramas de actividad de análisis     | Diagramas de actividad que describen las interacciones entre los objetos del dominio del problema.                                   |
| Análisis de sistemas                   | Reporte y diagramas que describen los sistemas de bajo nivel y de hardware en los que estará el proyecto.                            |
| Documento de análisis de la aplicación | Reporte y diagramas que describen los requerimientos específicos del cliente para este proyecto.                                     |
| Reporte de restricciones operacionales | Reporte que describe las características y limitaciones del rendimiento impuestas por este cliente.                                  |
| Documento de costo y planeación        | Reporte con diagramas de Gantt y Pert que indican el tiempo, avance y costos programados.                                            |

## Diseño

El análisis se enfoca en comprender el dominio del problema, mientras que el diseño se enfoca en crear la solución. El *diseño* es el proceso de transformar la comprensión de los requerimientos en un modelo que se pueda implementar en software. El resultado de este proceso es la producción de un documento de diseño.

El documento de diseño se divide en dos secciones: Diseño de clases y Mecanismos de la arquitectura. A su vez, la sección Diseño de clases se divide en diseño estático (el cual describe detalladamente las diversas clases y sus relaciones y características) y diseño dinámico (el cual describe con detalle la forma en que interactúan las clases).

La sección Mecanismos de la arquitectura proporciona detalles acerca de cómo se va a implementar la persistencia de los objetos, la concurrencia, un sistema de objetos distribuido, etcétera. El resto de esta lección se enfoca en el aspecto del documento de diseño relacionado con el diseño de las clases; otras lecciones del resto de este libro explican cómo implementar varios mecanismos de arquitectura.

### Qué son las clases

Como programador de C++, usted está acostumbrado a crear clases. La metodología del diseño formal requiere que usted separe la clase de C++ de la clase de diseño, aunque estarán íntimamente relacionadas. La clase de C++ que usted escribe en el código es la implementación de la clase que diseñó. Éstas son isomorfas: cada clase de su diseño corresponderá a una clase de su código, pero no las confunda. Ciertamente, es posible implementar sus clases de diseño en otro lenguaje, y la *sintaxis* de las definiciones de la clase podría cambiar.

Dicho esto, la mayor parte del tiempo hablamos sobre estas clases sin hacer distinción entre ellas debido a que las diferencias son altamente abstractas. Cuando usted dice que en su modelo la clase Gato tendrá un método `Maullar()`, debe entender que esto significa que también colocará un método `Maullar()` en su clase de C++.

Las clases del modelo se capturan en diagramas de UML, y las clases de C++ se capturan en código que se puede compilar. La distinción es significativa, aunque sutil.

De cualquier forma, el mayor obstáculo para muchos novatos es encontrar el conjunto inicial de clases y comprender lo que conforma a una clase bien diseñada. Una técnica simplista sugiere escribir los escenarios de los casos de uso y luego crear una clase para cada sustantivo. Considere el siguiente escenario de un caso de uso:

El cliente elige retirar efectivo de su cuenta de cheques. Hay suficiente efectivo en la cuenta, hay suficiente efectivo y recibos en el cajero automático, y la red está funcionando. El cajero automático pide al cliente que indique una cantidad para el retiro, y el cliente pide \$300, una cantidad válida para retirar en este momento. La máquina entrega \$300 e imprime un recibo, y el cliente toma el dinero y el recibo.

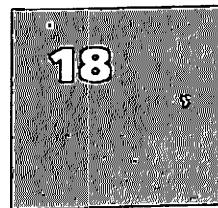
De este escenario, podría obtener las siguientes clases:

- Cliente
- Efectivo
- Cheques
- Cuenta
- Recibos
- CajeroAutomatico
- Red
- Cantidad
- Retiro
- Maquina
- Dinero

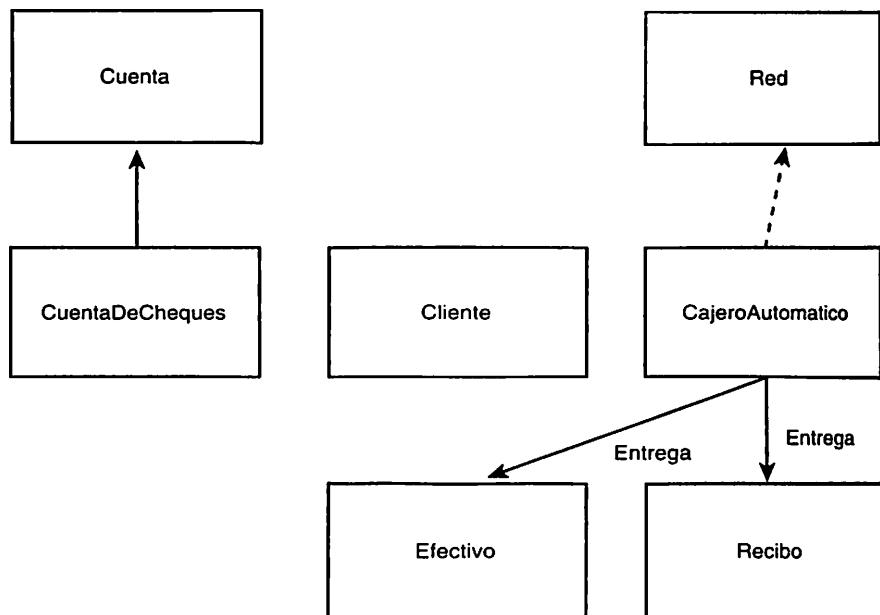
Después podría agregar los sinónimos para crear esta lista, y luego crear clases para cada uno de estos sustantivos:

- Cliente
- Efectivo (dinero, cantidad, retiro)
- Cheques
- Cuenta
- Recibos
- Cajero automático (máquina)
- Red

Ésta no es una mala forma de empezar, hasta donde se puede ver. Podría entonces describir en un diagrama las relaciones obvias entre algunas de estas clases, como se muestra en la figura 18.12.



**FIGURA 18.12**  
*Clases preliminares.*



## Transformaciones

Lo que empezó a hacer en la sección anterior no era tanto extraer los sustantivos del escenario, sino empezar a transformar objetos del análisis del dominio en objetos del diseño. Ése es un excelente primer paso. A menudo, muchos de los objetos del dominio tendrán *sustitutos* en el diseño. Se llama sustituto a un objeto para distinguir entre el verdadero recibo físico entregado por un cajero automático y el objeto de su diseño que es sólo una abstracción intelectual implementada en el código.

Probablemente encontrará que *la mayoría* de los objetos del dominio tiene una representación isomorfa en el diseño, es decir, existe una correspondencia exacta entre el objeto del dominio y el objeto del diseño. Sin embargo, otras veces un solo objeto del dominio se representa en el diseño por medio de toda una serie de objetos de diseño. Y a veces, una serie de objetos del dominio se puede representar por medio de un solo objeto de diseño.

Observe en la figura 18.12 que ya hemos capturado el hecho de que **CuentaDeCheques** es una especialización de **Cuenta**. No salimos a buscar la relación de generalización, pero ésta era evidente, por lo que la capturamos. De la misma manera, gracias al análisis del dominio supimos que **CajeroAutomatico** entrega tanto **Efectivo** como **Recibos**, por lo que inmediatamente capturamos esa información en el diseño.

La relación entre **Cliente** y **CuentaDeCheques** es menos obvia. Sabemos que dicha relación existe, pero los detalles no son obvios, por lo que preferimos esperar.

## Otras transformaciones

Después de haber transformado los objetos del dominio, puede empezar a buscar otros objetos útiles en tiempo de diseño. Un buen lugar para empezar es con las interfaces. Cada interfaz entre su nuevo sistema y los sistemas existentes (heredados) debe estar encapsulada en una clase de interfaz. Si va a interactuar con una base de datos de cualquier tipo, también es un buen candidato para una clase de interfaz.

Estas clases de interfaces ofrecen la encapsulación del protocolo de la interfaz y por ende protegen al código de cambios en el otro sistema. Las clases de interfaces le permiten cambiar su propio diseño, o adecuar cambios en el diseño de otros sistemas sin tener que descomponer el resto del código. Siempre y cuando los dos sistemas sigan soportando la interfaz acordada, pueden moverse independientemente el uno del otro.

## Manipulación de datos

De la misma manera, usted creará las clases para manipulación de datos. Si tiene que transformar datos de un formato a otro (por ejemplo, de Fahrenheit a centígrados o de pulgadas/yardas al sistema métrico), tal vez necesite encapsular estas manipulaciones en una clase de manipulación de datos. Puede utilizar esta técnica al enviar mensajes con datos en los formatos requeridos para otros sistemas o para transmitir por medio de Internet; en resumen, cada vez que tenga que manipular datos en un formato específico, debe encapsular el protocolo en una clase de manipulación de datos.

18

## Vistas

Cada “vista” o “reporte” que genere su sistema (o, si usted genera muchos reportes, cada conjunto de reportes) es un candidato para una clase. Las reglas del reporte (tanto la forma en que se recopila la información como la forma en que se va a desplegar) se pueden encapsular en forma productiva dentro de una clase de vistas.

## Dispositivos

Si su sistema interactúa con dispositivos (como impresoras, módems, escáneres, etc.) o los manipula, los detalles específicos del protocolo del dispositivo deben estar encapsulados en una clase. De nuevo, al crear clases para la interfaz del dispositivo, puede incluir nuevos dispositivos con nuevos protocolos sin tener que descomponer el resto de su código; sólo cree una nueva clase de interfaz que soporte la misma interfaz (o una interfaz derivada), y ya está.

## Modelo estático

Cuando ya tiene establecido su conjunto preliminar de clases, es momento de empezar a modelar sus relaciones e interacciones. Con el fin de tener una mayor claridad, esta lección explica primero el modelo estático y después el modelo dinámico. En el proceso de diseño

real, puede moverse libremente entre los modelos estático y dinámico, proporcionar detalles de ambos y, de hecho, agregar nuevas clases y delinearlas a medida que avanza.

El modelo estático se enfoca en tres áreas de interés: responsabilidades, atributos y relaciones. La más importante de éstas, y en la que se debe enfocar primero, es el conjunto de responsabilidades para cada clase. El principio guía más importante es éste: *cada clase debe ser responsable de una cosa*.

Esto no quiere decir que cada clase tenga sólo un método. Al contrario, muchas clases pueden tener docenas de métodos. Pero todos estos métodos deben ser coherentes y cohesivos; es decir, todos deben relacionarse entre sí y contribuir a la capacidad de la clase para lograr cubrir una sola área de responsabilidad.

En un sistema bien diseñado, cada objeto es una instancia de una clase bien definida y bien comprendida que es responsable de un área de interés. Por lo general, las clases delegan las responsabilidades ajena a otras clases relacionadas. Al crear clases que tengan sólo un área de interés, se promueve la creación de código fácil de mantener.

Para tener una idea de cuáles deben ser las responsabilidades de sus clases, puede ser benéfico que empiece su trabajo de diseño con el uso de tarjetas CRC.

## Tarjetas CRC

CRC significa Clase, Responsabilidad y Colaboración. Una tarjeta CRC no es más que una ficha de  $4 \times 6$ . Este sencillo dispositivo de baja tecnología le permite trabajar con otras personas para entender cuáles son las principales responsabilidades de su conjunto inicial de clases. Debe formar una pila de fichas de  $4 \times 6$  en blanco y reunirse en una mesa de conferencias para tener una serie de sesiones con tarjetas CRC.

### Cómo conducir una sesión CRC

Lo ideal es que a cada sesión CRC asista un grupo de tres a seis personas; más de seis son difíciles de manejar. Debe tener un moderador, cuyo trabajo será evitar que la sesión se salga de su objetivo, así como ayudar a que los participantes comprendan bien lo que se está tratando. Por lo menos debe estar presente un arquitecto de software en jefe, de preferencia alguien con experiencia considerable en el análisis y diseño orientados a objetos. Además, necesitará incluir por lo menos uno o dos “expertos de dominio” que comprendan los requerimientos del sistema y que puedan dar asesoría especializada en relación con la forma en que deben funcionar las cosas.

El ingrediente más esencial en una sesión CRC es la ausencia de gerentes. Ésta es una sesión creativa y libre de preocupaciones que no se debe entorpecer por la necesidad de impresionar a los jefes. El objetivo aquí es explorar, arriesgarse, separar las responsabilidades de las clases y comprender cómo podrían interactuar entre sí.

La sesión CRC se empieza acomodando al grupo alrededor de una mesa para conferencias, con una pequeña pila de fichas de  $4 \times 6$  en blanco. En la parte superior de cada tarjeta CRC escribirá el nombre de una clase individual. Dibuje una línea vertical en el centro de la tarjeta y en la parte izquierda escriba *Responsabilidades*, y en la derecha escriba *Colaboraciones*.

Empiece llenando tarjetas para las clases más importantes que haya identificado. Para cada tarjeta, escriba una definición de una o dos oraciones en la parte posterior. También puede capturar qué otra clase es especializada por esta clase si eso es obvio al momento de estar trabajando con la tarjeta CRC. Sólo escriba *Superclase*: abajo del nombre de la clase y escriba el nombre de la clase de la que se deriva esta clase.

### Enfóquese en las responsabilidades

El objetivo de la sesión CRC es identificar las *responsabilidades* de cada clase. No dé mucha importancia a los atributos, capture sólo los atributos más esenciales y obvios al ir avanzando. El trabajo importante es identificar las responsabilidades. Si, al cumplir con una responsabilidad, la clase debe delegar trabajo a otra clase, debe capturar esa información bajo la sección de colaboraciones.

Vigile sus listas de responsabilidades a medida que vaya progresando. Si se le acaba el espacio en la ficha de  $4 \times 6$ , tal vez tenga sentido que se pregunte si le está pidiendo demasiado a esta clase. Recuerde, cada clase debe ser responsable de un área general de trabajo, y las diversas responsabilidades enlistadas deben ser cohesivas y coherentes, es decir, deben trabajar en conjunto para lograr cumplir la responsabilidad general de la clase.

En este punto *no* debe enfocarse en las relaciones, ni preocuparse por la interfaz de la clase o por cuáles métodos serán públicos y cuáles privados. Sólo debe enfocarse en comprender lo que hace cada clase.

18

### Tarjetas CRC antropomorfas y dirigidas por el caso de uso

La característica clave de las tarjetas CRC es hacerlas *antropomorfas*, es decir, atribuir calidades de tipo humano a cada clase. Esto funciona así: cuando ya tengan un conjunto preliminar de clases, regresen a sus escenarios CRC. Dividan las tarjetas alrededor de la mesa en forma arbitraria, y caminen juntos por el escenario. Por ejemplo, vayan al siguiente escenario de actividades bancarias:

El cliente elige retirar efectivo de la cuenta de cheques. Hay suficiente efectivo en la cuenta, hay suficiente efectivo y recibos en el cajero automático, y la red está funcionando. El cajero automático pide al cliente que indique una cantidad para el retiro, y el cliente pide \$300, una cantidad válida para retirar en este momento. La máquina entrega \$300 e imprime un recibo, y el cliente toma el dinero y el recibo.

Suponga que tenemos cinco participantes en nuestra sesión CRC: Amy, la moderadora y diseñadora orientada a objetos; Barry, el programador en jefe; Charlie, el cliente; Dorris, la experta del dominio; y Ed, un programador.

Amy levanta una tarjeta CRC que representa a `CuentaDeCheques` y dice: "Le digo al cliente cuánto dinero tiene disponible. Él me pide que le dé \$300. Envío un mensaje al distribuidor automático indicándole que dé \$300 en efectivo". Barry levanta su tarjeta y dice: "Yo soy el distribuidor automático. Libero \$300 y envío a Amy un mensaje diciéndole que decremente su balance en \$300. ¿A quién le digo que ahora tengo \$300 menos? ¿Tengo que mantener un registro de eso?". Charlie dice: "Creo que necesitamos un objeto que mantenga el registro del efectivo que hay en la máquina". Ed dice: "No, el distribuidor automático debe saber cuánto efectivo tiene; eso es parte del distribuidor automático". Amy no está de acuerdo y dice: "No, alguien tiene que coordinar la distribución de efectivo. El distribuidor automático necesita saber si hay efectivo disponible y si el cliente tiene suficiente en la cuenta, y tiene que contar el dinero y saber cuándo cerrar la caja. Debe delegar en algún tipo de cuenta interna la responsabilidad de mantener el registro del efectivo disponible. Cualquiera que sepa sobre el efectivo disponible también puede notificar al sistema de apoyo de la oficina cuando sea tiempo de volver a llenar con efectivo el cajero automático. De no ser así, se está pidiendo demasiado al distribuidor automático".

La discusión continúa. Al levantar tarjetas e interactuar entre todos, se descubren los requerimientos y las oportunidades a delegar; cada clase toma vida, y se aclaran sus responsabilidades. Cuando el grupo se atora en las cuestiones de diseño, el moderador puede tomar una decisión y ayudar a que el grupo avance.

### **Limitaciones de las tarjetas CRC**

Aunque las tarjetas CRC pueden ser una herramienta poderosa para empezar el diseño, tienen limitaciones inherentes. El primer problema es que no tienen un escalamiento exitoso. En un proyecto muy complejo, puede agobiarse con las tarjetas CRC; el simple hecho de mantener un registro de todas ellas puede ser difícil.

Las tarjetas CRC tampoco capturan la interrelación entre clases. Aunque es cierto que todas las colaboraciones se toman en cuenta, la naturaleza de la colaboración no se modela bien. Al ver las tarjetas CRC no se puede saber si las clases se agregan unas a otras, quién crea a quién, etc. Las tarjetas CRC tampoco capturan los atributos, por lo que es difícil pasar de las tarjetas CRC al código. Lo que es más importante, las tarjetas CRC son estáticas; aunque se pueden representar las interacciones entre clases, las tarjetas CRC no capturan esta información por sí mismas.

En resumen, las tarjetas CRC son un buen inicio, pero necesita mover las clases hacia el UML si va a crear un modelo robusto y completo de su diseño. Aunque la transición *hacia* el UML no es muy difícil, es de un solo sentido. Después de mover las clases hacia los diagramas del UML, ya no hay regreso; deja a un lado las tarjetas CRC y ya no regresa a ellas. Simplemente es demasiado difícil mantener los dos modelos sincronizados entre sí.

### Cómo transformar las tarjetas CRC en UML

Cada tarjeta CRC puede convertirse directamente en una clase modelada con UML. Las responsabilidades se convierten en métodos de la clase, y los atributos que se tengan capturados también se agregan. La definición de la clase de la parte trasera de la tarjeta se coloca en la documentación de la clase. La figura 18.13 muestra la relación existente entre la tarjeta CRC de CuentaDeCheques y la clase en UML creada a partir de esa tarjeta.

**Clase:** CuentaDeCheques

**Superclase:** Cuenta

**Responsabilidades:**

Registrar el balance actual

Aceptar depósitos y transferencias

Llenar cheques

Entregar efectivo

Mantener el balance diario de retiros del cajero automático

**Colaboraciones:**

Otras cuentas

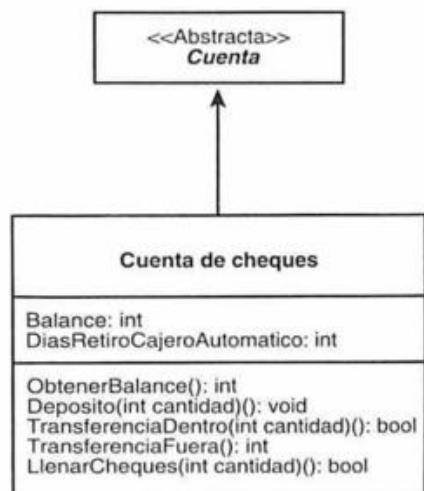
Sistemas de apoyo para las oficinas

Distribuidor automático de efectivo

18

**FIGURA 18.13**

Tarjeta CRC.



## Relaciones entre clases

Una vez que las clases estén en el UML, puede empezar a centrar su atención en las relaciones entre las diversas clases. Las principales relaciones que debe modelar son las siguientes:

- Generalización
- Asociación
- Agregación
- Composición

Éstas son relaciones en el modelo del dominio que se llevan (traducen) a las clases de C++.

La relación de generalización se implementa en C++ mediante la herencia pública. Sin embargo, desde la perspectiva del diseño, nos enfocamos menos en el mecanismo y más en la semántica: qué es lo que implica esta relación.

Examinamos la relación de generalización en la fase de análisis, pero ahora centramos nuestra atención no sólo en los objetos del dominio, sino también en los objetos de nuestro diseño. Ahora nuestros esfuerzos se concentran en “factorizar” la funcionalidad común de las clases relacionadas en clases base que puedan encapsular las responsabilidades compartidas.

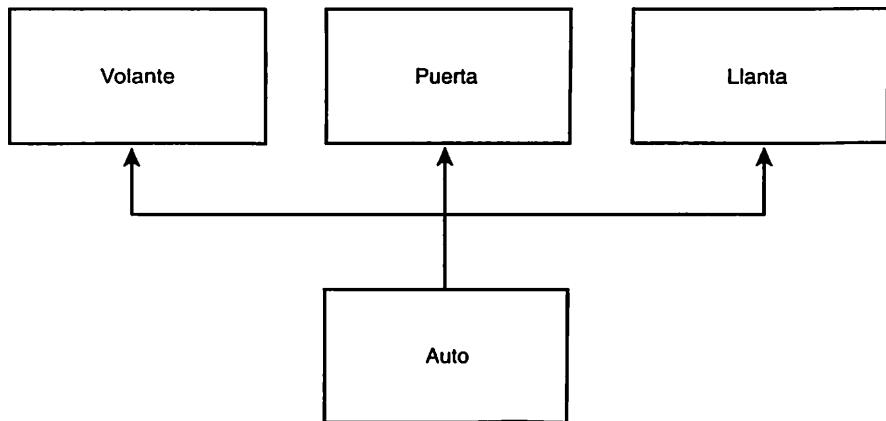
Cuando se “factoriza” la funcionalidad común, esa funcionalidad se saca de las clases especializadas y se sube hacia la clase más general. Por lo tanto, si descubre que tanto su cuenta de cheques como la de ahorros necesitan métodos para transferir dinero a otras cuentas y de otras cuentas, puede subir el método `TransferirFondos()` hacia la clase base `Cuenta`. Entre más factorice las clases derivadas, su diseño será más polimorfo.

Una de las capacidades disponibles en C++, que no está disponible en Java, es la *herencia múltiple* (aunque Java tiene una capacidad similar, aunque limitada, con sus *interfaces múltiples*). La herencia múltiple permite que una clase herede de más de una clase base, recibiendo los miembros y métodos de dos o más clases.

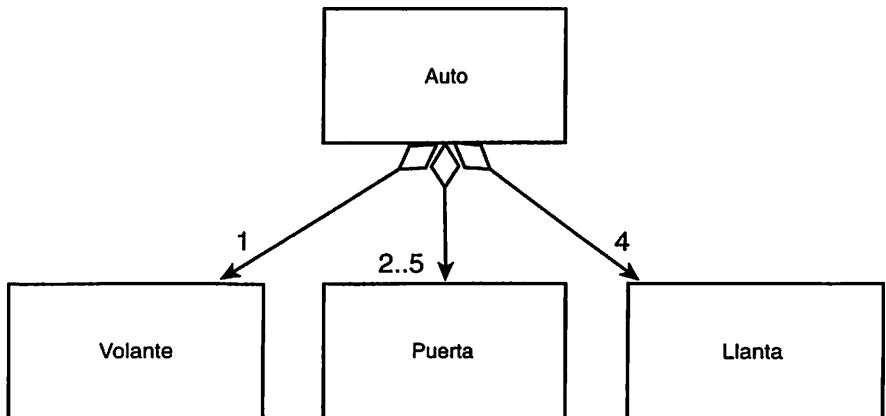
La experiencia ha demostrado que se debe usar la herencia múltiple con cautela ya que puede complicar tanto el diseño como la implementación. Muchos problemas que se solucionaban inicialmente con herencia múltiple ahora se solucionan por medio de la agregación. Dicho esto, la herencia múltiple es una herramienta poderosa, y su diseño tal vez requiera que una sola clase generalice el comportamiento de dos o más clases distintas.

### Herencia múltiple en comparación con la contención

¿Es un objeto la suma de sus partes? ¿Tiene sentido modelar un objeto `Auto` como una especialización de `Volante`, `Puerta` y `Llanta`, como se muestra en la figura 18.14?

**FIGURA 18.14***Herencia falsa.*

Es importante regresar a los fundamentos: la herencia pública siempre debe modelar la generalización. La expresión común para la relación generalización-especialización es que la herencia debe modelar relaciones de tipo *es un*. Si quiere modelar la relación de tipo *tiene un* (por ejemplo, un auto tiene un volante), debe hacerlo con la agregación, como se muestra en la figura 18.15.

**FIGURA 18.15***Agregación.*

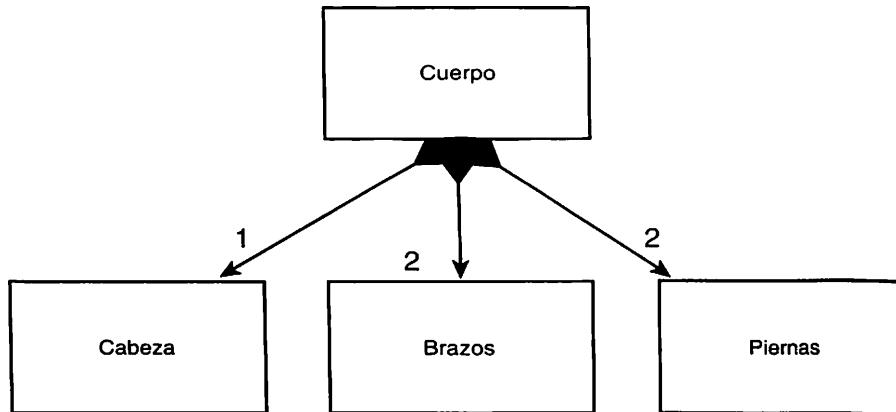
18

El diagrama de la figura 18.15 indica que un auto tiene un volante, cuatro ruedas y de dos a cinco puertas. Éste es un modelo más preciso de la relación entre un auto y sus partes. Observe que el rombo del diagrama no está relleno; esto se debe a que estamos modelando esta relación como una agregación, no como una composición. La composición implica el control del tiempo de vida del objeto. Aunque el auto *tiene* llantas y una puerta, las llantas y la puerta pueden existir antes de ser parte del auto y pueden seguir existiendo cuando ya no sean parte del auto.

La figura 18.16 modela la composición. Este modelo dice que el cuerpo no es sólo una agregación de una cabeza, dos brazos y dos piernas, sino que estos objetos (cabeza, brazos y piernas) se crean cuando el cuerpo se crea, y desaparecen cuando el cuerpo desaparece. Es decir, no tienen existencia independiente; el cuerpo se compone de estas cosas y sus tiempos de vida están entrelazados.

**FIGURA 18.16**

*Composición.*

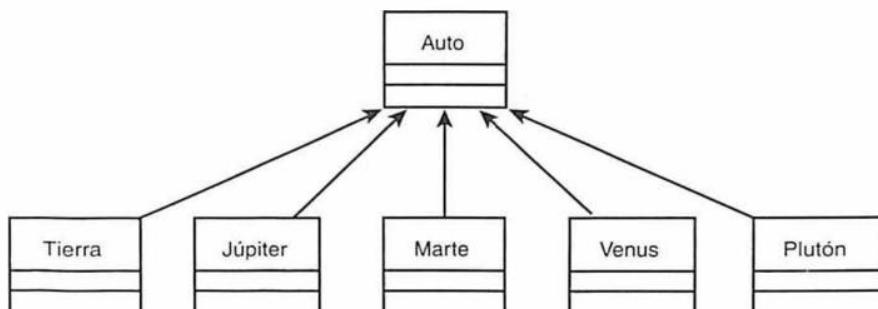
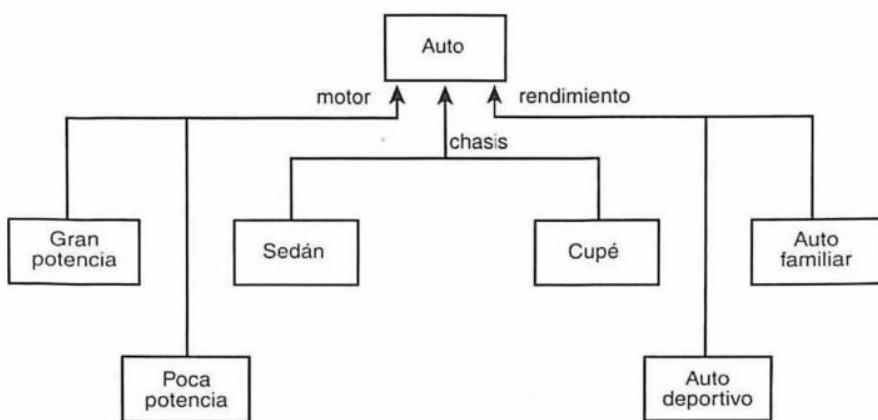


### Discriminadores y tipos de poder

¿Cómo podría diseñar las clases requeridas para reflejar las diversas líneas de modelos de un fabricante típico de autos? Suponga que se le contrata para diseñar un sistema para Acme Motors, que actualmente fabrica cinco autos: el Plutón (un auto compacto lento con motor pequeño); el Venus (un sedán de cuatro puertas con un motor de tamaño mediano); el Marte (un auto deportivo con el motor más grande de la compañía, diseñado para un rendimiento máximo); el Júpiter (una mini vagoneta con el mismo motor que el auto deportivo, pero diseñada para cambiar velocidades a menos revoluciones por minuto y utilizar su poder para mover su mayor peso); y la Tierra (una vagoneta con motor pequeño, pero de gran potencia).

Podría empezar por crear subtipos del auto que reflejen los diversos modelos, y luego crear instancias de cada modelo a medida que van saliendo por la línea de ensamblaje, como se muestra en la figura 18.17.

¿Cómo se diferencian estos modelos? Como vio, se diferencian por el tamaño del motor, el tipo de chasis y las características de rendimiento. Estas diversas características discriminatorias se pueden mezclar y relacionar para crear varios modelos. Podemos modelar esto en UML con el estereotipo *discriminador*, como se muestra en la figura 18.18.

**FIGURA 18.17***Modelado de subtipos.***FIGURA 18.18***Modelado del discriminador.*

18

El diagrama de la figura 18.18 indica que las clases se pueden derivar de *Auto* con base en la mezcla y la relación de los tres atributos discriminatorios. El tamaño del motor estipula qué tan poderoso es el auto, y las características de rendimiento indican qué tan deportivo es. Por lo tanto, puede tener una vagoneta deportiva potente, un sedán familiar de poca potencia, y así por el estilo.

Cada atributo se puede implementar con un simple enumerador. Por ejemplo, el tipo de chasis se podría implementar en el código con la siguiente instrucción:

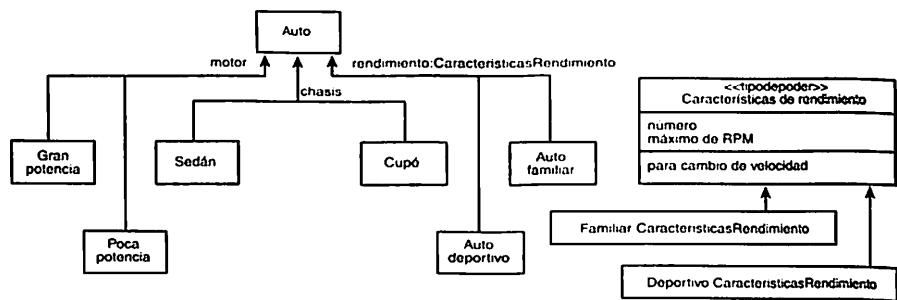
```
enum TipoChasis = { sedan, deportivo, minivan, vagoneta };
```

Sin embargo, puede pasar que un simple valor sea insuficiente para modelar un discriminador específico. Por ejemplo, la característica de rendimiento puede ser bastante compleja. En este caso se puede modelar el discriminador como una clase, y la discriminación se puede encapsular en una instancia de ese tipo.

Por ejemplo, el auto podría modelar las características de rendimiento en un tipo llamado *rendimiento*, el cual contiene información acerca de cuándo cambia velocidades el motor y qué tan rápido puede girar. El estereotipo de UML para una clase que encapsula a un discriminador, y que se puede utilizar para crear *instancias* de una clase (*Auto*) que sean lógicamente de tipos distintos (por ejemplo, *AutoDeportivo* comparado con *AutoFamiliar*) es "tipo de poder". En este caso, la clase *Rendimiento* es un tipo de poder para auto. Al instanciar a *Auto*, también se crea una instancia de un objeto *Rendimiento*, y se asocia a un objeto *Rendimiento* dado con un *Auto* dado, como se muestra en la figura 18.19.

**FIGURA 18.19**

*Un discriminador como tipo de poder.*

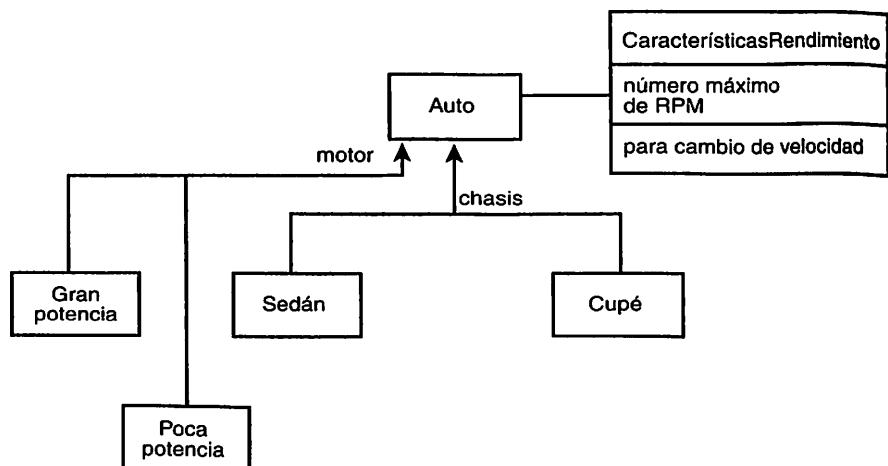


Los tipos de poder le permiten crear una variedad de tipos lógicos sin utilizar herencia. Gracias a esto puede manejar un conjunto grande y complejo de tipos sin la explosión combinatoria que podría encontrarse con la herencia.

Por lo general, el tipo de poder se *implementa* en C++ con apuntadores. En este caso, la clase *Auto* contiene un apuntador a una instancia de la clase *CaracterísticasRendimiento* (vea la figura 18.20). Dejo como ejercicio para el lector ambicioso convertir los discriminadores de chasis y de motor en tipos de poder.

**FIGURA 18.20**

*La relación entre un objeto Auto y su tipo de poder.*



```
Class Auto : public Vehiculo
{
public:
    Auto();
    ~Auto();
    // otros métodos públicos suprimidos
private:
    CaracteristicasRendimiento * apRendimiento;
};
```

Como observación final, los tipos de poder le permiten crear nuevos *tipos* (no sólo instancias) en tiempo de ejecución. Debido a que cada tipo lógico se diferencia sólo por los atributos del tipo de poder asociado, estos atributos pueden ser parámetros para el constructor del tipo de poder. Esto significa que usted puede, en tiempo de ejecución, crear nuevos tipos de autos al instante. Es decir, al pasar diferentes tamaños de motor y revoluciones para cambio de velocidad al tipo de poder, puede crear efectivamente nuevas características de rendimiento. Al asignar esas características a varios autos, puede engrandecer impresionantemente el conjunto de tipos de autos *en tiempo de ejecución*.

## Modelo dinámico

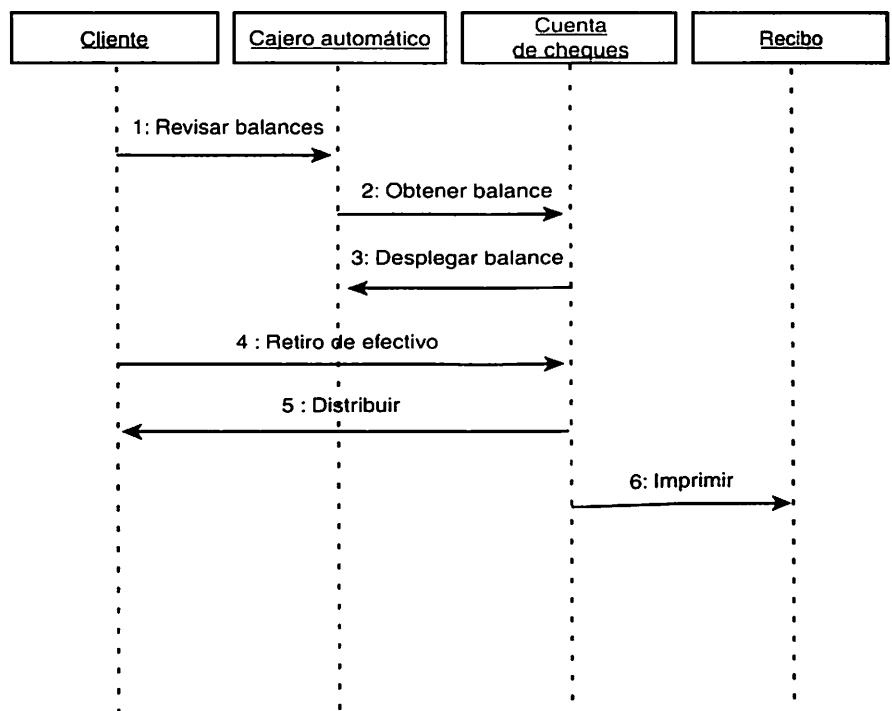
Además de modelar las relaciones entre las clases, es muy importante modelar la forma en que interactúan. Por ejemplo, volviendo al escenario de las actividades bancarias, las clases CuentaDeCheques, CajeroAutomatico y Recibo pueden interactuar con el Cliente para completar el caso de uso “Retiro de efectivo”. Regresemos a los diagramas de secuencia utilizados inicialmente en el análisis, pero ahora desarrollemos los detalles con base en los métodos que desarrollamos en las clases, como se muestra en la figura 18.21.

Este simple diagrama de interacción muestra la interacción entre un número de clases de diseño con el paso del tiempo. Sugiere que la clase CajeroAutomatico delegue en la clase CuentaDeCheques toda la responsabilidad de manejar el balance, mientras que CuentaDeCheques se apoyará en la clase CajeroAutomatico para manejar el despliegue de información para el usuario.

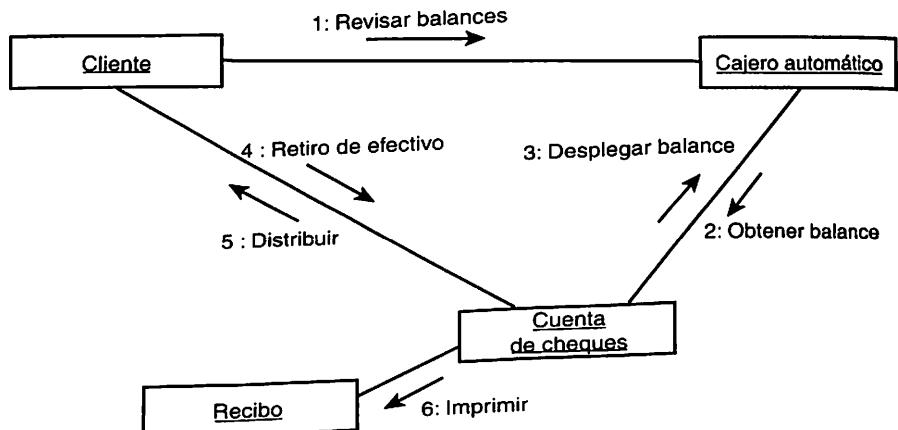
Los diagramas de interacción pueden ser de dos tipos. El de la figura 18.21 se llama *diagrama de secuencia*. Se proporciona otra vista de la misma información por medio del *diagrama de colaboración*. El diagrama de secuencia enfatiza la secuencia de eventos con el paso del tiempo; el diagrama de colaboración enfatiza las interacciones entre las clases. Puede generar un diagrama de colaboración directamente de un diagrama de secuencia; las herramientas como el programa Rational Rose automatizan esta tarea con el clic de un botón (vea la figura 18.22).

**FIGURA 18.21**

*Diagrama de secuencia.*

**FIGURA 18.22**

*Diagrama de colaboración.*

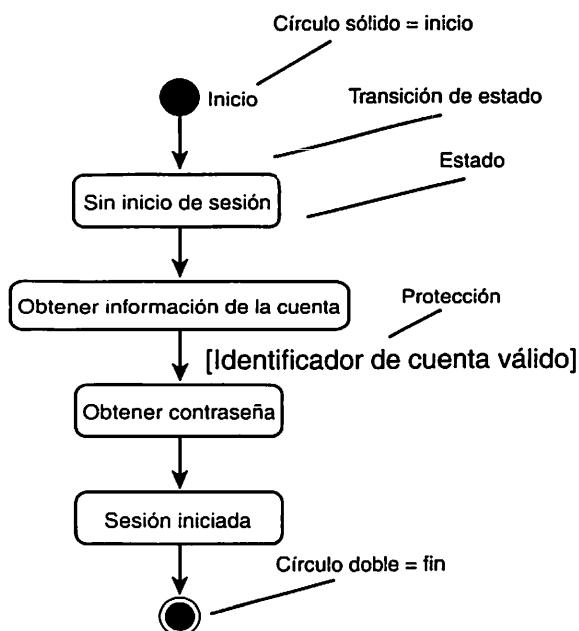


### Diagramas de transición de estado

Para llegar a entender las interacciones entre los objetos, tenemos que entender los diversos *estados* posibles de cada objeto individual. Podemos modelar las transiciones entre los diversos estados en un diagrama de estado (o diagrama de transición de estado). La figura 18.23 muestra los diversos estados de la clase CuentaCliente cuando el cliente entra al sistema.

**FIGURA 18.23**

*Estado de la cuenta del cliente.*



Cada diagrama de estado empieza con un solo estado **inicio** y termina con cero o más estados **finales**. Los estados individuales tienen nombre, y las transiciones se pueden etiquetar. **protección** indica una condición que se debe satisfacer para que un objeto pase de un estado a otro.

18

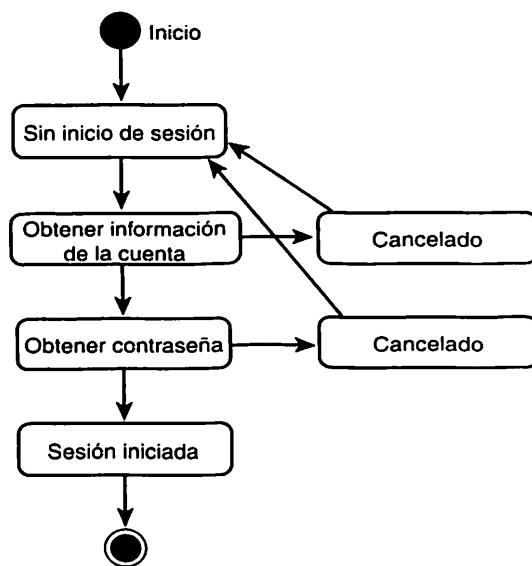
### Superestados

El cliente puede cambiar de opinión en cualquier momento y decidir no entrar al sistema. Puede hacerlo después de introducir su tarjeta para identificar su cuenta, o después de escribir su contraseña. En cualquier caso, el sistema debe aceptar su petición de cancelar y regresar al estado "Sin inicio de sesión" (ver la figura 18.24).

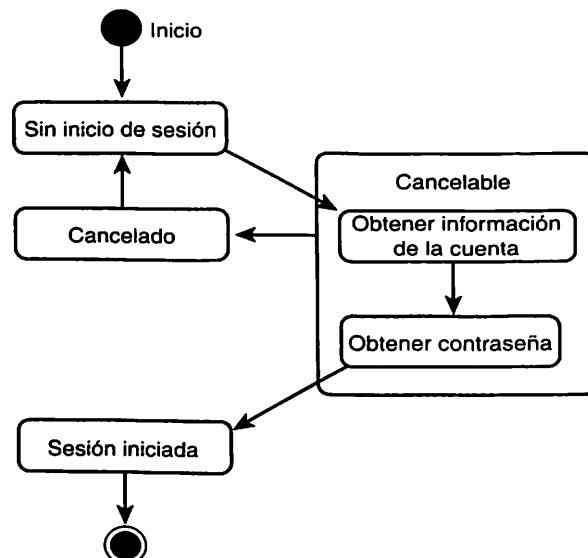
Como puede ver, en un diagrama más complicado el estado **Cancelado** se convertirá rápidamente en una distracción. Esto es muy molesto ya que la cancelación es una condición excepcional que no debe tener importancia en el diagrama. Puede simplificar este diagrama utilizando un superestado, como se muestra en la figura 18.25.

**FIGURA 18.24**

*El usuario puede cancelar.*

**FIGURA 18.25**

*Superestado.*



El diagrama de la figura 18.25 proporciona la misma información que la figura 18.24, pero es mucho más limpio y fácil de leer. Desde el momento en que usted inicia la sesión hasta que el sistema termina de iniciarla, puede cancelar el proceso. Si cancela, regresa al estado "Sin inicio de sesión".

## No se apresure a llegar al código

Uno de los mayores problemas en las organizaciones de desarrollo de software es su prisa por empezar a codificar. Siempre hay presiones de tiempo, y las porciones de codificación/prueba del proyecto tienden a ser la mayor parte. Y por lo general se asignan varias personas a la tarea.

Como resultado, las compañías por lo general se apresuran o escatiman en la fase de análisis para crear la apariencia de estar progresando y mantener a los programadores ocupados. Desgraciadamente, esto produce una “solución” que no resuelve el problema esencial. Este hecho se descubre cuando el código está a punto de ser terminado. Debido a esto, los programadores tienen que regresar y arreglar las diferencias, y el resultado es que trabajan apurados, hay errores en el software, los tiempos programados para la entrega del software no se cumplen y se tienen soluciones incompletas.

Trate de resistir las presiones organizacionales. Trabaje para tener un análisis completo antes de empezar la codificación. De ser necesario, puede presentar información relacionada con proyectos que fallaron por codificar antes de tiempo.

Claro que ésta no es una excusa para permitir una “parálisis en el análisis” en la que el análisis continúa eternamente sin resultados ni progreso. Hay una línea delgada que establece la distinción entre un análisis completo y uno que se va a paralizar. Por desgracia, la experiencia es con lo único que se puede determinar la diferencia.

Existe el concepto “lo suficientemente bueno”. Es muy difícil hacer algo verdaderamente “completo”. Hay una sentencia en esta industria que establece que el 80% de los resultados ocupa un 20% del tiempo y el 20% restante ocupa el 80% del tiempo. En algún momento debe tomar la decisión de avanzar al siguiente paso, o empezará a sufrir lo que se conoce como parálisis en el análisis (nunca llegar al producto final porque está tratando de obtener el modelo perfecto).

Por desgracia, el mejor momento para determinar si el análisis y el diseño son “lo suficientemente buenos” es una vez que termina el proyecto. Si el código que entrega cumple con los requerimientos organizacionales, entonces fue lo suficientemente bueno. Si faltan piezas o características, esto tiende a indicar que no invirtió tiempo ni esfuerzo suficientes. Si hay características que nadie quiere o nadie sabe para qué son, eso indica que invirtió demasiado tiempo en el diseño.

Sólo mirando hacia atrás y aprendiendo de los resultados podrá saber si el esfuerzo invertido en un diseño es lo suficientemente bueno.

¿Qué tan bueno o completo tiene que ser su análisis? ¿Qué tanto es lo suficientemente bueno? Sólo usted y su organización pueden decidir esto. Pero ¡piénselo! Hay demasiadas organizaciones que no piensan sobre el proceso y continuamente entregan software que se pasó del tiempo programado, del presupuesto o que no tiene las características necesarias.

## Resumen

Esta lección le proporcionó una introducción a las cuestiones relacionadas con el análisis y el diseño orientados a objetos. La esencia de este método es analizar la forma en que se utilizará su sistema (casos de uso) y cómo debe funcionar, y luego diseñar las clases y modelar sus relaciones e interacciones.

En el pasado bosquejábamos una idea de lo que queríamos lograr y empezábamos a escribir el código. El problema es que los proyectos complejos nunca se terminan; y en caso de terminarse, son poco confiables y frágiles. Al dedicarnos a comprender los requerimientos y a modelar el diseño, aseguramos que el producto terminado esté correcto (es decir, que cumpla con el diseño) y que sea robusto, confiable y extensible.

Las cuestiones relacionadas con la prueba y la distribución están más allá del alcance de este libro, por lo que sólo queda mencionar que debe planear la prueba de sus unidades a medida que implementa, y que utilizará el documento de requerimientos como la base de su plan de prueba previo a la distribución.

## Preguntas y respuestas

- P ¿En qué forma el análisis y el diseño orientados a objetos son fundamentalmente distintos de otros métodos?**
- R** Antes del desarrollo de estas técnicas orientadas a objetos, los analistas y los programadores pensaban en los programas como grupos de funciones que actuaban sobre los datos. La programación orientada a objetos se enfoca en los datos y la funcionalidad integrados como unidades discretas que tienen tanto conocimiento (datos) como capacidades (funciones). Por otro lado, los programas procedurales se enfocan en las funciones y la forma en que actúan sobre los datos. Se ha dicho que los programas escritos en Pascal y en C son colecciones de procedimientos, y los programas de C++ son colecciones de clases.
- P ¿Es la programación orientada a objetos la bala de plata que resolverá todos los problemas de programación?**
- R** No, nunca se planeó que fuera así. Sin embargo, para problemas grandes y complejos, el análisis, el diseño y la programación orientados a objetos pueden proporcionar herramientas al programador para que pueda manejar una enorme complejidad en formas que anteriormente eran imposibles.
- P ¿Es C++ el lenguaje orientado a objetos perfecto?**
- R** C++ tiene varias ventajas y desventajas al compararlo con otros lenguajes alternativos de programación orientada a objetos, pero tiene una ventaja inigualable sobre todos los demás: es el lenguaje de programación orientada a objetos más popular en todo el mundo. Francamente, la mayoría de los programadores no decide

programar en C++ después de analizar exhaustivamente los lenguajes alternativos de programación orientada a objetos; se van hacia donde está la acción, pero ahora la acción está en C++. Hay buenos motivos para esto; C++ tiene mucho que ofrecer, pero este libro existe debido a que C++ es el lenguaje de desarrollo preferido en muchas empresas, y Linux es un área que está creciendo mucho.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Cuál es la diferencia entre programación orientada a objetos y programación procedural?
2. ¿Cuáles son las fases del análisis y del diseño orientados a objetos?
3. ¿Cómo se relacionan los diagramas de secuencia y los diagramas de colaboración?

### Ejercicios

1. Suponga que tiene que simular la intersección de la avenida Massachusetts con la calle Vassar (dos caminos típicos de dos carriles, con semáforos y cruce de peatones). El propósito de la simulación es determinar si la sincronización del semáforo permite un flujo continuo de tráfico.

¿Qué tipos de objetos debe modelar en la simulación? ¿Cuáles serían las clases para la simulación?

2. Suponga que la intersección del ejercicio 1 está en un suburbio de Boston, que sin duda tiene las calles menos amigables de todo Estados Unidos. A cualquier hora hay tres tipos de conductores en Boston:

Los locales, quienes siguen conduciendo por las intersecciones aunque el semáforo esté en rojo; los turistas, que manejan lenta y cautelosamente (por lo general, en un auto rentado); y los taxistas, que tienen una amplia variedad de patrones de manejo, dependiendo de los tipos de pasajeros que lleven.

Además, Boston tiene dos tipos de peatones: los locales, que cruzan la calle cuando les da la gana y raras veces utilizan las áreas para cruce de peatones; y los turistas, quienes siempre utilizan las áreas para cruce de peatones y cruzan sólo cuando el semáforo lo permite.

18

Finalmente, Boston tiene ciclistas que nunca ponen atención a las señales de alto.  
¿Cómo cambian el modelo estas consideraciones?

3. Diseñe un programador de grupos. Este software le permite programar juntas entre individuos o grupos y reservar un número limitado de salones para conferencias. Identifique los subsistemas principales.
4. Diseñe y muestre las interfaces para las clases del módulo de reservación de salones del programa que se describe en el ejercicio 3.

# SEMANA 3

Día 19

## Plantillas

Una nueva y poderosa herramienta para los programadores de C++ son los “tipos parametrizados” o plantillas. Las plantillas son tan útiles que en la definición del lenguaje C++ se ha adoptado la STL (Biblioteca Estándar de Plantillas).

Hoy aprenderá lo siguiente:

- Qué son las plantillas y cómo utilizarlas
- Cómo crear plantillas de clases
- Cómo crear plantillas de funciones
- Qué es la Biblioteca Estándar de Plantillas y cómo utilizarla

### Qué son las plantillas

Al final de la semana 2 vio cómo crear un objeto llamado `ListaPiezas` y cómo utilizarlo para crear un `CatalogoPiezas`. Si se quiere basar en el objeto `ListaPiezas` para crear una lista de gatos, hay un problema: `ListaPiezas` sólo conoce piezas.

Para solucionar este problema, puede crear una clase base llamada `Lista` y de ahí derivar las clases `ListaPiezas` y `ListaGatos`. Luego podría cortar y pegar la mayor parte de la clase `ListaPiezas` en la nueva declaración de `ListaGatos`. La siguiente semana, cuando quiera crear una lista de objetos llamados `Auto`, tendrá que crear una nueva clase, y de nuevo usar los métodos cortar y pegar.

Sin necesidad de decirlo, ésta no es una solución satisfactoria. Con el tiempo, tendrá que extender la clase `Lista` y sus clases derivadas. Asegurarse de que todos los cambios se propaguen a todas las clases relacionadas sería una pesadilla.

Las plantillas resuelven este problema, y con la adopción del estándar ANSI son una parte integral del lenguaje. Como todo lo que hay en C++, tienen seguridad de tipos y son muy flexibles.

## Tipos parametrizados

Las plantillas le enseñan al compilador cómo crear una lista de cualquier tipo de objeto, en lugar de crear un conjunto de listas de tipos específicos (una `ListaPiezas` es una lista de piezas, una `ListaGatos` es una lista de gatos. En lo único que son diferentes es en el tipo del objeto que hay en la lista). Con las plantillas, el tipo de objeto de la lista se convierte en un parámetro para la definición de la clase.

Un componente común de casi todas las bibliotecas de C++ es una clase de arreglos. Como vio con las clases `Lista`, es tedioso e ineficiente crear una clase de arreglos para enteros, otra para dobles, y otra más para un arreglo de objetos `Animal`. Las plantillas le permiten declarar una clase de arreglos parametrizada y luego especificar qué tipo de objeto guardará cada instancia del arreglo. Hay que tener en cuenta que la STL (Biblioteca Estándar de Plantillas) proporciona un conjunto estandarizado de clases *contenedoras*, incluyendo arreglos, listas, etc. En esta lección verá lo que necesita para hacer sus propias plantillas y para entender perfectamente su funcionamiento; sin embargo, en un programa comercial, muy probablemente utilice las clases de la STL en lugar de las suyas.

### Cómo crear una instancia a partir de una plantilla

La *instanciación* es el acto de crear un tipo específico a partir de una plantilla. Las clases individuales se llaman *instancias* de la plantilla.

Las *plantillas parametrizadas* le proporcionan la capacidad de crear una clase general y pasar tipos como parámetros a esa clase para crear instancias específicas.

## Definición de una plantilla

Para declarar un objeto parametrizado `Arreglo` (una plantilla para un arreglo) se escribe

```
1: template <class T>      // declarar la plantilla y el parámetro
2: class Arreglo           // la clase que se va a parametrizar
3: {
```

```
4:     public:  
5:         Arreglo();  
6:     // aquí va la declaración completa de la clase  
7: };
```

La palabra reservada `template` se utiliza al principio de cualquier declaración y definición de una clase de una plantilla. Los parámetros de la plantilla van después de la palabra reservada `template`. Los parámetros son las cosas que cambiarán con cada instancia. Por ejemplo, en la plantilla del arreglo mostrado anteriormente va a cambiar el tipo de los objetos guardados en el arreglo. Una instancia podría guardar un arreglo de enteros, y otra podría guardar un arreglo de objetos `Animal`.

En este ejemplo se utiliza la palabra reservada `class` seguida del identificador `T`. La palabra reservada `class` indica que este parámetro es un tipo. El identificador `T` se utiliza en el resto de la definición de la plantilla para referirse al tipo parametrizado. Una instancia de esta clase substituirá a `T` con `int` en cualquier parte en que aparezca, y otra lo substituirá con `Gato`.

Para declarar una instancia `int` y una instancia `Gato` de la clase parametrizada `Arreglo`, se escribiría lo siguiente:

```
Arreglo<int> unArregloInt;  
Arreglo<Gato> unArregloGato;
```

El objeto `unArregloInt` es del tipo *arreglo de enteros*; el objeto `unArregloGato` es del tipo *arreglo de gatos*. Ahora puede utilizar el tipo `Arreglo<int>` en cualquier parte en que normalmente utilizaría un tipo (como en el valor de retorno de una función, como un parámetro para una función, etcétera). El listado 19.1 muestra la declaración completa de esta plantilla `Arreglo` simplificada.

### Nota

¡El listado 19.1 no es un programa completo y no compilará!

19

### ENTRADA

#### LISTADO 19.1 Una plantilla para una clase llamada `Arreglo`

```
1: #include <iostream.h>  
2:  
3: //Listado 19.1 La plantilla de una clase de tipo arreglo  
4:  
5: const int TamanioPredet = 10;  
6:  
7: // declarar la plantilla y el parámetro  
8: template < class T >  
9: // la clase que se va a parametrizar  
10: class Arreglo  
11: {  
12:     public:  
13:         // constructores
```

continúa

Para solucionar este problema, puede crear una clase base llamada `Lista` y de ahí derivar las clases `ListaPiezas` y `ListaGatos`. Luego podría cortar y pegar la mayor parte de la clase `ListaPiezas` en la nueva declaración de `ListaGatos`. La siguiente semana, cuando quiera crear una lista de objetos llamados `Auto`, tendrá que crear una nueva clase, y de nuevo usar los métodos cortar y pegar.

Sin necesidad de decirlo, ésta no es una solución satisfactoria. Con el tiempo, tendrá que extender la clase `Lista` y sus clases derivadas. Asegurarse de que todos los cambios se propaguen a todas las clases relacionadas sería una pesadilla.

Las plantillas resuelven este problema, y con la adopción del estándar ANSI son una parte integral del lenguaje. Como todo lo que hay en C++, tienen seguridad de tipos y son muy flexibles.

## Tipos parametrizados

Las plantillas le enseñan al compilador cómo crear una lista de cualquier tipo de objeto, en lugar de crear un conjunto de listas de tipos específicos (una `ListaPiezas` es una lista de piezas, una `ListaGatos` es una lista de gatos. En lo único que son diferentes es en el tipo del objeto que hay en la lista). Con las plantillas, el tipo de objeto de la lista se convierte en un parámetro para la definición de la clase.

Un componente común de casi todas las bibliotecas de C++ es una clase de arreglos. Como vio con las clases `Lista`, es tedioso e ineficiente crear una clase de arreglos para enteros, otra para dobles, y otra más para un arreglo de objetos `Animal`. Las plantillas le permiten declarar una clase de arreglos parametrizada y luego especificar qué tipo de objeto guardará cada instancia del arreglo. Hay que tener en cuenta que la STL (Biblioteca Estándar de Plantillas) proporciona un conjunto estandarizado de clases *contenedoras*, incluyendo arreglos, listas, etc. En esta lección verá lo que necesita para hacer sus propias plantillas y para entender perfectamente su funcionamiento; sin embargo, en un programa comercial, muy probablemente utilice las clases de la STL en lugar de las suyas.

### Cómo crear una instancia a partir de una plantilla

La *instanciación* es el acto de crear un tipo específico a partir de una plantilla. Las clases individuales se llaman *instancias* de la plantilla.

Las *plantillas parametrizadas* le proporcionan la capacidad de crear una clase general y pasar tipos como parámetros a esa clase para crear instancias específicas.

## Definición de una plantilla

Para declarar un objeto parametrizado `Arreglo` (una plantilla para un arreglo) se escribe

```
1: template <class T>      // declarar la plantilla y el parámetro
2: class Arreglo           // la clase que se va a parametrizar
3: {
```

```
4:     public:  
5:         Arreglo();  
6:     // aqui va la declaración completa de la clase  
7: };
```

La palabra reservada `template` se utiliza al principio de cualquier declaración y definición de una clase de una plantilla. Los parámetros de la plantilla van después de la palabra reservada `template`. Los parámetros son las cosas que cambiarán con cada instancia. Por ejemplo, en la plantilla del arreglo mostrado anteriormente va a cambiar el tipo de los objetos guardados en el arreglo. Una instancia podría guardar un arreglo de enteros, y otra podría guardar un arreglo de objetos `Animal`.

En este ejemplo se utiliza la palabra reservada `class` seguida del identificador `T`. La palabra reservada `class` indica que este parámetro es un tipo. El identificador `T` se utiliza en el resto de la definición de la plantilla para referirse al tipo parametrizado. Una instancia de esta clase substituirá a `T` con `int` en cualquier parte en que aparezca, y otra lo substituirá con `Gato`.

Para declarar una instancia `int` y una instancia `Gato` de la clase parametrizada `Arreglo`, se escribiría lo siguiente:

```
Arreglo<int> unArregloInt;  
Arreglo<Gato> unArregloGato;
```

El objeto `unArregloInt` es del tipo *arreglo de enteros*; el objeto `unArregloGato` es del tipo *arreglo de gatos*. Ahora puede utilizar el tipo `Arreglo<int>` en cualquier parte en que normalmente utilizaría un tipo (como en el valor de retorno de una función, como un parámetro para una función, etcétera). El listado 19.1 muestra la declaración completa de esta plantilla `Arreglo` simplificada.

## Nota

¡El listado 19.1 no es un programa completo y no compilará!

19

### ENTRADA

### LISTADO 19.1 Una plantilla para una clase llamada `Arreglo`

```
1: #include <iostream.h>  
2:  
3: //Listado 19.1 La plantilla de una clase de tipo arreglo  
4:  
5: const int TamanioPredet = 10;  
6:  
7: // declarar la plantilla y el parámetro  
8: template < class T >  
9: // la clase que se va a parametrizar  
10: class Arreglo  
11: {  
12:     public:  
13:         // constructores
```

continúa

**LISTADO 19.1** CONTINUACIÓN

```

14:     Arreglo(int suTamanio = TamanioPredet);
15:     Arreglo(const Arreglo & rhs);
16:     ~Arreglo()
17:     { delete [] apTipo; }
18:     // operadores
19:     Arreglo& operator=(const Arreglo & );
20:     T & operator[](int desplazamiento)
21:     { return apTipo[ desplazamiento ]; }
22:     // métodos de acceso
23:     int obtenerTamanio()
24:     { return suTamanio; }
25: private:
26:     T * apTipo;
27:     int suTamanio;
28: };

```

**SALIDA** No hay salida. Este programa está incompleto.

**ANÁLISIS** La definición de la plantilla empieza en la línea 8 con la palabra reservada `template` seguida del parámetro. En este caso, el parámetro se identifica como un tipo mediante la palabra reservada `class`, y se utiliza el identificador `T` para representar el tipo parametrizado.

Desde la línea 10 hasta el final de la plantilla en la línea 28, el resto de la declaración es como cualquier otra declaración de clase. La única diferencia es que en lugar del tipo del objeto se utiliza el identificador `T`. Por ejemplo, se esperaría que `operator[]` regresara una referencia a un objeto del arreglo, y de hecho se declara para regresar una referencia a un `T`.

Cuando se declara una instancia de un arreglo de enteros, el `operator=` que se proporciona para ese arreglo regresará una referencia a un entero. Cuando se declare una instancia de un arreglo de tipo `Animal`, el `operator=` proporcionado para el arreglo `Animal` regresará una referencia a un `Animal`.

## Uso del nombre

Dentro de la declaración de la clase, se puede utilizar la palabra `Arreglo` sin necesidad de más identificación. En cualquier otra parte del programa se haría referencia a esta clase como `Arreglo<T>`. Por ejemplo, si no escribe el constructor dentro de la declaración de la clase, debe escribir lo siguiente:

```

template < class T >
Arreglo< T >::Arreglo(int tamano):
suTamanio = tamano
{
    apTipo = new T[ tamano ];
    for (int i = 0; i < tamano; i++)
        apTipo[ i ] = 0;
}

```

La declaración en la primera línea de este fragmento de código se requiere para identificar el tipo (`class T`). El nombre de la plantilla es `Arreglo< T >`, y el nombre de la función es `Arreglo(int tamano)`.

El resto de la función es igual a como sería para una función que no sea de una plantilla. Un método común es hacer que la clase y sus funciones trabajen como una simple declaración antes de convertirla en una plantilla.

## Implementación de la plantilla

La implementación completa de la clase de la plantilla `Arreglo` requiere que se implementen el constructor de copia, `operator=`, y así sucesivamente. El listado 19.2 proporciona un programa controlador simple para practicar con esta clase de plantilla.

### Nota

Los compiladores GNU 2.7.2 y posteriores soportan el uso de plantillas. Sin embargo, algunos compiladores antiguos no soportan las plantillas. En la actualidad, las plantillas forman parte del nuevo estándar ANSI de C++. Los principales fabricantes de compiladores soportan plantillas en sus versiones actuales. Si tiene un compilador muy antiguo, no podrá compilar y ejecutar los ejercicios que vienen en esta lección. De todas formas sería bueno que leyera toda la lección, y que viera este material cuando actualice su compilador. Recuerde que el CD-ROM tiene la versión 2.9.5, la cual sí soporta el uso de plantillas.

### ENTRADA

### LISTADO 19.2 La implementación de la plantilla Arreglo

```
1: // Listado 19.2: Implementación de la plantilla arreglo
2:
3: #include<iostream.h>
4:
5: const int TamanoPredet = 10;
6:
7:
8: // declarar una clase Animal simple para poder
9: // crear un arreglo de animales
10: class Animal
11: {
12: public:
13:     Animal(int);
14:     Animal();
15:     ~Animal() {}
16:     int ObtenerPeso() const
17:     { return suPeso; }
18:     void Desplegar() const
19:     { cout << suPeso; }
20: private:
```

19

continúa

**LISTADO 19.2** CONTINUACIÓN

```
21:         int suPeso;
22:     };
23:
24:     Animal::Animal(int peso):
25:         suPeso(peso)
26:     {}
27:
28:     Animal::Animal():
29:         suPeso(0)
30:     {}
31:
32: // declarar la plantilla y el parámetro
33: template < class T >
34: // la clase que se va a parametrizar
35: class Arreglo
36: {
37: public:
38:     // constructores
39:     Arreglo(int suTamanio = TamanioPredet);
40:     Arreglo(const Arreglo & rhs);
41:     ~Arreglo()
42:     { delete [] apTipo; }
43:     // operadores
44:     Arreglo & operator=(const Arreglo &);
45:     T & operator[](int desplazamiento)
46:     { return apTipo[ desplazamiento ]; }
47:     const T & operator[](int desplazamiento) const
48:     { return apTipo[ desplazamiento ]; }
49:     // métodos de acceso
50:     int ObtenerTamanio() const
51:     { return suTamanio; }
52: private:
53:     T * apTipo;
54:     int suTamanio;
55: };
56:
57: // las implementaciones están a continuación...
58: // implementar el Constructor
59: template < class T >
60: Arreglo< T >::Arreglo(int tamanio):
61: suTamanio(tamanio)
62: {
63:     apTipo = new T[ tamanio ];
64:
65:     for (int i = 0; i < tamanio; i++)
66:         apTipo[ i ] = 0;
67: }
68:
69: // constructor de copia
70: template < class T >
```

```
71:     Arreglo< T >::Arreglo(const Arreglo & rhs)
72:     {
73:         suTamanio = rhs.ObtenerTamanio();
74:         apTipo = new T[ suTamanio ];
75:
76:         for (int i = 0; i < suTamanio; i++)
77:             apTipo[ i ] = rhs[ i ];
78:     }
79:
80: // operator=
81: template < class T >
82: Arreglo< T > & Arreglo< T >::operator=(const Arreglo & rhs)
83: {
84:     if (this == &rhs)
85:         return *this;
86:     delete [] apTipo;
87:     suTamanio = rhs.ObtenerTamanio();
88:     apTipo = new T[ suTamanio ];
89:     for (int i = 0; i < suTamanio; i++)
90:         apTipo[ i ] = rhs[ i ];
91:     return *this;
92: }
93:
94: // programa controlador
95: int main()
96: {
97:     // un arreglo de enteros
98:     Arreglo< int > elArreglo;
99:     // un arreglo de Animales
100:    Arreglo< Animal > elZoologico;
101:    Animal * apAnimal;
102:
103:    // llenar los arreglos
104:    for (int i = 0; i < elArreglo.ObtenerTamanio(); i++)
105:    {
106:        elArreglo[ i ] = i * 2;
107:        apAnimal = new Animal(i * 3);
108:        elZoologico[ i ] = *apAnimal;
109:        delete apAnimal;
110:    }
111:    // imprimir el contenido de los arreglos
112:    for (int j = 0; j < elArreglo.ObtenerTamanio(); j++)
113:    {
114:        cout << "elArreglo[" << j << "]:\t";
115:        cout << elArreglo[ j ] << "\t\t";
116:        cout << "elZoologico[" << j << "]:\t";
117:        elZoologico[ j ].Desplegar();
118:        cout << endl;
119:    }
120:    return 0;
121: }
```

**SALIDA**

|               |    |                 |    |
|---------------|----|-----------------|----|
| elArreglo[0]: | 0  | elZoologico[0]: | 0  |
| elArreglo[1]: | 2  | elZoologico[1]: | 3  |
| elArreglo[2]: | 4  | elZoologico[2]: | 6  |
| elArreglo[3]: | 6  | elZoologico[3]: | 9  |
| elArreglo[4]: | 8  | elZoologico[4]: | 12 |
| elArreglo[5]: | 10 | elZoologico[5]: | 15 |
| elArreglo[6]: | 12 | elZoologico[6]: | 18 |
| elArreglo[7]: | 14 | elZoologico[7]: | 21 |
| elArreglo[8]: | 16 | elZoologico[8]: | 24 |
| elArreglo[9]: | 18 | elZoologico[9]: | 27 |

**ANÁLISIS**

Las líneas 10 a 30 proporcionan una clase `Animal` simplificada, creada aquí para que los objetos de un tipo definido por el usuario estén disponibles para agregar al arreglo.

La línea 33 declara que lo que sigue a continuación es una plantilla y que el parámetro para la plantilla es un tipo, designado como `T`. La clase `Arreglo` tiene dos constructores como se muestra, el primero de los cuales toma un tamaño y utiliza como valor predeterminado la constante de tipo entero llamada `TamanioPredet`.

Se declaran los operadores de asignación y de desplazamiento, el último de los cuales declara dos variantes, una `const` y una que no es `const`. El único método de acceso proporcionado es `ObtenerTamanio()`, el cual regresa el tamaño del arreglo.

Ciertamente, uno podría imaginar una interfaz más completa y, para cualquier clase sería de tipo `Arreglo`, lo que se ha proporcionado aquí sería inadecuado. Como mínimo se requerirían los operadores para quitar elementos, expandir el arreglo, empacar el arreglo, etc. Todo esto lo proporcionan las clases contenedoras de la STL, como se explica al final de esta lección.

Los datos privados constan del tamaño del arreglo y de un apuntador al arreglo de objetos que está actualmente en memoria.

## Funciones de plantillas

Si quiere pasar un objeto tipo arreglo a una función, debe pasar una instancia específica del arreglo, no una plantilla. Por lo tanto, si `UnaFuncion()` toma un arreglo de enteros como parámetro, puede escribir

```
void UnaFuncion(Arreglo< int > &); // ok
```

pero no puede escribir

```
void UnaFuncion(Arreglo< T > &); // error!
```

porque no hay forma de saber qué es un `T`. Tampoco puede escribir

```
void UnaFuncion(Arreglo &); // error!
```

porque no hay una clase `Arreglo`, sólo la plantilla y las instancias.

Para llegar al método más general, debe declarar una función de plantilla.

```
template < class T >
void MiFuncionPlantilla(Arreglo< T > &); // ok
```

Aquí, la función `MiFuncionPlantilla()` se declara como una función de plantilla mediante la declaración de la línea superior. Observe que al igual que otras funciones, las funciones de plantilla pueden tener cualquier nombre.

Las funciones de plantilla también pueden tomar instancias de la plantilla además de la forma parametrizada. Lo siguiente es un ejemplo:

```
template < class T >
void MiOtraFuncion(Arreglo< T > &, Arreglo< int > &); // ok
```

Observe que esta función toma dos arreglos: un arreglo parametrizado y un arreglo de enteros. El primero puede ser un arreglo de cualesquier objetos, pero el segundo siempre será un arreglo de enteros.

## Plantillas y funciones amigas

Las clases de plantillas pueden declarar tres tipos de funciones amigas:

- Clases y funciones amigas que no sean de plantilla
- Clases y funciones amigas de plantilla general
- Clases y funciones amigas de plantilla de tipo específico

### Clases y funciones amigas que no son de plantilla

Es posible declarar cualquier clase o función como amiga para la clase de plantilla. Cada instancia de la clase tratará a la amiga en forma apropiada, como si la declaración de amistad se hubiera hecho en esa instancia particular. El listado 19.3 agrega una función amiga trivial, llamada `Intruso()`, a la definición de la plantilla de la clase `Arreglo`, y el programa controlador invoca a `Intruso()`. Como es una amiga, `Intruso()` puede tener acceso a los datos privados del `Arreglo`. Como ésta no es una función de plantilla, sólo se puede llamar en clases `Arreglo` de tipo `int`.

19

#### ENTRADA LISTADO 19.3 Función amiga que no es de plantilla

```
1: // Listado 19.3 - Funciones amigas de tipo específico en plantillas
2:
3: #include <iostream.h>
4:
5: const int TamanioPredef = 10;
6:
7:
8: // declarar una clase Animal simple para poder
```

continúa

**LISTADO 19.3** CONTINUACIÓN

```
9:     crear un arreglo de animales
10:    class Animal
11:    {
12:    public:
13:        Animal(int);
14:        Animal();
15:        ~Animal() {}
16:        int ObtenerPeso() const
17:        { return suPeso; }
18:        void Desplegar() const
19:        { cout << suPeso; }
20:    private:
21:        int suPeso;
22:    };
23:
24:    Animal::Animal(int peso):
25:    suPeso(peso)
26:    {}
27:
28:    Animal::Animal():
29:    suPeso(0)
30:    {}
31:
32:    // declarar la plantilla y el parámetro
33:    template < class T >
34:    // la clase que se va a parametrizar
35:    class Arreglo
36:    {
37:    public:
38:        // constructores
39:        Arreglo(int suTamaño = TamanoPredet);
40:        Arreglo(const Arreglo & rhs);
41:        ~Arreglo()
42:        { delete [] apTipo; }
43:        // operadores
44:        Arreglo & operator=(const Arreglo & );
45:        T & operator[](int desplazamiento)
46:        { return apTipo[ desplazamiento ]; }
47:        const T & operator[](int desplazamiento) const
48:        { return apTipo[ desplazamiento ]; }
49:        // métodos de acceso
50:        int ObtenerTamaño() const
51:        { return suTamaño; }
52:        // función amiga
53:        friend void Intruso(Arreglo< int > );
54:    private:
55:        T * apTipo;
56:        int suTamaño;
57:    };
58:
59:    // función amiga. No es una plantilla, sólo se puede usar
```

```

60:      // con arreglos de enteros! Se inmiscuye en los datos privados.
61:      void Intruso(Arreglo< int > elArreglo)
62:      {
63:          cout << "\n*** Intruso ***\n";
64:          for (int i = 0; i < elArreglo.suTamanio; i++)
65:              cout << "i: " << elArreglo.apTipo[ i ] << endl;
66:          cout << "\n";
67:      }
68:
69:      // las implementaciones están a continuación...
70:      // implementar el Constructor
71:      template < class T >
72:      Arreglo< T >::Arreglo(int tamanio):
73:          suTamanio(tamanio)
74:      {
75:          apTipo = new T[ tamanio ];
76:
77:          for (int i = 0; i < tamanio; i++)
78:              apTipo[i] = 0;
79:      }
80:
81:      // constructor de copia
82:      template < class T >
83:      Arreglo< T >::Arreglo(const Arreglo & rhs)
84:      {
85:          suTamanio = rhs.ObtenerTamanio();
86:          apTipo = new T[ suTamanio ];
87:
88:          for (int i = 0; i < suTamanio; i++)
89:              apTipo[ i ] = rhs[ i ];
90:      }
91:
92:      // operator=
93:      template < class T >
94:      Arreglo< T > & Arreglo< T >::operator=(const Arreglo & rhs)
95:      {
96:          if (this == &rhs)
97:              return *this;
98:          delete [] apTipo;
99:          suTamanio = rhs.ObtenerTamanio();
100:         apTipo = new T[ suTamanio ];
101:         for (int i = 0; i < suTamanio; i++)
102:             apTipo[ i ] = rhs[ i ];
103:         return *this;
104:     }
105:
106:     // programa controlador
107:     int main()
108:     {
109:         // un arreglo de enteros
110:         Arreglo< int > elArreglo;
111:         // un arreglo de animales

```

19

continua

**LISTADO 19.3** CONTINUACIÓN

```
112:             Arreglo< Animal > elZoologico;
113:             Animal *apAnimal;
114:
115:             // llenar los arreglos
116:             for (int i = 0; i < elArreglo.ObtenerTamanio(); i++)
117:             {
118:                 elArreglo[ i ] = i * 2;
119:                 apAnimal = new Animal(i * 3);
120:                 elZoologico[ i ] = *apAnimal;
121:             }
122:             for (int j = 0; j < elArreglo.ObtenerTamanio(); j++)
123:             {
124:                 cout << "elZoologico[" << j << "]::\t";
125:                 elZoologico[ j ].Desplegar();
126:                 cout << endl;
127:             }
128:             cout << "Usar ahora la función amiga para ";
129:             cout << "encontrar los miembros de Arreglo<int>";
130:             Intruso(elArreglo);
131:             cout << "\n\nListo.\n";
132:             return 0;
133: }
```

---

**SALIDA**

```
elZoologico[0]:      0
elZoologico[1]:      3
elZoologico[2]:      6
elZoologico[3]:      9
elZoologico[4]:     12
elZoologico[5]:     15
elZoologico[6]:     18
elZoologico[7]:     21
elZoologico[8]:     24
elZoologico[9]:     27
Usar ahora la función amiga para encontrar los miembros de Arreglo<int>
*** Intruso ***
i: 0
i: 2
i: 4
i: 6
i: 8
i: 10
i: 12
i: 14
i: 16
i: 18

Listo.
```

**ANÁLISIS**

La declaración de la plantilla Arreglo se ha extendido para incluir a la función amiga `Intruso()`. Esto declara que cada instancia de un arreglo de enteros considerará a `Intruso()` como una función amiga; por lo tanto, `Intruso()` tendrá acceso a los datos y funciones miembro privados de la instancia del arreglo.

En la línea 64 `Intruso()` tiene acceso directo a `suTamanio`, y en la línea 65 tiene acceso directo a `apTipo`. Este uso trivial de estos miembros fue innecesario, ya que la clase `Arreglo` proporciona métodos públicos de acceso para estos datos, pero sirve para demostrar cómo se pueden declarar las funciones amigas con las plantillas.

## Clases y funciones amigas de plantilla general

Sería útil agregar un operador de despliegue para la clase `Arreglo`. Una forma sería declarar un operador de despliegue para cada posible tipo de `Arreglo`, pero esto debilitaría el propósito en sí de hacer que `Arreglo` sea una plantilla.

Lo que se necesita es un operador de inserción que funcione para cualquier posible tipo de `Arreglo`.

```
ostream& operator<< (ostream &, Arreglo< T > &);
```

Para hacer que esto funcione, necesita declarar a `operator<<` como una función de plantilla.

```
template < class T > ostream & operator<< (ostream &, Arreglo< T > &)
```

Ahora que `operator<<` es una función de plantilla, sólo necesita proporcionar una implementación. El listado 19.4 muestra la plantilla `Arreglo` extendida para incluir esta declaración, y proporciona la implementación para `operator<<`.

19

**ENTRADA LISTADO 19.4 Uso del operador ostream**

```

1: // Listado 19.4: uso del operador ostream
2:
3: #include <iostream.h>
4:
5: const int TamanoPredet = 10;
6:
7:
8: class Animal
9:
10: public:
11:     Animal(int);
12:     Animal();
13:     ~Animal() {}
14:     int ObtenerPeso() const
15:     { return suPeso; }
16:     void Desplegar() const
17:     { cout << suPeso; }
18: private:
```

continúa

**LISTADO 19.4** CONTINUACIÓN

```
19:         int suPeso;
20:     };
21:
22:     Animal::Animal(int peso):
23:         suPeso(peso)
24:     {}
25:
26:     Animal::Animal():
27:         suPeso(0)
28:     {}
29:
30:     // declarar la plantilla y el parámetro
31:     template < class T >
32:     // la clase que se va a parametrizar
33:     class Arreglo
34:     {
35:         public:
36:             // constructores
37:             Arreglo(int suTamanio = TamanioPredet);
38:             Arreglo(const Arreglo & rhs);
39:             ~Arreglo()
40:             { delete [] apTipo; }
41:             // operadores
42:             Arreglo & operator=(const Arreglo &);

43:             T & operator[](int desplazamiento)
44:             { return apTipo[ desplazamiento ]; }

45:             const T & operator[](int desplazamiento) const
46:             { return apTipo[ desplazamiento ]; }

47:             // métodos de acceso
48:             int ObtenerTamanio() const
49:             { return suTamanio; }

50:             friend ostream & operator<< >> (ostream &, Arreglo< T > &);

51:         private:
52:             T * apTipo;
53:             int suTamanio;
54:         };

55:
56:         template < class T >
57:         ostream & operator<< (ostream & salida, Arreglo< T > & elArreglo)
58:         {
59:             for (int i = 0; i < elArreglo.ObtenerTamanio(); i++)
60:                 salida << "[" << i << "] " << elArreglo[ i ] << endl;
61:             return salida;
62:         }

63:
64:         // las implementaciones están a continuación...
65:         // implementar el Constructor
66:         template < class T >
67:         Arreglo< T >::Arreglo(int tamanio):
68:             suTamanio(tamanio)
69:         {
70:             apTipo = new T[ tamanio ];
```

```
71:         for (int i = 0; i < tamano; i++)
72:             apTipo[ i ] = 0;
73:     }
74:
75:     // constructor de copia
76:     template < class T >
77:     Arreglo< T >::Arreglo(const Arreglo & rhs)
78:     {
79:         suTamanio = rhs.ObtenerTamanio();
80:         apTipo = new T[ suTamanio ];
81:
82:         for (int i = 0; i < suTamanio; i++)
83:             apTipo[ i ] = rhs[ i ];
84:     }
85:
86:     // operator=
87:     template < class T >
88:     Arreglo< T > & Arreglo< T >::operator=(const Arreglo & rhs)
89:     {
90:         if (this == & rhs)
91:             return *this;
92:         delete [] apTipo;
93:         suTamanio = rhs.ObtenerTamanio();
94:         apTipo = new T[ suTamanio ];
95:         for (int i = 0; i < suTamanio; i++)
96:             apTipo[ i ] = rhs[ i ];
97:         return *this;
98:     }
99:
100:
101:    int main()
102:    {
103:        // indicador para el ciclo
104:        bool Detener = false;
105:        int desplazamiento, valor;
106:        Arreglo< int > elArreglo;
107:
108:        while (!Detener)
109:        {
110:            cout << "Escriba un desplazamiento (0-9) ";
111:            cout << "y un valor. (-1 para detener): " ;
112:            cin >> desplazamiento >> valor;
113:            if (desplazamiento < 0)
114:                break;
115:            if (desplazamiento > 9)
116:            {
117:                cout << "***Utilice valores entre 0 y 9.***\n";
118:                continue;
119:            }
120:            elArreglo[ desplazamiento ] = valor;
121:        }
122:        cout << "\nHe aqui el arreglo completo:\n";
123:        cout << elArreglo << endl;
124:        return 0;
125:    }
```

**SALIDA**

```
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 1 10
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 2 20
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 3 30
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 4 40
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 5 50
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 6 60
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 7 70
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 8 80
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 9 90
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 10 10
***Utilice valores entre 0 y 9.***
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): -1 -1
```

He aquí el arreglo completo:

```
[0] 0
[1] 10
[2] 20
[3] 30
[4] 40
[5] 50
[6] 60
[7] 70
[8] 80
[9] 90
```

**Nota**

Debe escribir -1 dos veces para que se pueda completar el programa. Esto se debe a que `cin` espera dos números enteros. Si sólo escribe -1 una vez, el programa se quedará esperando la segunda entrada (sin proporcionar un indicador de ningún tipo).

**ANÁLISIS**

En la línea 50 se declara la plantilla de la función `operator<<()` como amiga de la clase de plantilla `Arreglo`. Como `operator<<()` se implementa como función de plantilla, cada instancia de este tipo de arreglo parametrizado tendrá automáticamente una función `operator<<()`. La implementación para este operador empieza en la línea 56. Cada miembro de un arreglo se llama uno por uno. Esto funciona sólo si se define una función `operator<<()` para cada tipo de objeto guardado en el arreglo.

**Nota**

Muchos compiladores no requieren de los símbolos <> mostrados en la línea 50 del listado 19.4; g++ sí. Si tiene la línea sin ellos (como se muestra en la siguiente versión de la línea 50, aceptada por la mayoría de los compiladores):

```

50: friend ostream & operator<< (ostream &, Arreglo< T > &);

La versión 2.9.5 emitirá advertencias acerca de este código:
lst19-04.cxx:50: warning: friend declaration 'class ostream &
➥operator <<(class ostream &, class Arreglo<T> &)'
lst19-04.cxx:50: warning: declares a non-template function
lst19-04.cxx:50: warning: (if this is not what you intended,
➥make sure
lst19-04.cxx:50: warning: the function template has already been
➥declared,
lst19-04.cxx:50: warning: and add <> after the function name
➥here)
/ttmp/ccm8T7bn.o: In function `main':
/ttmp/ccm8T7bn.o(.text+0x101): undefined reference to
➥`operator<<(ostream &, Arreglo<int> &)'
collect2: ld returned 1 exit status

```

Así que, si utiliza ejemplos de otros libros y obtiene ese mensaje, necesitará hacer el cambio sugerido en el código (como se muestra en la línea 50 del listado 19.4).

## Uso de elementos de plantilla

Puede tratar a los elementos de plantilla de la misma forma que a cualquier otro tipo. Puede pasarlos como parámetros, ya sea por referencia o por valor, y puede regresarlos como los valores de retorno de las funciones, también por valor o por referencia. El listado 19.5 muestra cómo pasar objetos de plantilla.

19

### ENTRADA LISTADO 19.5 Paso de un objeto de plantilla hacia funciones y desde ellas

```

1: // Listado 19.5: Paso de objetos de plantilla
2:
3: #include <iostream.h>
4:
5: const int TamanioPredet = 10;
6:
7:
8: // Una clase trivial para agregar a los arreglos
9: class Animal
10: {
11: public:
12:     // constructores
13:     Animal(int);
14:     Animal();
15:     ~Animal();
16:     // métodos de acceso
17:     int ObtenerPeso() const
18:     { return suPeso; }
19:     void AsignarPeso(int elPeso)

```

continúa

**LISTADO 19.5** CONTINUACIÓN

```
20:         { suPeso = elPeso; }
21:         // operadores amigos
22:         friend ostream & operator<< (ostream &, const Animal &);
23:     private:
24:         int suPeso;
25:     };
26:
27:     // operador de extracción para imprimir animales
28:     ostream & operator<< (ostream & elFlujo, const Animal & elAnimal)
29:     {
30:         elFlujo << elAnimal.ObtenerPeso();
31:         return elFlujo;
32:     }
33:
34:     Animal::Animal(int peso):
35:         suPeso(peso)
36:     {
37:         // cout << "Animal(int)\n";
38:     }
39:
40:     Animal::Animal():
41:         suPeso(0)
42:     {
43:         // cout << "Animal()\n";
44:     }
45:
46:     Animal::~Animal()
47:     {
48:         // cout << "Se destruyó un animal...\n";
49:     }
50:
51:     // declarar la plantilla y el parámetro
52:     template < class T >
53:     // la clase que se va a parametrizar
54:     class Arreglo
55:     {
56:     public:
57:         Arreglo(int suTamanio = TamanoPredet);
58:         Arreglo(const Arreglo &rhs);
59:         ~Arreglo()
60:         { delete [] apTipo; }
61:         Arreglo & operator=(const Arreglo &);
62:         T & operator[](int desplazamiento)
63:         { return apTipo[ desplazamiento ]; }
64:         const T & operator[](int desplazamiento) const
65:         { return apTipo[ desplazamiento ]; }
66:         int ObtenerTamanio() const
67:         { return suTamanio; }
68:         // función amiga
69:         friend ostream & operator<< >> (ostream &, const Arreglo< T > &);
70:     private:
71:         T *apTipo;
```

```
72:         int suTamanio;
73:     };
74:
75:     template < class T >
76:     ostream & operator<< (ostream & salida, const Arreglo< T > & elArreglo)
77:     {
78:         for (int i = 0; i < elArreglo.ObtenerTamanio(); i++)
79:             salida << "[" << i << "] " << elArreglo[ i ] << endl;
80:         return salida;
81:     }
82:
83: // las implementaciones están a continuación...
84: // implementar el Constructor
85: template < class T >
86: Arreglo< T >::Arreglo(int tamanio):
87: suTamanio(tamanio)
88: {
89:     apTipo = new T[ tamanio ];
90:
91:     for (int i = 0; i < tamanio; i++)
92:         apTipo[ i ] = 0;
93: }
94:
95: // constructor de copia
96: template < class T >
97: Arreglo< T >::Arreglo(const Arreglo & rhs)
98: {
99:     suTamanio = rhs.ObtenerTamanio();
100:    apTipo = new T[ suTamanio ];
101:
102:    for (int i = 0; i < suTamanio; i++)
103:        apTipo[ i ] = rhs[ i ];
104: }
105:
106: void FuncionLlenarInt(Arreglo< int > & elArreglo);
107:
108: void FuncionLlenarAnimal(Arreglo< Animal > & elArreglo);
109:
110: int main()
111: {
112:     Arreglo< int > arregloInt;
113:     Arreglo< Animal > arregloAnimal;
114:
115:     FuncionLlenarInt(arregloInt);
116:     FuncionLlenarAnimal(arregloAnimal);
117:     cout << "arregloInt...\\n" << arregloInt;
118:     cout << "\\narregloAnimal...\\n" << arregloAnimal << endl;
119:     return 0;
120: }
121:
122: void FuncionLlenarInt(Arreglo< int > & elArreglo)
123: {
124:     bool Detener = false;
```

**LISTADO 19.5** CONTINUACIÓN

---

```
125:         int desplazamiento, valor;
126:
127:         while (!Detener)
128:         {
129:             cout << "Escriba un desplazamiento (0-9) ";
130:             cout << "y un valor. (-1 para detener): " ;
131:             cin >> desplazamiento >> valor;
132:             if (desplazamiento < 0)
133:                 break;
134:             if (desplazamiento > 9)
135:             {
136:                 cout << "***Utilice valores entre 0 y 9.***\n";
137:                 continue;
138:             }
139:             elArreglo[ desplazamiento ] = valor;
140:         }
141:     }
142:
143:
144:     void FuncionLlenarAnimal(Arreglo< Animal > & elArreglo)
145:     {
146:         Animal * apAnimal;
147:
148:         for (int i = 0; i < elArreglo.ObtenerTamanio(); i++)
149:         {
150:             apAnimal = new Animal;
151:             apAnimal->AsignarPeso(i * 100);
152:             elArreglo[i] = *apAnimal;
153:             // se colocó una copia en el arreglo
154:             delete apAnimal;
155:         }
156:     }
```

---

**SALIDA**

```
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 1 10
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 2 20
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 3 30
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 4 40
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 5 50
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 6 60
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 7 70
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 8 80
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 9 90
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 10 10
***Utilice valores entre 0 y 9.***
Escriba un desplazamiento (0-9) y un valor. (-1 para detener): -1 -1

arregloInt...
[0] 0
[1] 10
```

```
[2] 20  
[3] 30  
[4] 40  
[5] 50  
[6] 60  
[7] 70  
[8] 80  
[9] 90  
  
arregloAnimal...  
[0] 0  
[1] 100  
[2] 200  
[3] 300  
[4] 400  
[5] 500  
[6] 600  
[7] 700  
[8] 800  
[9] 900
```

**ANÁLISIS**

La mayor parte de la implementación de la clase `Arreglo` se omite para ahorrar espacio. La clase `Animal` se declara en las líneas 9 a 25. Aunque ésta es una clase simplificada, proporciona su propio operador de inserción (`<<`) para permitir la impresión de objetos `Animal`. Lo que se hace es simplemente imprimir el peso actual de `Animal`.

Observe que `Animal` tiene un constructor predeterminado. Esto es necesario ya que cuando se agrega un objeto a un arreglo, se utiliza el constructor predeterminado del objeto para crearlo. Esto crea algunas dificultades, como veremos más adelante.

En la línea 106 se declara la función `FuncionLlenarInt()`. El prototipo indica que esta función toma un arreglo de enteros. Observe que ésta no es una función de plantilla. `FuncionLlenarInt()` sólo espera un tipo de arreglo (un arreglo de enteros). De la misma manera, en la línea 108 se declara la función `FuncionLlenarAnimal()` para tomar un `Arreglo` de objetos de tipo `Animal`.

Las implementaciones para estas funciones son distintas, pues llenar un arreglo de enteros no se hace de la misma forma que llenar un arreglo de objetos de tipo `Animal`.

**19**

## Funciones especializadas

Si quita las marcas de comentario de las instrucciones `cout` de los constructores (líneas 37 y 43) y del destructor (línea 48) de `Animal` del listado 19.5, puede ver las construcciones y destrucciones adicionales imprevistas de los objetos de tipo `Animal`.

Cuando se agrega un objeto a un arreglo, se llama al constructor predeterminado del objeto. Sin embargo, el constructor de `Arreglo` asigna un `0` al valor de cada miembro del arreglo, como se muestra en las líneas 91 y 92.

Al escribir `unAnimal = (Animal) 0;`, está llamando al `operator=` predeterminado para `Animal`. Esto ocasiona que se cree un objeto `Animal` temporal usando el constructor, el cual toma un entero (cero). Ese objeto temporal se utiliza como el lado derecho de `operator=` y luego se destruye.

Esto es una desafortunada pérdida de tiempo, pues el objeto `Animal` ya estaba inicializado de manera apropiada. Sin embargo, no puede quitar esta línea porque los enteros no se inicializan automáticamente con un valor de `0`. La solución es enseñar a la plantilla a no utilizar este constructor para objetos de tipo `Animal`, sino utilizar un constructor especial para `Animal`.

Puede proporcionar una implementación explícita para la clase `Animal`, como se indica en el listado 19.6.

---

**ENTRADA LISTADO 19.6 Especialización de las implementaciones de plantilla**

---

```
1: // Listado 19.6: Especialización de implementaciones de plantilla
2:
3: #include <iostream.h>
4:
5: const int TamanoPredet = 3;
6:
7:
8: // Una clase trivial para agregar a los arreglos
9: class Animal
10: {
11: public:
12:     // constructores
13:     Animal(int);
14:     Animal();
15:     ~Animal();
16:     // métodos de acceso
17:     int ObtenerPeso() const
18:     { return suPeso; }
19:     void AsignarPeso(int elPeso)
20:     { suPeso = elPeso; }
21:     // operadores amigos
22:     friend ostream & operator<< (ostream &, const Animal &);
23: private:
24:     int suPeso;
25: };
26:
27: // operador de extracción para imprimir animales
28: ostream & operator<< (ostream & elFlujo, const Animal & elAnimal)
29: {
30:     elFlujo << elAnimal.ObtenerPeso();
31:     return elFlujo;
32: }
33:
34: Animal::Animal(int peso):
35: suPeso(peso)
```

```
36:     {
37:         cout << "animal(int) \n";
38:     }
39:
40:     Animal::Animal():
41:         suPeso(0)
42:     {
43:         cout << "animal() \n";
44:     }
45:
46:     Animal::~Animal()
47:     {
48:         cout << "Se destruyó un animal...\n";
49:     }
50:
51: // declarar la plantilla y el parámetro
52: template < class T >
53: // la clase que se va a parametrizar
54: class Arreglo
55: {
56: public:
57:     Arreglo(int suTamanio = TamanoPredef);
58:     Arreglo(const Arreglo & rhs);
59:     ~Arreglo()
60:     { delete [] apTipo; }
61:     // operadores
62:     Arreglo & operator=(const Arreglo &);

19
63:     T & operator[](int desplazamiento)
64:     { return apTipo[desplazamiento]; }
65:     const T & operator[](int desplazamiento) const
66:     { return apTipo[desplazamiento]; }
67:     // métodos de acceso
68:     int ObtenerTamano() const
69:     { return suTamanio; }
70:     // función amiga
71:     friend ostream & operator<< >> (ostream &, const Arreglo< T > &);
72: private:
73:     T *apTipo;
74:     int suTamanio;
75: };
76:
77: template < class T >
78: Arreglo< T >::Arreglo(int tamano = TamanoPredef):
79: suTamanio(tamano)
80: {
81:     apTipo = new T[ tamano ];
82:
83:     for (int i = 0; i < tamano; i++)
84:         apTipo[ i ] = (T)0;
85: }
86:
87: template < class T >
88: Arreglo< T > & Arreglo< T >::operator=(const Arreglo & rhs)
89: {
90:     if (this == &rhs)
```

**LISTADO 19.5** CONTINUACIÓN

```
91:         return *this;
92:         delete [] apTipo;
93:         suTamanio = rhs.ObtenerTamanio();
94:         apTipo = new T[ suTamanio ];
95:         for (int i = 0; i < suTamanio; i++)
96:             apTipo[ i ] = rhs[ i ];
97:         return *this;
98:     }
99:
100:    template < class T >
101:    Arreglo< T >::Arreglo(const Arreglo & rhs)
102:    {
103:        suTamanio = rhs.ObtenerTamanio();
104:        apTipo = new T[ suTamanio ];
105:
106:        for (int i = 0; i < suTamanio; i++)
107:            apTipo[ i ] = rhs[ i ];
108:    }
109:
110:
111:    template < class T >
112:    ostream & operator<< (ostream & salida, const Arreglo< T > & elArreglo)
113:    {
114:        for (int i = 0; i < elArreglo.ObtenerTamanio(); i++)
115:            salida << "[" << i << "] " << elArreglo[ i ] << endl;
116:        return salida;
117:    }
118:
119:    Arreglo< Animal >::Arreglo(int TamanioArregloAnimal):
120:    suTamanio(TamanioArregloAnimal)
121:    {
122:        apTipo = new Animal[ TamanioArregloAnimal ];
123:    }
124:
125:    void FuncionLlenarInt(Arreglo< int > & elArreglo);
126:
127:    void FuncionLlenarAnimal(Arreglo< Animal > & elArreglo);
128:
129:    int main()
130:    {
131:        Arreglo< int > arregloInt;
132:        Arreglo< Animal > arregloAnimal;
133:
134:        FuncionLlenarInt(arregloInt);
135:        FuncionLlenarAnimal(arregloAnimal);
136:        cout << "arregloInt...\\n" << arregloInt;
137:        cout << "\\narregloAnimal...\\n" << arregloAnimal << endl;
138:        return 0;
139:    }
140:
141:    void FuncionLlenarInt(Arreglo< int > & elArreglo)
142:    {
143:        bool Detener = false;
```

```
144:     int desplazamiento, valor;
145:
146:     while (!Detener)
147:     {
148:         cout << "Escriba un desplazamiento (0-9) y un valor. ";
149:         cout << "(-1 para detener): ";
150:         cin >> desplazamiento >> valor;
151:         if (desplazamiento < 0)
152:             break;
153:         if (desplazamiento > 9)
154:         {
155:             cout << "***Utilice valores entre 0 y 9.***\n";
156:             continue;
157:         }
158:         elArreglo[ desplazamiento ] = valor;
159:     }
160:
161:
162:
163: void FuncionLlenarAnimal(Arreglo< Animal > & elArreglo)
164: {
165:     Animal * apAnimal;
166:
167:     for (int i = 0; i < elArreglo.ObtenerTamano(); i++)
168:     {
169:         apAnimal = new Animal(i * 10);
170:         elArreglo[ i ] = *apAnimal;
171:         delete apAnimal;
172:     }
173: }
```

**Nota**

Se han agregado números de línea a la salida para facilitar el análisis. Los números de línea no aparecerán en la salida que usted obtenga.

**19****SALIDA**

```
1: animal()
2: animal()
3: animal()
4: Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 0 0
5: Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 1 1
6: Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 2 2
7: Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 3 3
8: Escriba un desplazamiento (0-9) y un valor. (-1 para detener): -1
→ -1
9: animal(int)
10: Se destruyó un animal...
11: animal(int)
12: Se destruyo un animal...
13: animal(int)
14: Se destruyó un animal...
15: arregloInt...
16: [0] 0
17: [1] 1
```

```
18: [2] 2
19:
20: arregloAnimal...
21: [0] 0
22: [1] 10
23: [2] 20
24:
25: Se destruyó un animal...
26: Se destruyó un animal...
27: Se destruyó un animal...
28: <<< Segunda ejecución >>>
29: animal(int)
30: Se destruyó un animal...
31: animal(int)
32: Se destruyó un animal...
33: animal(int)
34: Se destruyó un animal...
35: Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 0 0
36: Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 1 1
37: Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 2 2
38: Escriba un desplazamiento (0-9) y un valor. (-1 para detener): 3 3
39: animal(int)
40: Se destruyó un animal...
41: animal(int)
42: Se destruyó un animal...
43: animal(int)
44: Se destruyó un animal...
45: arregloInt...
46: [0] 0
47: [1] 1
48: [2] 2
49:
50: arregloAnimal...
51: [0] 0
52: [1] 10
53: [2] 20
54:
55: Se destruyó un animal...
56: Se destruyó un animal...
57: Se destruyó un animal...
```

**ANÁLISIS**

El listado 19.6 reproduce ambas clases en su totalidad para que pueda ver la creación y destrucción de objetos Animal temporales. El valor de TamañoPredet se ha reducido a 3 para simplificar la salida.

Cada uno de los constructores y destructores de Animal imprime un enunciado, líneas 37, 43 y 48, que indica el momento en que se le llama.

En las líneas 77 a 85 se declara el comportamiento de plantilla de un constructor de Arreglo. En las líneas 119 a 123 se muestra el constructor especializado para un Arreglo de objetos de tipo Animal. Observe que en este constructor especial se permite que el constructor predeterminado establezca el valor inicial para cada Animal, y no se hace ninguna asignación explícita.

La primera vez que se ejecuta este programa se muestra el primer conjunto de la salida. Las líneas 1 a 3 de la salida muestran los tres constructores predeterminados llamados al crear el arreglo. El usuario escribe cuatro números, y éstos se introducen en el arreglo de enteros.

La ejecución salta hasta `FuncionLlenarAnimal()`. Aquí se crea un objeto `Animal` temporal en el heap, línea 169, y su valor se utiliza para modificar el objeto `Animal` del arreglo de la línea 170. En la línea 171 se destruye el `Animal` temporal. Esto se repite para cada miembro del arreglo, y se refleja en las líneas 9 a 14 de la salida.

Al final del programa se destruyen los arreglos, y cuando se llama a sus destructores, también se destruyen todos sus objetos. Esto se refleja en las líneas 25 a 27 de la salida.

Para el segundo conjunto de salida (líneas 29 a 57), se colocan marcas de comentarios en la implementación especial del constructor del arreglo de animales, en las líneas 119 a 123 del programa. Cuando se vuelve a ejecutar el programa, se ejecuta el constructor de plantilla (mostrado en las líneas 77 a 85) cuando se construye el arreglo de objetos de tipo `Animal`.

Esto ocasiona que se llamen objetos `Animal` temporales para cada miembro del arreglo (líneas 83 y 84), y esto se refleja en las líneas 29 a 34 de la salida.

Para todo lo demás, la salida de las dos ejecuciones es idéntica, como se podría esperar.

## Miembros estáticos y plantillas

Una plantilla puede declarar datos miembro estáticos. Cada instancia de la plantilla tiene entonces su propio conjunto de datos estáticos, uno por tipo de clase. Es decir, si agrega un miembro estático a la clase `Arreglo` (por ejemplo, un contador que lleve la cuenta de cuántos arreglos se han creado), tendrá uno de esos miembros por cada tipo: uno para los arreglos de objetos de tipo `Animal` y otro para todos los arreglos de enteros. El listado 19.7 agrega un miembro estático y una función estática a la clase `Arreglo`.

19

**ENTRADA****LISTADO 19.7** Uso de funciones y datos miembro estáticos con plantillas

```
1: // Listado 19.7: Uso de funciones y datos miembro estáticos
2:
3: #include <iostream.h>
4:
5: const int TamanioPredet = 3;
6:
7:
8: // Una clase trivial para agregar a los arreglos
9: class Animal
10: {
11: public:
12:     // constructores
13:     Animal(int);
14:     Animal();
15:     ~Animal();
```

continúa

**LISTADO 19.7** CONTINUACIÓN

```
16:          // métodos de acceso
17:          int ObtenerPeso() const
18:          { return suPeso; }
19:          void AsignarPeso(int elPeso)
20:          { suPeso = elPeso; }
21:          // operadores amigos
22:          friend ostream & operator<< (ostream &, const Animal &);
23: private:
24:         int suPeso;
25:     };
26:
27:     // operador de extracción para imprimir animales
28:     ostream & operator<< (ostream & elFlujo, const Animal & elAnimal)
29:     {
30:         elFlujo << elAnimal.ObtenerPeso();
31:         return elFlujo;
32:     }
33:
34:     Animal::Animal(int peso):
35:         suPeso(peso)
36:     {
37:         //cout << "animal(int) \n";
38:     }
39:
40:     Animal::Animal():
41:         suPeso(0)
42:     {
43:         //cout << "animal() \n";
44:     }
45:
46:     Animal::~Animal()
47:     {
48:         //cout << "Se destruyó un animal...\n";
49:     }
50:
51:     // declarar la plantilla y el parámetro
52:     template < class T >
53:     // la clase que se va a parametrizar
54:     class Arreglo
55:     {
56: public:
57:     // constructores
58:     Arreglo(int suTamanio = TamanioPredet);
59:     Arreglo(const Arreglo & rhs);
60:     ~Arreglo()
61:     { delete [] apTipo; suNumeroArreglos--; }
62:     // operadores
63:     Arreglo & operator=(const Arreglo &);
64:     T & operator[](int desplazamiento)
65:     { return apTipo[ desplazamiento ]; }
66:     const T & operator[](int desplazamiento) const
```

```
67:         { return apTipo[ desplazamiento ]; }
68: // métodos de acceso
69:     int ObtenerTamanio() const
70:     { return suTamanio; }
71:     static int ObtenerNumeroArreglos()
72:     { return suNumeroArreglos; }
73:     // función amiga
74:     friend ostream & operator<< >> (ostream &, const Arreglo< T > &);
75: private:
76:     T *apTipo;
77:     int suTamanio;
78:     static int suNumeroArreglos;
79: };
80:
81: template < class T >
82: int Arreglo< T >::suNumeroArreglos = 0;
83:
84: template < class T >
85: Arreglo< T >::Arreglo(int tamanio = TamanioPredet):
86: suTamanio(tamanio)
87:
88:     apTipo = new T[ tamanio ];
89:
90:     for (int i = 0; i < tamanio; i++)
91:         apTipo[ i ] = (T)0;
92:     suNumeroArreglos++;
93: }
94:
95: template < class T >
96: Arreglo< T > & Arreglo< T >::operator=(const Arreglo & rhs)
97:
98:     if (this == &rhs)
99:         return *this;
100:    delete [] apTipo;
101:    suTamanio = rhs.ObtenerTamanio();
102:    apTipo = new T[ suTamanio ];
103:    for (int i = 0; i < suTamanio; i++)
104:        apTipo[ i ] = rhs[ i ];
105: }
106:
107: template < class T >
108: Arreglo< T >::Arreglo(const Arreglo & rhs)
109:
110:    suTamanio = rhs.ObtenerTamanio();
111:    apTipo = new T[ suTamanio ];
112:
113:    for (int i = 0; i < suTamanio; i++)
114:        apTipo[ i ] = rhs[ i ];
115:    suNumeroArreglos++;
116: }
117:
118:
```

**LISTADO 19.7** CONTINUACIÓN

```
119:    template < class T >
120:    ostream & operator<< (ostream & salida, const Arreglo< T > & elArreglo)
121:    {
122:        for (int i = 0; i < elArreglo.ObtenerTamanio(); i++)
123:            salida << "[" << i << "] " << elArreglo[ i ] << endl;
124:        return salida;
125:    }
126:
127:    int main()
128:    {
129:        cout << Arreglo< int >::ObtenerNumeroArreglos();
130:        cout << " arreglos de enteros\n";
131:        cout << Arreglo< Animal >::ObtenerNumeroArreglos();
132:        cout << " arreglos de animales\n\n";
133:
134:        Arreglo< int > arregloInt;
135:        Arreglo< Animal > arregloAnimal;
136:
137:        cout << arregloInt.ObtenerNumeroArreglos();
138:        cout << " arreglos de enteros\n";
139:        cout << arregloAnimal.ObtenerNumeroArreglos();
140:        cout << " arreglos de animales\n\n";
141:
142:        Arreglo< int > *apArregloInt = new Arreglo< int >;
143:
144:        cout << Arreglo< int >::ObtenerNumeroArreglos();
145:        cout << " arreglos de enteros\n";
146:        cout << Arreglo< Animal >::ObtenerNumeroArreglos();
147:        cout << " arreglos de animales\n\n";
148:
149:        delete apArregloInt;
150:
151:        cout << Arreglo< int >::ObtenerNumeroArreglos();
152:        cout << " arreglos de enteros\n";
153:        cout << Arreglo< Animal >::ObtenerNumeroArreglos();
154:        cout << " arreglos de animales\n\n";
155:        return 0;
156:    }
```

**SALIDA**

0 arreglos de enteros  
0 arreglos de animales

1 arreglos de enteros  
1 arreglos de animales

2 arreglos de enteros  
1 arreglos de animales

1 arreglos de enteros  
1 arreglos de animales

**ANÁLISIS**

La declaración de la clase `Animal` se ha omitido para ahorrar espacio. La clase `Arreglo` ha agregado la variable estática `suNumeroArreglos` en la línea 78, y como este dato es privado, se agregó el método de acceso público estático `ObtenerNumeroArreglos()` en la línea 71.

La inicialización de los datos estáticos se logra con una identificación completa de la plantilla, como se muestra en las líneas 81 y 82. Cada uno de los constructores de `Arreglo`, así como el destructor, se modifican para llevar la cuenta de cuántos arreglos existen en un momento dado.

El acceso a los miembros estáticos se logra de la misma forma que con los miembros estáticos de cualquier clase: puede hacerlo con un objeto existente, como se muestra en las líneas 137 y 139, o mediante el uso de la especificación completa de la clase, como se muestra en las líneas 129 y 131. Observe que debe utilizar un tipo de arreglo específico al acceder a los datos estáticos. Existe una variable para cada tipo.

**Nota**

La versión 2.7.2 no puede compilar este código—aparecen los siguientes mensajes:

```
lst19-07.cxx:78: sorry, not implemented: static data member  
templates  
lst19-07.cxx:78: end of file read inside definition
```

Es mejor utilizar la versión 2.9.5 para todos los ejemplos restantes de esta lección. Es el compilador más reciente y tiene características más avanzadas.

**DEBE**

DEBE utilizar miembros estáticos con las plantillas cuando sea necesario.

DEBE especializar el comportamiento de la plantilla redefiniendo por tipo las funciones de plantilla.

DEBE utilizar los parámetros para funciones de plantilla para que sus instancias tengan seguridad en los tipos.

**NO DEBE****19**

## La Biblioteca Estándar de Plantillas

Una nueva característica de C++ es la adopción de la STL (Biblioteca Estándar de Plantillas). Los principales fabricantes de compiladores ofrecen ahora la STL como parte de sus compiladores. Las versiones más recientes del compilador GNU incluyen la STL. Ésta es una biblioteca de clases contenedoras basadas en plantillas, incluyendo vectores, listas, colas y pilas. La STL también incluye una variedad de algoritmos comunes, como el ordenamiento y la búsqueda.

El objetivo de la STL es ofrecerle una alternativa para que no tenga que volver a inventar estos requerimientos comunes. La STL está probada y depurada, ofrece un alto rendimiento y es gratis. Lo que es más importante, la STL es reutilizable; después de que comprenda cómo utilizar un contenedor de la STL, podrá utilizarlo en todos sus programas sin necesidad de reinventarlo.

## Contenedores

Un *contenedor* es un objeto que guarda otros objetos. La biblioteca estándar de C++ proporciona una serie de clases contenedoras que son herramientas poderosas que ayudan a los desarrolladores de C++ a manejar tareas comunes de programación. Dos de los tipos de clases contenedoras de la STL son la de secuencia y la asociativa. Los contenedores *de secuencia* están diseñados para proporcionar un acceso secuencial y aleatorio a sus miembros, o *elementos*. Los contenedores *asociativos* están optimizados para tener acceso a sus elementos mediante valores clave. Igual que otros componentes de la biblioteca estándar de C++, la STL es portable entre varios sistemas operativos. Todas las clases contenedoras de la STL están definidas en el espacio de nombres std.

## Contenedores de secuencia

Los contenedores de secuencia de la STL proporcionan un acceso secuencial eficiente para una lista de objetos. La biblioteca estándar de C++ proporciona tres contenedores de secuencia: vector, list y deque.

### El contenedor vector

Por lo regular, los arreglos se utilizan para guardar y tener acceso a una variedad de elementos. Los elementos de un arreglo son del mismo tipo y se tiene acceso a ellos mediante un índice. La STL proporciona una clase contenedora llamada vector que se comporta igual que un arreglo, pero es más poderosa y segura de utilizar que el arreglo estándar de C++.

Un *vector* es un contenedor optimizado para proporcionar un acceso rápido a sus elementos mediante un índice. La clase contenedora vector se define en el archivo de encabezado <vector> que está en el espacio de nombres std (vea el día 17, “Espacios de nombres”, para obtener más información acerca del uso de espacios de nombres). Un vector puede crecer por sí mismo cuando sea necesario. Suponga que ha creado un vector para contener 10 elementos. Después de agregarle esos 10 objetos, puede decir que el vector está lleno. Si luego agrega otro objeto al vector, éste incrementa automáticamente su capacidad para poder acomodar el undécimo objeto. He aquí la forma en que se define la clase vector:

```
template < class T, class A = allocator< T > > class vector
{
    // miembros de la clase
};
```

El primer argumento (class T) es el tipo de los elementos del vector. El segundo argumento (class A) es una clase asignadora. Los *asignadores* son los administradores de

memoria responsables de la asignación y liberación de memoria para los elementos de cada contenedor. El concepto y la implementación de los asignadores son temas avanzados que están más allá del alcance de este libro.

De manera predeterminada, los elementos se crean mediante el operador `new()` y se liberan mediante el operador `delete()`. Es decir, el constructor predeterminado de la clase `T` se llama para crear un nuevo elemento. Esto proporciona otro argumento que sirve para definir explícitamente un constructor predeterminado para sus propias clases. Si no define uno explícitamente, no podrá utilizar el contenedor vector estándar para guardar un conjunto de instancias de su clase.

Puede definir vectores que guarden enteros y valores de punto flotante de la siguiente manera:

```
vector< int >      vInts;           // vector que guarda elementos de tipo int
vector< float >    vFloats;          // vector que guarda elementos de tipo float
```

Por lo general, debe tener una idea de cuántos elementos contendrá un vector. Por ejemplo, suponga que en su escuela el número máximo de estudiantes es 50. Para crear un vector de estudiantes en una clase, el vector debe ser lo suficientemente grande como para contener 50 elementos. La clase vector estándar proporciona un constructor que acepta el número de elementos como su parámetro. De esta manera, puede definir un vector de 50 estudiantes como se muestra a continuación:

```
vector< Estudiante > ClaseMatematicas(50);
```

El compilador asignará suficiente espacio en memoria para 50 estudiantes; cada elemento se crea utilizando el constructor predeterminado `Estudiante::Estudiante()`.

Puede recuperar el número de elementos de un vector usando una función miembro llamada `size()`. En este ejemplo, `ClaseMatematicas.size()` regresará 50.

Otra función miembro, `capacity()`, le indica exactamente cuántos elementos puede alojar un vector antes de que necesite incrementar su tamaño. Verá más sobre esto más adelante.

Se dice que un vector está vacío si no hay ningún elemento en él; es decir, el tamaño del vector es cero. Para facilitar la prueba para ver si un vector está vacío, la clase vector proporciona una función miembro llamada `empty()` que se evalúa como `true` (verdadero) si el vector está vacío.

Para asignar un objeto `Estudiante` llamado `Harry` a `ClaseMatematicas`, puede utilizar el operador de subíndice `[ ]`:

```
ClaseMatematicas[5] = Harry;
```

El subíndice empieza en `0`. Como pudo haberse dado cuenta, el operador de asignación sobrecargado de la clase `Estudiante` se utiliza aquí para asignar a `Harry` al sexto elemento de `ClaseMatematicas`. De la misma manera, para encontrar la edad de `Harry`, puede tener acceso a su registro escribiendo lo siguiente:

```
ClaseMatematicas[5].ObtenerEdad();
```

Como se mencionó anteriormente, los vectores pueden crecer de manera automática cuando se les agregan más elementos de los que pueden manejar. Por ejemplo, suponga que una clase de su escuela se ha vuelto tan popular que el número de estudiantes pasa de 50. (Bueno, tal vez no ocurra en esta clase de matemáticas, pero quién sabe, a veces ocurren cosas raras.) Cuando se agregue a `ClaseMatematicas` el estudiante número 51, `Sally`, el compilador expandirá el espacio de almacenamiento para darle alojamiento.

Puede agregar un elemento a un vector de varias formas; una de ellas es con `push_back()`:

```
ClaseMatematicas.push_back(Sally);
```

Esta función miembro agrega el nuevo objeto `Estudiante` al final del vector `ClaseMatematicas`. Ahora `ClaseMatematicas` tiene 51 elementos, y `Sally` se coloca en `ClaseMatematicas[50]`.

Para que esta función trabaje, la clase `Estudiante` debe definir un constructor de copia. De no ser así, la función `push_back()` no podrá crear una copia del objeto `Sally`.

La STL no especifica el número máximo de elementos de un vector; los fabricantes de compiladores son los que toman esta decisión. La clase `vector` proporciona una función miembro que le indica cuál es este número mágico en su compilador: `max_size()`. Para GNU, el número máximo de elementos depende del tamaño de cada elemento. Sin importar cuál compilador esté utilizando, la mejor manera de determinar este número es con la función miembro `max_size()`.

El listado 19.8 muestra los miembros de la clase `vector` descritos hasta ahora. Verá que en este listado se utiliza la clase `string` estándar para simplificar el manejo de las cadenas. Para obtener más detalles acerca de la clase `string`, revise la documentación del compilador GNU.

**ENTRADA****LISTADO 19.8 Creación de un vector y acceso a los elementos**

---

```
1:      // Listado 19.8: Vectores
2:
3:      #include <iostream>
4:      #include <string>
5:      #include <vector>
6:
7:      using namespace std;
8:
9:
10:     class Estudiante
11:    {
12:    public:
13:        Estudiante();
14:        Estudiante(const string & nombre, const int edad);
15:        Estudiante(const Estudiante & rhs);
16:        ~Estudiante();
```

```
17:         void AsignarNombre(const string & nombre);
18:         string ObtenerNombre() const;
19:         void AsignarEdad(const int edad);
20:         int ObtenerEdad() const;
21:         Estudiante & operator=(const Estudiante & rhs);
22:     private:
23:         string suNombre;
24:         int suEdad;
25:     };
26:
27:     Estudiante::Estudiante():
28:         suNombre("Nuevo Estudiante"),
29:         suEdad(16)
30:     {}
31:
32:     Estudiante::Estudiante(const string & nombre, const int edad):
33:         suNombre(nombre),
34:         suEdad(edad)
35:     {}
36:
37:     Estudiante::Estudiante(const Estudiante & rhs):
38:         suNombre(rhs.ObtenerNombre()),
39:         suEdad(rhs.ObtenerEdad())
40:     {}
41:
42:     Estudiante::~Estudiante()
43:     {}
44:
45:     void Estudiante::AsignarNombre(const string & nombre)
46:     {
47:         suNombre = nombre;
48:     }
49:
50:     string Estudiante::ObtenerNombre() const
51:     {
52:         return suNombre;
53:     }
54:
55:     void Estudiante::AsignarEdad(const int edad)
56:     {
57:         suEdad = edad;
58:     }
59:
60:     int Estudiante::ObtenerEdad() const
61:     {
62:         return suEdad;
63:     }
64:
65:     Estudiante & Estudiante::operator=(const Estudiante & rhs)
66:     {
67:         suNombre = rhs.ObtenerNombre();
68:         suEdad = rhs.ObtenerEdad();
```

19

continúa

**LISTADO 19.8 CONTINUACIÓN**

```
69:             return *this;
70:     }
71:
72:     ostream & operator<<(ostream & os, const Estudiante & rhs)
73:     {
74:         os << rhs.ObtenerNombre() << " tiene ";
75:         os << rhs.ObtenerEdad() << " años de edad";
76:         return os;
77:     }
78:
79:     template< class T >
80:     // desplegar propiedades del vector
81:     void MostrarVector(const vector< T > & v);
82:
83:     typedef vector< Estudiante > ClaseEscuela;
84:
85:     int main()
86:     {
87:         Estudiante Harry;
88:         Estudiante Sally("Sally", 15);
89:         Estudiante Bill("Bill", 17);
90:         Estudiante Peter("Peter", 16);
91:
92:         ClaseEscuela ClaseVacia;
93:         cout << "ClaseVacia:\n";
94:         MostrarVector(ClaseVacia);
95:
96:         ClaseEscuela ClaseCreciendo(3);
97:         cout << "ClaseCreciendo(3):\n";
98:         MostrarVector(ClaseCreciendo);
99:
100:        ClaseCreciendo[ 0 ] = Harry;
101:        ClaseCreciendo[ 1 ] = Sally;
102:        ClaseCreciendo[ 2 ] = Bill;
103:        cout << "ClaseCreciendo(3) después de asignar estudiantes:\n";
104:        MostrarVector(ClaseCreciendo);
105:
106:        ClaseCreciendo.push_back(Peter);
107:        cout << "ClaseCreciendo() después de agregar el
108:        ↪4to estudiante:\n";
109:        MostrarVector(ClaseCreciendo);
110:
111:        ClaseCreciendo[ 0 ].AsignarNombre("Harry");
112:        ClaseCreciendo[ 0 ].AsignarEdad(18);
113:        cout << "ClaseCreciendo() después de Asignar:\n";
114:        MostrarVector(ClaseCreciendo);
115:        return 0;
116:
117:     // Desplegar propiedades del vector
```

```

118:     template< class T >
119:     void MostrarVector(const vector< T > & v)
120:     {
121:         cout << "max_size() = " << v.max_size();
122:         cout << "\tsize() = " << v.size();
123:         cout << "\tcapacity() = " << v.capacity();
124:         cout << "\t" << (v.empty())? "vacío": "no vacío";
125:         cout << "\n";
126:         for (int i = 0; i < v.size(); ++i)
127:             cout << v[ i ] << "\n";
128:         cout << endl;
129:     }

```

**SALIDA**

ClaseVacia:  
 max\_size() = 536870911 size() = 0 capacity() = 0 vacío

ClaseCreciendo(3):

max\_size() = 536870911 size() = 3 capacity() = 3 no vacío  
 Nuevo Estudiante tiene 16 años de edad  
 Nuevo Estudiante tiene 16 años de edad  
 Nuevo Estudiante tiene 16 años de edad

ClaseCreciendo(3) después de asignar estudiantes:

max\_size() = 536870911 size() = 3 capacity() = 3 no vacío  
 Nuevo Estudiante tiene 16 años de edad  
 Sally tiene 15 años de edad  
 Bill tiene 17 años de edad

ClaseCreciendo() después de agregar el 4to estudiante:

max\_size() = 536870911 size() = 4 capacity() = 6 no vacío  
 Nuevo Estudiante tiene 16 años de edad  
 Sally tiene 15 años de edad  
 Bill tiene 17 años de edad  
 Peter tiene 16 años de edad

ClaseCreciendo() después de Asignar:

max\_size() = 536870911 size() = 4 capacity() = 6 no vacío  
 Harry tiene 18 años de edad  
 Sally tiene 15 años de edad  
 Bill tiene 17 años de edad  
 Peter tiene 16 años de edad

**19**

**ANÁLISIS**

La clase Estudiante se define en las líneas 10 a 25. Las implementaciones de sus funciones miembro se encuentran en las líneas 27 a 70. Es simple y amigable para el contenedor vector. Por las razones mencionadas anteriormente, definimos un constructor predeterminado, un constructor de copia y un operador de asignación sobrecargado. Observe que su variable miembro suNombre se define como una instancia de la clase string de C++. Como puede ver aquí, es mucho más sencillo trabajar con una cadena de C++ que con una cadena char\* estilo C.

La función de plantilla `MostrarVector()` se declara en las líneas 79 y 81 y se define en las líneas 118 a 129. Esta función muestra el uso de algunas de las funciones miembro del vector: `max_size()`, `size()`, `capacity()` y `empty()`. Como puede ver en la salida, el número máximo de objetos `Estudiante` que un vector puede alojar en g++ es 536,870,911 (y sólo 214,748,364 en Visual C++). Este número puede ser distinto para otros tipos de elementos. Por ejemplo, un vector de enteros puede tener muchos elementos más. Si está utilizando otros compiladores, tal vez tenga valores y número máximo de elementos diferentes.

En las líneas 126 y 127 el programa pasa por cada elemento del vector y despliega su valor utilizando el operador de inserción `<<` sobrecargado, el cual se define en las líneas 72 a 77.

En las líneas 87 a 90 se crean cuatro estudiantes. En la línea 92 se define un vector vacío, llamado apropiadamente `ClaseVacia`, por medio del constructor predeterminado de la clase `vector`. Cuando se crea un vector de esta forma, el compilador no le asigna espacio. Como puede ver en la salida producida por `MostrarVector(ClaseVacia)`, su tamaño y capacidad son cero.

En la línea 96 se define un vector de tres objetos de tipo `Estudiante`. Su tamaño y capacidad es de tres, como era de esperarse. Los elementos de `ClaseCreciendo` se asignan con los objetos `Estudiante` en las líneas 100 a 102 por medio del operador de subíndice `[ ]`.

En la línea 106 se agrega el cuarto estudiante, `Peter`, al vector. Esto incrementa el tamaño del vector a cuatro. Es interesante ver que su capacidad ahora se establece en seis. Esto significa que el compilador ha asignado suficiente espacio para guardar hasta seis objetos de tipo `Estudiante`. Debido a que los vectores se deben asignar a un bloque continuo de memoria, su expansión requiere de un conjunto de operaciones. Primero se asigna un nuevo bloque de memoria lo suficientemente grande para los cuatro objetos de tipo `Estudiante`. Luego se copian los tres elementos a esta nueva memoria asignada y el cuarto elemento se agrega después del tercer elemento. Por último, se regresa a la memoria el bloque original. Cuando se tiene un número de elementos grande en un vector, este proceso de liberación y reasignación puede ser muy tardado. Por lo tanto, un compilador emplea una estrategia de optimización para reducir la posibilidad de tener operaciones tan tardadas. En este ejemplo, si agregamos uno o dos objetos más al vector, no hay necesidad de liberar y reasignar memoria.

En las líneas 110 y 111 otra vez utilizamos el operador de subíndice `[ ]` para cambiar las variables miembro para el primer objeto que se encuentra en `ClaseCreciendo`.

**Nota**

El compilador GNU versión 2.7.2 no puede compilar este código. Utilice la versión 2.9.5 para éste y todos los ejemplos que quedan.

**DEBE**

**DEBE** definir un constructor predeterminado para una clase si existe la posibilidad de guardar sus instancias en un vector.

**DEBE** definir un constructor de copia para dicha clase.

**DEBE** definir un operador de asignación sobrecargado para dicha clase.

**No DEBE**

La clase contenedora `vector` tiene otras funciones miembro. La función `front()` regresa una referencia al primer elemento de una lista. La función `back()` regresa una referencia al último elemento. La función `at()` funciona igual que el operador de subíndice `[ ]`. Es más segura porque comprueba si el subíndice que recibe se encuentra dentro del rango de elementos disponibles. Si está fuera del rango, se produce una excepción `out_of_range`. (Las excepciones se tratan en el día 20, “Excepciones y manejo de errores”.)

La función `insert()` inserta uno o más nodos en una posición determinada de un vector. Entonces, la función `pop_back()` quita el último elemento de un vector. Por último, la función `remove()` quita uno o más elementos de un vector.

### El contenedor `list`

Una lista es un contenedor diseñado para optimizar la inserción y eliminación frecuentes de elementos.

La clase contenedora `list` de la STL se define en el archivo de encabezado `<list>` que se encuentra en el espacio de nombres `std`. La clase `list` se implementa como una lista con doble enlace, en la que cada nodo tiene enlaces tanto con el nodo anterior como con el siguiente nodo de la lista.

La clase `list` tiene todas las funciones miembro que proporciona la clase `vector`. Como vio en el repaso de la semana 2, puede desplazarse por una lista siguiendo los enlaces proporcionados en cada nodo. Por lo general, los enlaces se implementan por medio de apuntadores. La clase contenedora `list` estándar utiliza un mecanismo llamado iterador para este mismo propósito.

Un iterador es una generalización de un apuntador. Puede desreferenciar un iterador para recuperar el nodo al que apunta. El listado 19.9 muestra el uso de iteradores para tener acceso a los nodos de una lista.

**ENTRADA****LISTADO 19.9** Cómo desplazarse por una lista usando un iterador

```

1:      // Listado 19.9: Desplazamiento a través de una lista por medio de
2:      ↪un iterador
3:
4:      #include <iostream.h>
5:      #include <list.h>
6:
7:      using namespace std;
8:
9:      typedef list< int > ListaEnteros;
10:
11:     int main()
12:     {
13:         ListaEnteros listaInt;
14:
15:         for (int i = 1; i <= 10; ++i)
16:             listaInt.push_back(i * 2);
17:         for (ListaEnteros::const_iterator ci = listaInt.begin();
18:              ci != listaInt.end(); ++ci)
19:             cout << *ci << " ";
20:             cout << endl;
21:     return 0;
22: }
```

**SALIDA**

2 4 6 8 10 12 14 16 18 20

**ANÁLISIS**

En la línea 13 se define a `listaInt` como una lista de enteros. Los primeros 10 números pares positivos se agregan a la lista usando la función `push_back()` en las líneas 15 y 16.

En las líneas 17 a 19 accedemos a cada nodo de la lista por medio de un iterador constante. Esto indica que no tenemos la intención de cambiar los nodos con este iterador. Si queremos cambiar un nodo a un iterador, necesitamos utilizar un iterador que no sea `const`:

`listaInt::iterator`

La función miembro `begin()` regresa un iterador que apunta al primer nodo de la lista. Como puede ver aquí, se puede utilizar el operador de incremento `++` para apuntar a un iterador al siguiente nodo. La función miembro `end()` es un poco rara: regresa un iterador que apunta a un nodo que está después del último nodo de una lista. Debido a esto, debe asegurarse de que su iterador no llegue a `end()`.

El iterador es desreferenciado igual que un apuntador para regresar el nodo al que apunta, como se muestra en la línea 19.

Aunque esta lección presenta el uso de los iteradores con la clase `list`, la clase `vector` también proporciona iteradores. Además de las funciones proporcionadas en la clase `vector`, la clase `list` también proporciona las funciones `push_front()` y `pop_front()` que funcionan igual que `push_back()` y `pop_back()`. En lugar de agregar y quitar elementos de la parte posterior de la lista, agregan y quitan elementos en la parte frontal de la lista.

### El contenedor `deque`

Un contenedor `deque` es como un vector con dos extremos: hereda la eficiencia de la clase contenedora `vector` en las operaciones secuenciales de lectura y escritura. Pero, además, la clase contenedora `deque` proporciona operaciones optimizadas en primer y segundo planos. Estas operaciones se implementan de manera similar a la clase contenedora `list`, en la que las asignaciones de memoria se aplican sólo para nuevos elementos. Esta característica de la clase `deque` elimina la necesidad de reasignar todo el contenedor a una nueva ubicación de memoria, como se hace con la clase `vector`. Por lo tanto, los contenedores `deque` están idealmente adaptados para aplicaciones en las que las inserciones y las eliminaciones ocurren en uno o en ambos extremos, y para las que es importante el acceso secuencial de los elementos. Un ejemplo de dicha aplicación sería un simulador de enganche de trenes, en el que los carros se pueden unir al tren en ambos extremos.

## Contenedores asociativos

Aunque los contenedores de secuencia están diseñados para un acceso secuencial y aleatorio de elementos por medio del índice o de un iterador, los contenedores asociativos están diseñados para un rápido acceso aleatorio de elementos por medio de claves. La biblioteca estándar de C++ proporciona cuatro contenedores asociativos: `map`, `multimap`, `set` y `multiset`.

19

### El contenedor `map`

Ya vio que un vector es como una versión mejorada de un arreglo. Tiene todas las características de un arreglo y algunas características adicionales. Desafortunadamente, el vector también sufre de una de las debilidades más considerables de los arreglos: no se puede tener acceso aleatorio de los elementos por medio de valores clave que no sean el índice o el iterador. Por otra parte, los contenedores asociativos proporcionan un acceso aleatorio rápido con base en valores clave.

Como ya se dijo, la biblioteca estándar de C++ proporciona cuatro contenedores asociativos: `map`, `multimap`, `set` y `multiset`. En el listado 19.10 se utiliza el contenedor `map` para implementar el ejemplo de la clase de la escuela que se muestra en el listado 19.8.

**ENTRADA****LISTADO 19.10 La clase contenedora map**

---

```
1: // Listado 19.10: Clase contenedora map
2:
3: #include <iostream>
4: #include <string>
5: #include <map>
6:
7: using namespace std;
8:
9:
10: class Estudiante
11: {
12: public:
13:     Estudiante();
14:     Estudiante(const string & nombre, const int edad);
15:     Estudiante(const Estudiante & rhs);
16:     ~Estudiante();
17:     void AsignarNombre(const string & nombre);
18:     string ObtenerNombre() const;
19:     void AsignarEdad(const int edad);
20:     int ObtenerEdad() const;
21:     Estudiante & operator=(const Estudiante & rhs);
22: private:
23:     string suNombre;
24:     int suEdad;
25: };
26:
27: Estudiante::Estudiante():
28:     suNombre("Nuevo Estudiante"),
29:     suEdad(16)
30: {}
31:
32: Estudiante::Estudiante(const string & nombre, const int edad):
33:     suNombre(nombre),
34:     suEdad(edad)
35: {}
36:
37: Estudiante::Estudiante(const Estudiante & rhs):
38:     suNombre(rhs.ObtenerNombre()),
39:     suEdad(rhs.ObtenerEdad())
40: {}
41:
42: Estudiante::~Estudiante()
43: {}
44:
45: void Estudiante::AsignarNombre(const string & nombre)
46: {
47:     suNombre = nombre;
48: }
49:
50: string Estudiante::ObtenerNombre() const
51: {
```

```
52:         return suNombre;
53:     }
54:
55:     void Estudiante::AsignarEdad(const int edad)
56:     {
57:         suEdad = edad;
58:     }
59:
60:     int Estudiante::ObtenerEdad() const
61:     {
62:         return suEdad;
63:     }
64:
65:     Estudiante & Estudiante::operator=(const Estudiante & rhs)
66:     {
67:         suNombre = rhs.ObtenerNombre();
68:         suEdad = rhs.ObtenerEdad();
69:         return *this;
70:     }
71:
72:     ostream & operator<<(ostream & os, const Estudiante & rhs)
73:     {
74:         os << rhs.ObtenerNombre() << " tiene ";
75:         os << rhs.ObtenerEdad() << " años de edad";
76:         return os;
77:     }
78:
79:     template< class T, class A >
80:     // desplegar propiedades del contenedor map
81:     void MostrarMap(const map< T, A > & v);
82:
83:     typedef map< string, Estudiante > ClaseEscuela;
84:
85:     int main()
86:     {
87:         Estudiante Harry("Harry", 18);
88:         Estudiante Sally("Sally", 15);
89:         Estudiante Bill("Bill", 17);
90:         Estudiante Peter("Peter", 16);
91:
92:         ClaseEscuela ClaseMatematicas;
93:         ClaseMatematicas[ Harry.ObtenerNombre() ] = Harry;
94:         ClaseMatematicas[ Sally.ObtenerNombre() ] = Sally;
95:         ClaseMatematicas[ Bill.ObtenerNombre() ] = Bill;
96:         ClaseMatematicas[ Peter.ObtenerNombre() ] = Peter;
97:
98:         cout << "ClaseMatematicas:\n";
99:         MostrarMap(ClaseMatematicas);
100:
101:        cout << "Sabemos que ";
102:                cout << ClaseMatematicas[ "Bill" ].ObtenerNombre();
```

**LISTADO 19.10** CONTINUACIÓN

---

```

103:             cout << " tiene ";
104:             cout << ClaseMatematicas[ "Bill" ].ObtenerEdad();
105:             cout << " años de edad\n";
106:         return 0;
107:     }
108:
109:     // Desplegar propiedades del contenedor map
110:     template< class T, class A >
111:     void MostrarMap(const map< T, A > & v)
112:     {
113:         for (map< T, A >::const_iterator ci = v.begin();
114:              ci != v.end(); ++ci)
115:             cout << ci->first << ":" << ci->second << "\n";
116:         cout << endl;
117:     }

```

---

**SALIDA**

ClaseMatematicas:  
 Bill: Bill tiene 17 años de edad  
 Harry: Harry tiene 18 años de edad  
 Peter: Peter tiene 16 años de edad  
 Sally: Sally tiene 15 años de edad

Sabemos que Bill tiene 17 años de edad

**ANÁLISIS**

En la línea 5 incluimos el archivo de encabezado `<map>` ya que estaremos utilizando la clase contenedora `map` estándar. En consecuencia, definimos la función de plantilla `MostrarMap` para desplegar los elementos de un contenedor `map`. En la línea 83 se define `ClaseEscuela` como un contenedor `map` de elementos; cada uno consta del par (`clave, valor`). El primer elemento en el par es la clave. En `ClaseEscuela` usamos los nombres de los estudiantes como sus claves, que son de tipo `string`. La clave asociada a cada uno de los elementos del contenedor `map` debe ser única; es decir, no puede haber dos elementos con la misma clave. El segundo elemento del par es el objeto en sí, que en el ejemplo es un objeto `Estudiante`. El tipo de datos en par se implementa en la STL como un tipo `struct` de dos elementos: `first` y `second`. Puede utilizar estos elementos para tener acceso a la clave y al valor de un nodo.

Puede saltarse la función `main()` y ver primero la función `MostrarMap`. Esta función utiliza un iterador constante para tener acceso a un objeto de tipo `map`. En la línea 115, `ci->first` apunta a la clave, que es el nombre de un estudiante. `ci->second` apunta al objeto `Estudiante`.

En las líneas 87 a 90 se crean cuatro objetos de tipo `Estudiante`. En la línea 92, `ClaseMatematicas` se define como una instancia de `ClaseEscuela`. En las líneas 93 a 96 agregamos los cuatro estudiantes a `ClaseMatematicas` usando la siguiente sintaxis:

`objeto_map[valor_clave] = valor_objeto;`

También puede utilizar las funciones `push_back()` o `insert()` para agregar un par (clave, valor) al contenedor `map`; para obtener más detalles acerca de esto, consulte la documentación de su compilador GNU.

Después de que se han agregado todos los objetos de tipo `Estudiante` al contenedor `map`, podemos tener acceso a cualquiera de ellos usando sus claves. En las líneas 102 y 104 utilizamos `ClaseMatematicas[ "Bill" ]` para recuperar el registro de Bill.

### Los demás contenedores asociativos

La clase contenedora `multimap` es una clase `map` sin la restricción de claves únicas. Dos o más elementos pueden tener la misma clave.

La clase contenedora `set` también es similar a la clase `map`. La única diferencia es que sus elementos no son pares (clave, valor). Un elemento es sólo la clave.

Por último, la clase contenedora `multiset` es una clase `set` que permite claves duplicadas.

## Pilas

La pila es una de las estructuras de datos utilizadas con más frecuencia en la programación de computadoras. Sin embargo, la pila no se implementa como una clase contenedora independiente. En vez de eso, se implementa como una envoltura de un contenedor. La clase de plantilla `stack` se define en el archivo de encabezado `<stack>` que se encuentra en el espacio de nombres `std`.

Una *pila* es un bloque asignado en forma continua que puede crecer o encoger en su parte posterior. Sólo se puede tener acceso a los elementos de una pila, y sólo es posible eliminarlos, desde la parte posterior. Ya ha visto características similares en los contenedores de secuencia, especialmente en `vector` y `deque`. De hecho, para implementar una pila se puede usar cualquier contenedor de secuencia que soporte las operaciones `back()`, `push_back()` y `pop_back()`. Los demás métodos de contenedores no se requieren para la pila y, por lo tanto, no se exponen aquí.

La clase de plantilla `stack` está diseñada para contener cualquier tipo de objetos. La única restricción es que todos los elementos deben ser del mismo tipo.

Una pila es una estructura *LIFO* (*Último en Entrar, Primero en Salir*). Es como un elevador atestado de gente: la primera persona que entra es empujada hacia la pared, y la última persona está parada justo frente a la puerta. Cuando el elevador llega al piso destinado, la última persona en entrar es la primera en salir. Si alguien quiere salir del elevador antes, todos los que están entre esa persona y la puerta deben quitarse del paso, probablemente saliendo del elevador y luego entrando otra vez.

Por convención, el extremo abierto de una pila se llama *tope* de la pila, y las operaciones que se realizan en una pila se llaman *push* (empujar) y *pop* (sacar). La clase `stack` hereda estos términos convencionales.

**Nota**

La clase `stack` de la STL no es la misma que el mecanismo de pila utilizado internamente por los compiladores y los sistemas operativos, en el que las pilas pueden contener distintos tipos de objetos. Sin embargo, la funcionalidad fundamental es muy similar.

## Colas

Una *cola* es otra estructura de datos utilizada comúnmente en la programación de computadoras. Los elementos se agregan a la cola en un extremo y se sacan por el otro. La analogía clásica es ésta: una pila es como un montón de platos apilados en una mesa. Se agrega a la pila colocando un plato encima (empujando la pila hacia abajo), y se quita de la pila “sacando” el plato de la parte superior (el que se agregó más recientemente a la pila).

Una cola es como una fila en el cine. Se entra a la cola por atrás, y se sale de la cola por enfrente. Esto se conoce como estructura *FIFO* (*Primero en Entrar, Primero en Salir*); una pila es una estructura *LIFO* (*Último en Entrar, Primero en Salir*). Claro que, de vez en cuando, usted está antes de la última persona en una larga fila del supermercado, cuando alguien abre una nueva caja y agarra a esa última persona de la línea, y convierte en una pila lo que debería de ser una cola, lo cual provoca que usted rechine los dientes de frustración.

Al igual que `stack`, `queue` (que es el nombre de la clase de tipo cola) se implementa como una clase de envoltura para un contenedor. El contenedor debe soportar las operaciones `front()`, `back()`, `push_back()` y `pop_front()`.

## Clases de algoritmos

Un contenedor es un lugar útil para guardar una secuencia de elementos. Todos los contenedores estándar definen operaciones que manipulan los contenedores y sus elementos. Sin embargo, implementar todas estas operaciones en sus propias secuencias puede ser laborioso y propenso a errores. Debido a que la mayoría de estas operaciones probablemente sean las mismas en casi todas las secuencias, un conjunto de algoritmos genéricos puede reducir la necesidad de escribir sus propias operaciones para cada contenedor nuevo. La biblioteca estándar proporciona aproximadamente 60 algoritmos estándar que realizan las operaciones más básicas y más utilizadas de los contenedores.

Los algoritmos estándar se definen en `<algorithm>`, el cual se encuentra en el espacio de nombres `std`.

Para comprender la forma en que funcionan los algoritmos, necesita conocer el concepto de los objetos de funciones. Un objeto de función es una instancia de una clase que define al operador `()` sobrecargado. Por lo tanto, se puede llamar como una función. El listado 19.11 muestra un objeto de función.

**ENTRADA****LISTADO 19.11** Un objeto de función

```

1: // Listado 19.11: Un objeto de función
2:
3: #include <iostream.h>
4:
5: using namespace std;
6:
7:
8: template< class T >
9: class Imprimir
10: {
11: public:
12:     void operator()(const T & t)
13:         { cout << t << " "; }
14: };
15:
16: int main()
17: {
18:     Imprimir< int > HacerImpresion;
19:
20:     for (int i = 0; i < 5; ++i)
21:         HacerImpresion(i);
22:     cout << endl;
23:     return 0;
24: }
```

**SALIDA**

0 1 2 3 4

**ANÁLISIS**

En las líneas 8 a 14 se define la clase de plantilla `Imprimir`. El operador () sobrecargado de las líneas 12 y 13 toma un objeto y lo envía a la salida estándar.

En la línea 18, `HacerImpresion` se define como una instancia de la clase `Imprimir`. Así, puede utilizar `HacerImpresion` igual que una función para imprimir cualquier valor entero, como se muestra en la línea 21.

19

**Algoritmos de secuencia no mutante**

Los algoritmos de secuencia no mutante no cambian los elementos de una secuencia. Esto incluye los operadores `for_each()`, `find()`, `search()`, `count()`, entre otros. El listado 19.12 muestra cómo utilizar un objeto de función y el algoritmo `for_each` para imprimir los elementos de un vector:

**ENTRADA****LISTADO 19.12** Uso del algoritmo `for_each()`

```

1: // Listado 19.12: Uso de for_each
2:
3: #include <iostream>
4: #include <vector>
5: #include <algorithm>
6:
```

continúa

**LISTADO 19.12** CONTINUACIÓN

```

7:     using namespace std;
8:
9:
10:    template< class T >
11:    class Imprimir
12:    {
13:    public:
14:        void operator()(const T & t)
15:            { cout << t << " "; }
16:    };
17:
18:    int main()
19:    {
20:        Imprimir< int > HacerImpresion;
21:        vector< int > vInt(5);
22:
23:        for (int i = 0; i < 5; ++i)
24:            vInt[ i ] = i * 3;
25:        cout << "for_each()\n";
26:        for_each(vInt.begin(), vInt.end(), HacerImpresion);
27:        cout << "\n";
28:        return 0;
29:    }

```

**SALIDA**

```
for_each()
0 3 6 9 12
```

**ANÁLISIS**

Observe que todos los algoritmos estándar de C++ se definen en `<algorithm>`, por lo que debemos incluir este archivo aquí. La mayor parte del programa debe ser fácil de entender para usted. En la línea 26 se llama a la función, o algoritmo, `for_each()` para que pase por todos los elementos del vector `vInt`. Para cada elemento, el algoritmo invoca al objeto de función `HacerImpresion` y pasa el elemento a `HacerImpresion.operator()`. Esto ocasiona que el valor del elemento se imprima en la pantalla.

## Algoritmos de secuencia mutante

Los algoritmos de secuencia mutante realizan operaciones que cambian los elementos de una secuencia, incluyendo operaciones que llenan o reordenan colecciones. El listado 19.13 muestra el algoritmo `fill()`.

**LISTADO 19.13** Un algoritmo de secuencia mutante

```

1: // Listado 19.13: Secuencia mutante
2:
3: #include <iostream>
4: #include <vector>
5: #include <algorithm>

```

```
6:  
7:    using namespace std;  
8:  
9:  
10:   template< class T >  
11:   class Imprimir  
12:   {  
13:       public:  
14:           void operator()(const T & t)  
15:               { cout << t << " "; }  
16:       };  
17:  
18:   int main()  
19:   {  
20:       Imprimir< int > HacerImpresion;  
21:       vector< int > vInt(10);  
22:  
23:       fill(vInt.begin(), vInt.begin() + 5, 1);  
24:       fill(vInt.begin() + 5, vInt.end(), 2);  
25:       for_each(vInt.begin(), vInt.end(), HacerImpresion);  
26:       cout << "\n";  
27:       return 0;  
28:   }
```

**SALIDA**

1 1 1 1 1 2 2 2 2 2

**ANÁLISIS**

El único contenido nuevo en este listado se encuentra en las líneas 23 y 24, en donde se utiliza el algoritmo `fill()`. Este algoritmo llena los elementos de una secuencia con un valor dado. En la línea 23 asigna el valor entero 1 a los primeros cinco elementos de `vInt`. En la línea 24 asigna el entero 2 a los últimos cinco elementos de `vInt`.

19

## Resumen

Hoy aprendió a crear y utilizar plantillas. Las plantillas son una comodidad integrada en C++, utilizadas para crear tipos parametrizados (tipos que cambian su comportamiento con base en los parámetros que reciben al momento de su creación). Son una manera de reutilizar código en forma segura y efectiva.

La definición de la plantilla determina el tipo parametrizado. Cada instancia de la plantilla es un objeto en sí, el cual se puede utilizar igual que cualquier otro objeto: como parámetro para una función, como valor de retorno, etcétera.

Las clases de plantillas pueden declarar tres tipos de funciones amigas: que no sean de plantilla, de plantilla general y de plantilla de tipo específico. Una plantilla puede declarar datos miembro estáticos, en cuyo caso cada instancia de la plantilla tiene su propio conjunto de datos estáticos.

Si necesita especializar el comportamiento para algunas funciones de plantilla con base en el tipo actual, puede redefinir una función de plantilla con un tipo específico. Esto también funciona para las funciones miembro.

## Preguntas y respuestas

- P ¿Por qué utilizar plantillas cuando se tienen las macros?**
- R Las plantillas ofrecen seguridad de tipos y están integradas en el lenguaje.**
- P ¿Cuál es la diferencia entre el tipo parametrizado de una función de plantilla y los parámetros para una función normal?**
- R Una función normal (que no sea de plantilla) toma parámetros sobre los cuales puede realizar alguna acción. Una función de plantilla le permite parametrizar el tipo de un parámetro específico para la función. Es decir, puede pasar un Arreglo de Tipo a una función y luego determinar el Tipo mediante la instancia de la plantilla.**
- P ¿Cuándo se deben utilizar las plantillas y cuándo se debe utilizar la herencia?**
- R Utilice plantillas cuando todo el comportamiento, o casi todo el comportamiento, permanezca sin cambio, excepto en relación con el tipo del elemento sobre el que actúa su clase. Si tiene que copiar una clase y cambiar sólo el tipo de uno o más de sus miembros, tal vez sea un buen momento para considerar el uso de una plantilla.**
- P ¿Cuándo se deben utilizar clases amigas de plantillas generales?**
- R Cuando cada instancia, sin importar el tipo, deba ser amiga para esta clase o función.**
- P ¿Cuándo se deben utilizar funciones o clases amigas de plantillas de tipo específico?**
- R Cuando quiera establecer una relación exacta entre dos clases. Por ejemplo, `array<int>` debería de concordar con `iterator<int>`, pero no con `iterator<Animal>`.**
- P ¿Cuáles son los dos tipos de contenedores estándar?**
- R Los contenedores de secuencia y los contenedores asociativos. Los contenedores de secuencia proporcionan un acceso secuencial y aleatorio optimizado para sus elementos. Los contenedores asociativos proporcionan un acceso optimizado a los elementos por medio de claves.**
- P ¿Qué atributos debe tener su clase para utilizarla con los contenedores estándar?**
- R La clase debe definir un constructor predeterminado, un constructor de copia y un operador de asignación sobrecargado.**

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Cuál es la diferencia entre una plantilla y una macro?
2. ¿Cuál es la diferencia entre el parámetro de una plantilla y el parámetro de una función?
3. ¿Cuál es la diferencia entre una clase amiga de plantilla de tipo específico y una clase amiga de plantilla general?
4. ¿Es posible proporcionar un comportamiento especial para una instancia de una plantilla, pero no para otras instancias?
5. ¿Cuántas variables estáticas se crean si se coloca un miembro estático en la definición de una clase de plantilla?
6. ¿Qué son los iteradores de la biblioteca estándar de C++?
7. ¿Qué es un objeto de función?

### Ejercicios

1. Cree una plantilla con base en esta clase `Lista`:

```
class Lista
{
private:

public:
    Lista():cabeza(0),cola(0),laCuenta(0) {}
    virtual ~Lista();
    void insertar(int valor);
    void agregar(int valor);
    int esta_presente(int valor) const;
    int esta_vacia() const { return cabeza == 0; }
    int cuenta() const { return laCuenta; }

private:
    class CeldaLista
    {
    public:
        CeldaLista(int valor, CeldaLista *celda =):val(valor),si-
guiente(cel){}
```

19

```
        int val;
        CeldaLista *siguiente;
    };
    CeldaLista *cabeza;
    CeldaLista *cola;
    int laCuenta;
};
```

2. Escriba la implementación para la versión (que no sea de plantilla) de la clase **Lista**.
3. Escriba la versión de plantilla de las implementaciones.
4. Declare tres objetos de tipo lista: una lista de objetos **Cadena**, una lista de objetos **Gato** y una lista de valores de tipo **int**.
5. **CAZA ERRORES:** ¿Qué está mal en el siguiente código? (Suponga que la plantilla **Lista** está definida y que **Gato** es la clase que se definió anteriormente en el libro.)  

```
Lista<Gato> Lista_Gato;
Gato Felix;
ListaGato.agregar(Felix);
cout << "Felix" <<
    (Lista_Gato.está_presente(Felix)) ? "" : "no" << " está
presente\n";
```

PISTA (esto está difícil): ¿Qué hace a **Gato** diferente de **int**?
6. Declare el **operator==** amigo para **Lista**.
7. Implemente el **operator==** amigo para **Lista**.
8. ¿Tiene **operator==** el mismo problema que en el ejercicio 5?
9. Implemente una función de plantilla para intercambiar dos variables.
10. Implemente a **ClaseEscuela** del listado 19.8 como una lista. Utilice la función **push\_back()** para agregar cuatro estudiantes a la lista. Luego desplácese por la lista resultante e incremente en uno la edad de cada estudiante.
11. Modifique el ejercicio 10 para utilizar un objeto de función para desplegar el registro de cada estudiante.

# SEMANA 3

DÍA 20

## Excepciones y manejo de errores

El código que ha visto en este libro se ha creado para propósitos ilustrativos. No trata sobre los errores para que usted no se distraiga de los temas principales que se están presentando. Los programas del mundo real tienen que considerar las condiciones de error.

Hoy aprenderá lo siguiente:

- Qué son las excepciones
- Cómo se utilizan las excepciones, y qué cuestiones surgen debido a ellas
- Cómo crear jerarquías de excepciones
- Cómo encajan las excepciones en un método general para manejo de errores
- Qué es un depurador

## Bugs y corrupción de código

Todos los programas tienen bugs (errores). Entre más grande sea el programa tendrá más bugs, y muchos de éstos irán en el software final y liberado. Que esto sea verdad no quiere decir que esté bien, pues hacer programas robustos y libres de bugs es la principal prioridad de cualquiera que considere seriamente la programación.

El principal problema en la industria del software es el código inestable y lleno de bugs. El mayor gasto en muchos esfuerzos importantes de programación es probar y arreglar. La persona que resuelva el problema de producir programas buenos, sólidos, a prueba de todo, a un bajo costo y a tiempo, revolucionará la industria del software.

Hay diversos tipos de bugs que pueden ocasionar problemas en un programa. El primero es una mala lógica: el programa hace lo que usted le pide, pero usted no ha pensado apropiadamente todos los algoritmos. El segundo es de tipo sintáctico: utilizó el idioma, la función o la estructura incorrectos. Éstos dos son los más comunes, y son los que la mayoría de los programadores trata de evitar.

La investigación y la experiencia en el mundo real han mostrado que, sin lugar a dudas, entre más tarde se encuentre un problema durante el proceso de desarrollo, más costará arreglarlo. Los problemas menos costosos o los bugs más fáciles de arreglar son los que usted evita crear. Los que siguen en menor costo son los que encuentra el compilador. Los estándares de C++ hacen que los compiladores inviertan mucha energía para lograr que aparezcan más bugs en tiempo de compilación.

Los bugs que se compilan pero que se detectan en la primera prueba (aquellos que ocasionan fallas en todo momento) son menos difíciles de encontrar y arreglar que los que aparecen sólo de vez en cuando.

Un problema mayor que los bugs lógicos o sintácticos es la fragilidad innecesaria: el programa funciona perfectamente si el usuario escribe un número cuando se le pide, pero falla si el usuario escribe letras. Otros programas fallan si se acaba la memoria, o si el disquete no está dentro de la unidad, o si el módem pierde la conexión.

Para combatir este tipo de fragilidad, los programadores se esfuerzan por hacer sus programas a prueba de todo. Un programa *a prueba de todo* es aquel que puede manejar cualquier cosa que surja en tiempo de ejecución, desde una entrada rara por parte del usuario hasta el agotamiento de la memoria. Otro término para este proceso es programación *a la defensiva*, que al igual que la conducción a la defensiva, significa estar preparado para lo inesperado.

Es importante distinguir entre los bugs, que aparecen debido a que el programador se equivocó en la sintaxis; los errores lógicos, que surgen porque el programador entendió mal el problema o la forma de resolverlo; y las excepciones, que surgen debido a problemas inusuales pero predecibles, como cuando se agotan los recursos (memoria o espacio en disco).

## Excepciones

Los programadores utilizan compiladores poderosos y rocían su código con aserciones para atrapar bugs de programación (las aserciones se explican en el día 21, “Qué sigue”). También realizan revisiones de diseño y pruebas exhaustivas para encontrar errores lógicos.

Sin embargo, las excepciones son distintas. Usted no puede eliminar circunstancias excepcionales; sólo puede estar preparado para ellas. A sus usuarios se les agotará la memoria de vez en cuando, y la única pregunta es qué es lo que usted hará. Sus opciones están limitadas a las siguientes:

- Hacer que el programa falle
- Informar al usuario y salir con elegancia
- Informar al usuario y permitir que trate de recuperarse y continuar
- Tomar una acción correctiva y continuar sin molestar al usuario

Aunque no es necesario, ni deseable, que cualquier programa que escriba se recupere automáticamente y silenciosamente de todas las circunstancias excepcionales, está claro que debe hacer algo mejor que fallar.

El manejo de excepciones de C++ proporciona un método integrado con seguridad de tipos para hacer frente a las condiciones predecibles pero inusuales que surgen al ejecutar un programa.

## Unas palabras acerca de la corrupción del código

La corrupción del código es un fenómeno perfectamente probado en el que el software se deteriora debido a la negligencia. Un programa bien escrito y completamente depurado se echará a perder en la repisa de su cliente unas cuantas semanas después de su entrega. Después de unos cuantos meses, su cliente se dará cuenta de que un moho verde ha cubierto su lógica, y muchos de sus objetos han empezado a desprenderse.

Además de enviar su código fuente en contenedores sellados herméticamente, su única protección es escribir sus programas de forma que cuando regrese a arreglar lo estropeado, pueda identificar rápida y fácilmente en donde se encuentran los problemas.

### Nota

La corrupción del código es algo así como una broma del programador, utilizada para explicar cómo un código supuestamente libre de errores, de repente se vuelve inestable. Sin embargo, esto enseña una lección importante. Los programas son muy complejos, y los bugs pueden permanecer escondidos por mucho tiempo antes de aparecer. Protéjase usted mismo escribiendo código fácil de mantener.

Un término similar se ha aplicado a los libros impresos. Sin importar qué tan cuidadosos sean el autor, los revisores técnicos, los editores y los correctores, los errores aparecen. Pero parece que entre más se lee el libro (cuando éste se encuentra ya en el mercado), los errores aparecen con más frecuencia.  
¡Qué le parece!

Esto significa que su código debe llevar comentarios, aun si no espera que nadie más lo vaya a analizar. Seis meses después de entregar su código, lo leerá con los ojos de un completo extraño, y se preguntará cómo pudo alguien escribir un código tan complicado y retorcido, y esperar algo que no fuera un desastre.

## Excepciones

En C++, una excepción es un objeto que se pasa desde el área del código en la que ocurre un problema hasta la parte del código que se va a encargar del problema. El tipo de la excepción determina cuál área de código se encargará del problema, y el contenido del objeto enviado, si lo hay, se puede utilizar para retroalimentar al usuario.

La idea básica de las excepciones es bastante clara:

- La asignación de los recursos (por ejemplo, la asignación de memoria o el bloqueo de un archivo) por lo general se hace a un nivel muy bajo en el programa.
- La lógica de lo que se debe hacer cuando falla una operación (no se puede asignar la memoria o no se puede bloquear un archivo) se encuentra por lo general en un nivel alto en el programa, con el código para interactuar con el usuario.
- Las excepciones proporcionan un camino rápido que va del código que asigna los recursos hasta el código que puede manejar la condición de error. Si existen niveles intermedios de funciones, se les da una oportunidad para limpiar las asignaciones de memoria, pero no se les pide que incluyan código cuyo único propósito sea pasar más adelante la condición de error.

## Cómo se utilizan las excepciones

Los bloques `try` se crean para rodear áreas de código que puedan tener un problema. Por ejemplo:

```
try
{
    UnaFuncionPeligrosa();
}
```

Los bloques `catch` manejan las excepciones producidas en el bloque `try`. Por ejemplo:

```
try
{
    UnaFuncionPeligrosa();
}
catch(NoHayMemoria)
{
    // realizar algunas acciones
}
catch(FileNotFound)
{
```

```
// realizar otra acción  
}
```

Los pasos básicos para utilizar las excepciones son los siguientes:

1. Identificar aquellas áreas del programa en las que se empieza una operación que podría provocar una excepción, y colocarlas en bloques `try`.
2. Crear bloques `catch` para atrapar las excepciones en caso de que se produzcan, limpiar la memoria asignada e informar al usuario según sea apropiado. El listado 20.1 muestra el uso de los bloques `try` y `catch`.

Las excepciones son objetos que se utilizan para transmitir información acerca de un problema.

Un bloque `try` está entre llaves, y en él se puede producir una excepción.

Un bloque `catch` sigue inmediatamente después de un bloque `try`, y en él se manejan las excepciones.

Cuando se produce (o surge) una excepción, el control se transfiere al bloque `catch` que sigue inmediatamente después del bloque `try` actual.

### Nota

Algunos compiladores muy antiguos, como el GNU 2.7.2, no soportan las excepciones. Sin embargo, éstas son parte del estándar ANSI de C++, y las ediciones más recientes de compiladores de cualquier fabricante soportan completamente las excepciones nativas. Si usted tiene un compilador más antiguo, no podrá compilar y ejecutar los ejercicios de esta lección. Sin embargo, sería conveniente que leyera toda la lección, y que regresara a este material cuando actualice su compilador.

Para estos ejemplos, utilice la versión de compilador que viene en el CD-ROM (2.9.5).

### ENTRADA LISTADO 20.1 Cómo se produce una excepción

```
1: // listado 20.1: Cómo se produce una excepción  
2:  
3: #include <iostream.h>  
4:  
5: const int TamanioPred = 10;  
6:  
7:  
8: class Arreglo  
9: {  
10: public:  
11:     // constructores  
12:     Arreglo(int suTamanio = TamanioPred);  
13:     Arreglo(const Arreglo & rhs);  
14:     ~Arreglo()
```

20

**LISTADO 20.1** CONTINUACIÓN

```
15:         { delete [] apTipo; }
16:         // operadores
17:         Arreglo & operator=(const Arreglo &);
18:         int & operator[](int desplazamiento);
19:         const int & operator[](int desplazamiento) const;
20:         // métodos de acceso
21:         int ObtenerSuTamanio() const
22:         { return suTamanio; }
23:         // función amiga
24:         friend ostream & operator<< (ostream &, const Arreglo &);
25:         // definir la clase de excepción
26:         class xLímite {};
27:     private:
28:         int *apTipo;
29:         int suTamanio;
30:     };
31:
32:
33:     Arreglo::Arreglo(int tamanio):
34:     suTamanio(tamanio)
35:     {
36:         apTipo = new int[ tamanio ];
37:
38:         for (int i = 0; i < tamanio; i++)
39:             apTipo[ i ] = 0;
40:     }
41:
42:
43:     Arreglo & Arreglo::operator=(const Arreglo & rhs)
44:     {
45:         if (this == &rhs)
46:             return *this;
47:         delete [] apTipo;
48:         suTamanio = rhs.ObtenerSuTamanio();
49:         apTipo = new int[ suTamanio ];
50:         for (int i = 0; i < suTamanio; i++)
51:             apTipo[ i ] = rhs[ i ];
52:         return *this;
53:     }
54:
55:     Arreglo::Arreglo(const Arreglo & rhs)
56:     {
57:         suTamanio = rhs.ObtenerSuTamanio();
58:         apTipo = new int[ suTamanio ];
59:
60:         for (int i = 0; i < suTamanio; i++)
61:             apTipo[ i ] = rhs[ i ];
62:     }
63:
64:     int & Arreglo::operator[](int desplazamiento)
65:     {
66:         int tamanio = ObtenerSuTamanio();
67:
```

```
68:         if (desplazamiento >= 0 && desplazamiento < ObtenerSuTamanio())
69:             return apTipo[ desplazamiento ];
70:         throw xLimite();
71:         // apaciguar a MSC
72:         return apTipo[ 0 ];
73:     }
74:
75:
76:     const int & Arreglo::operator[](int desplazamiento) const
77:     {
78:         int mitamanio = ObtenerSuTamanio();
79:
80:         if (desplazamiento >= 0 && desplazamiento < ObtenerSuTamanio())
81:             return apTipo[ desplazamiento ];
82:         throw xLimite();
83:         // apaciguar a MSC
84:         return apTipo[ 0 ];
85:     }
86:
87:     ostream & operator<< (ostream & salida, const Arreglo & elArreglo)
88:     {
89:         for (int i = 0; i < elArreglo.ObtenerSuTamanio(); i++)
90:             salida << "[" << i << "] " << elArreglo[ i ] << endl;
91:         return salida;
92:     }
93:
94:     int main()
95:     {
96:         Arreglo arregloInt(20);
97:
98:         try
99:         {
100:             for (int j = 0; j < 100; j++)
101:             {
102:                 arregloInt[ j ] = j;
103:                 cout << "arregloInt[" << j;
104:                 cout << "] está bien..." << endl;
105:             }
106:         }
107:         catch (Arreglo::xLimite)
108:         {
109:             cout << "¡No se pudo procesar su entrada!\n";
110:         }
111:         cout << "Listo.\n";
112:         return 0;
113:     }
```

20

**SALIDA**

```
arregloInt[0] está bien...
arregloInt[1] está bien...
arregloInt[2] está bien...
arregloInt[3] está bien...
arregloInt[4] está bien...
arregloInt[5] está bien...
```

```
arregloInt[6] está bien...
arregloInt[7] está bien...
arregloInt[8] está bien...
arregloInt[9] está bien...
arregloInt[10] está bien...
arregloInt[11] está bien...
arregloInt[12] está bien...
arregloInt[13] está bien...
arregloInt[14] está bien...
arregloInt[15] está bien...
arregloInt[16] está bien...
arregloInt[17] está bien...
arregloInt[18] está bien...
arregloInt[19] está bien...
¡No se pudo procesar su entrada!
Listo.
```

**ANÁLISIS**

El listado 20.1 presenta una clase Arreglo algo simplificada, basada en la plantilla desarrollada en el día 19, “Plantillas”.

En la línea 26 se declara una nueva clase llamada `xLímite`, dentro de la declaración de la clase Arreglo externa.

Esta nueva clase no se distingue de ninguna forma como una clase de excepción. Es sólo una clase como cualquier otra. Esta singular clase es muy simple; no tiene datos ni métodos. No obstante, es una clase válida en todos los aspectos.

De hecho, es incorrecto decir que no tiene métodos, ya que el compilador le asigna automáticamente un constructor predeterminado, un constructor de copia y el operador de asignación (operador igual a); así que en realidad tiene cuatro funciones, pero ningún dato.

Observe que declarar esta clase desde el interior de Arreglo sirve sólo para acoplar las dos clases entre sí. Como se explicó en el día 15, “Herencia avanzada”, Arreglo no tiene acceso especial a `xLímite`, ni `xLímite` tiene acceso preferencial a los miembros de Arreglo.

En las líneas 64 a 73 y 76 a 85 se modifican los operadores de desplazamiento para examinar el desplazamiento solicitado, y si está fuera de rango, para producir la clase `xLímite` como una excepción. Para distinguir entre esta llamada al constructor de `xLímite` y entre el uso de una constante enumerada, se requieren los paréntesis. Hay que tener en cuenta que algunos compiladores de Microsoft requieren que se proporcione una instrucción `return` para concordar con la declaración (en este caso, regresar una referencia a un entero); aunque se produzca una excepción en la línea 70, el código nunca llegará a la línea 72. Éste es un error del compilador, lo que prueba que ¡incluso Microsoft encuentra esto difícil y confuso!

En la línea 98, la palabra reservada `try` empieza un bloque `try` que termina en la línea 106. Dentro de ese bloque `try`, se agregan 101 enteros al arreglo que se declaró en la línea 96.

En la línea 107 se declara el bloque `catch` para atrapar las excepciones `xLímite`.

En el programa controlador de las líneas 94 a 113 se crea un bloque try en el que se inicializa cada miembro del arreglo. Cuando j (línea 100) se incrementa a 20, se accede al miembro que se encuentra en el desplazamiento 20. Esto ocasiona que falle la prueba de la línea 68, y operator[] produce una excepción xLímite en la línea 70.

El control del programa se va al bloque catch de la línea 107, y el bloque catch atrapa o maneja la excepción en la misma línea, la cual imprime un mensaje de error. El flujo del programa baja hasta el final del bloque catch en la línea 111.

#### Bloques try

Un bloque try es una serie de instrucciones que empiezan con la palabra reservada try, seguida de una llave de apertura y terminan con una llave de cierre.

He aquí un ejemplo:

```
try
{
    Funcion();
}
```

#### Bloques catch

Un bloque catch es una serie de instrucciones, cada una de las cuales empieza con la palabra reservada catch, seguida de un tipo de excepción entre paréntesis, una llave de apertura y una llave de cierre.

He aquí un ejemplo:

```
try
{
    Funcion();
}
catch (NoHayMemoria)
{
    // realizar una acción
}
```

## Uso de los bloques try y catch

No es fácil entender en dónde debe poner sus bloques try: no siempre es obvio qué acciones pueden provocar una excepción. La siguiente pregunta es en dónde atrapar la excepción. Tal vez quiera producir todas las excepciones de memoria en donde se asigna la memoria, pero quiera atrapar las excepciones en un nivel alto en el programa en el que trate con la interfaz de usuario.

Al tratar de determinar las ubicaciones de los bloques `try`, busque en donde asigna memoria o utiliza recursos. También puede buscar en los errores fuera de los límites, entradas ilegales, etc.

## Cómo atrapar excepciones

El proceso de atrapar una excepción funciona así: cuando se produce una excepción, se examina la pila de llamadas. Esta pila es la lista de llamadas a funciones que se crea cuando una parte del programa invoca a otra función.

La pila de llamadas rastrea la ruta de ejecución. Si `main()` llama a la función `Animal::ObtenerComidaFavorita()`, y ésta llama a `Animal::BuscarPreferencias()`, que a su vez llama a `fstream::operator>>()`, todas estas llamadas se encuentran en la pila de llamadas. Una función recursiva podría estar en la pila de llamadas muchas veces.

La excepción se pasa de la pila de llamadas a cada bloque que la rodea. A medida que se eliminan elementos de la pila, se invocan los destructores para los objetos locales que están en ella, y se destruyen dichos objetos.

Después de cada bloque `try` hay una o más instrucciones `catch`. Si la excepción concuerda con una de las instrucciones `catch`, se considera que se va a manejar al ejecutar esa instrucción. Si no concuerda con ninguna instrucción, continúa la eliminación de elementos de la pila.

Si la excepción llega hasta el comienzo del programa (`main()`) y aún no es atrapada, se llama a un manejador integrado que termina el programa.

Es importante observar que la búsqueda de bloques que manejan la excepción se realiza en un solo sentido. A medida que va progresando, la pila se reduce y los objetos que están en ella se destruyen. No hay regreso: después de encargarse de la excepción, el programa continúa en la instrucción `catch` que se encargó de la excepción.

Por ejemplo, en el listado 20.1 la ejecución continuará en la línea 109, y después en la 111. Es decir, en la primera línea dentro de la instrucción `catch` que se encargó de la excepción `xLímite`, y posteriormente al final del conjunto de bloques `catch`. Recuerde que cuando se produce una excepción, el flujo del programa continúa después del bloque `catch`, no después del punto en el que se produjo la excepción.

## Más de una especificación `catch`

Es posible que dos o más condiciones produzcan una excepción. En este caso, las instrucciones `catch` se pueden alinear una después de otra, en forma muy parecida a las condiciones de una instrucción `switch`. El equivalente para la instrucción predeterminada es la instrucción “atrapsar todo”, la cual se indica escribiendo `catch(...)`. El listado 20.2 muestra condiciones de excepciones múltiples.

**ENTRADA LISTADO 20.2 Excepciones múltiples**

```
1: // Listado 20.2: Excepciones múltiples
2:
3: #include <iostream.h>
4:
5: const int TamanioPredet = 10;
6:
7:
8: class Arreglo
9: {
10: public:
11:     // constructores
12:     Arreglo(int suTamanio = TamanioPredet);
13:     Arreglo(const Arreglo & rhs);
14:     ~Arreglo()
15:     { delete [] apTipo; }
16:     // operadores
17:     Arreglo & operator=(const Arreglo &);

18:     int & operator[](int desplazamiento);
19:     const int & operator[](int desplazamiento) const;
20:     // métodos de acceso
21:     int ObtenerSuTamanio() const
22:     { return suTamanio; }
23:     // función amiga
24:     friend ostream & operator<< (ostream &, const Arreglo &);

25:     // definir las clases de excepciones
26:     class xLimite {};
27:     class xMuyGrande {};
28:     class xMuyChico {};
29:     class xCero {};
30:     class xNegativo {};
31: private:
32:     int *apTipo;
33:     int suTamanio;
34: };
35:
36:     int & Arreglo::operator[](int desplazamiento)
37:     {
38:         int tamanio = ObtenerSuTamanio();
39:
40:         if (desplazamiento >= 0 && desplazamiento <
41:             suTamanio)
42:             return apTipo[ desplazamiento ];
43:         throw xLimite();
44:         // apaciguar a MFC
45:         return apTipo[ 0 ];
46:     }
47:
48:     const int & Arreglo::operator[](int desplazamiento) const
49:     {
```

20

*continúa*

**LISTADO 20.2** CONTINUACIÓN

```
50:             int mitamanio = ObtenerSuTamanio();
51:
52:             if (desplazamiento >= 0 && desplazamiento <
53:                 ➔ObtenerSuTamanio())
54:                 return apTipo[ desplazamiento ];
55:             throw xLimite();
56:             // apaciguar a MFC
57:             return apTipo[ 0 ];
58:
59:         Arreglo::Arreglo(int tamano):
60:             suTamanio(tamano)
61:         {
62:             if (tamano == 0)
63:                 throw xCero();
64:             if (tamano < 10)
65:                 throw xMuyChico();
66:             if (tamano > 30000)
67:                 throw xMuyGrande();
68:             if (tamano < 0)
69:                 throw xNegativo();
70:             apTipo = new int[ tamano ];
71:             for (int i = 0; i < tamano; i++)
72:                 apTipo[ i ] = 0;
73:         }
74:
75:     int main()
76:     {
77:         try
78:         {
79:             Arreglo arregloInt(0);
80:
81:             for (int j = 0; j < 100; j++)
82:             {
83:                 arregloInt[ j ] = j;
84:                 cout << "arregloInt[" << j << "] está bien...\n";
85:             }
86:         }
87:         catch (Arreglo::xLimite)
88:         {
89:             cout << "¡No se pudo procesar su entrada!\n";
90:         }
91:         catch (Arreglo::xMuyGrande)
92:         {
93:             cout << "Este arreglo es muy grande...\n";
94:         }
95:         catch (Arreglo::xMuyChico)
96:         {
97:             cout << "Este arreglo es muy chico...\n";
```

```

98:      }
99:      catch (Arreglo::xCero)
100:     {
101:       cout << "¡Pidió un arreglo"
102:       cout << " de cero objetos!\n";
103:     }
104:     catch (...)
105:     {
106:       cout << "¡Algo salió mal!\n";
107:     }
108:     cout << "Listo.\n";
109:   return 0;
110: }

```

**SALIDA** ¡Pidió un arreglo de cero objetos!  
Listo.

**ANÁLISIS** En las líneas 27 a 30 se crean cuatro nuevas clases: xMuyGrande, xMuyChico, xCero y xNegativo. El constructor, líneas 59 a 73, examina el tamaño que se le pasa. Si es muy grande, muy pequeño, negativo o cero se produce una excepción.

El bloque `try` se cambia para incluir instrucciones `catch` para cada condición que no sea un número negativo, la cual es atrapada por medio de la instrucción `catch(...)` ("atrapar todo"), que se muestra en la línea 104.

Pruebe esto con varios valores para el tamaño del arreglo. Luego trate de colocar un -5. Tal vez haya esperado que se llamara a xNegativo, pero el orden de las pruebas del constructor previno esto: `tamanio < 10` se evaluó antes de `tamanio < 0`. Para arreglar esto, intercambie las líneas 64 y 65 con las líneas 68 y 69 y vuelva a compilar.

## Jerarquías de excepciones

Las excepciones son clases, y como tales, se pueden hacer derivaciones de ellas. Puede ser ventajoso crear una clase xTamanio, y derivar de ésta a xCero, xMuyChico, xMuyGrande y xNegativo. Por ejemplo, algunas funciones podrían simplemente atrapar errores de tipo xTamanio, y otras funciones podrían atrapar el tipo específico de error xTamanio. El listado 20.3 muestra esta idea.

20

**ENTRADA** **LISTADO 20.3** Jerarquías de clases y excepciones

```

1: // Listado 20.3: Jerarquias de clases y excepciones
2:
3: #include <iostream.h>
4:
5: const int TamanioPredet = 10;
6:

```

**LISTADO 20.3** CONTINUACIÓN

```
7:
8:     class Arreglo
9:     {
10:    public:
11:        // constructores
12:        Arreglo(int suTamanio = TamanioPredet);
13:        Arreglo(const Arreglo & rhs);
14:        ~Arreglo()
15:        { delete [] apTipo; }
16:        // operadores
17:        Arreglo & operator=(const Arreglo &);

18:        int & operator[](int desplazamiento);
19:        const int & operator[](int desplazamiento) const;
20:        // métodos de acceso
21:        int ObtenerSuTamanio() const
22:        { return suTamanio; }
23:        // función amiga
24:        friend ostream & operator<< (ostream &, const Arreglo &);

25:        // definir las clases de excepciones
26:        class xLímite {};
27:        class xTamanio {};
28:        class xMuyGrande : public xTamanio {};
29:        class xMuyChico : public xTamanio {};
30:        class xCero : public xMuyChico {};
31:        class xNegativo : public xTamanio {};
32:    private:
33:        int *apTipo;
34:        int suTamanio;
35:    };
36:
37:    Arreglo::Arreglo(int tamanio):
38:    suTamanio(tamanio)
39:    {
40:        if (tamanio == 0)
41:            throw xCero();
42:        if (tamanio > 30000)
43:            throw xMuyGrande();
44:        if (tamanio < 0)
45:            throw xNegativo();
46:        if (tamanio < 10)
47:            throw xMuyChico();
48:        apTipo = new int[ tamanio ];
49:        for (int i = 0; i < tamanio; i++)
50:            apTipo[ i ] = 0;
51:    }

52:
53:    int & Arreglo::operator[](int desplazamiento)
54:    {
```

```
55:         int tamano = ObtenerSuTamanio();
56:
57:         if (desplazamiento >= 0 && desplazamiento < ObtenerSuTamanio())
58:             return apTipo[ desplazamiento ];
59:         throw xLmite();
60:         // apaciguar a MFC
61:         return apTipo[ 0 ];
62:     }
63:
64:     const int & Arreglo::operator[](int desplazamiento) const
65:     {
66:         int mitamanio = ObtenerSuTamanio();
67:
68:         if (desplazamiento >= 0 && desplazamiento < ObtenerSuTamanio())
69:             return apTipo[ desplazamiento ];
70:         throw xLmite();
71:         // apaciguar a MFC
72:         return apTipo[ 0 ];
73:     }
74:
75:     int main()
76:     {
77:         try
78:         {
79:             Arreglo arregloInt(0);
80:
81:             for (int j = 0; j < 100; j++)
82:             {
83:                 arregloInt[ j ] = j;
84:                 cout << "arregloInt[" << j << "] está bien...\n";
85:             }
86:         }
87:         catch (Arreglo::xLmite)
88:         {
89:             cout << "¡No se pudo procesar su entrada!\n";
90:         }
91:         catch (Arreglo::xMuyGrande)
92:         {
93:             cout << "Este arreglo es muy grande...\n";
94:         }
95:         catch (Arreglo::xMuyChico)
96:         {
97:             cout << "Este arreglo es muy chico...\n";
98:         }
99:         catch (Arreglo::xCero)
100:        {
101:             cout << "¡Pidió un arreglo";
102:             cout << " de cero objetos!\n";
103:         }
104:     }
```

20

continúa

**LISTADO 20.3** CONTINUACIÓN

---

```

104:     catch (...)
105:     {
106:         cout << "¡Algo salió mal!\n";
107:     }
108:         cout << "Listo.\n";
109:         return 0;
110:     }

```

---

**SALIDA** Este arreglo es muy chico...  
Listo.

**ANÁLISIS** El cambio significativo se encuentra en las líneas 27 a 31, en donde se establece la jerarquía de clases. Las clases `xMuyGrande`, `xMuyChico` y `xNegativo` se derivan de `xTamanio`, y `xCero` se deriva de `xMuyChico`.

El Arreglo se crea con un tamaño de cero, pero, ¿qué ocurrió? ¡Parece que se atrapó la excepción equivocada! Sin embargo, examine cuidadosamente el bloque `catch`, y descubrirá que éste busca una excepción de tipo `xMuyChico` antes de buscar una excepción de tipo `xCero`. Como se produce un objeto `xCero`, y éste es un objeto `xMuyChico`, el manejador lo atrapa como `xMuyChico`. Después de manejar la excepción, ya no se pasa a los otros manejadores, por lo que nunca se llama al manejador de `xCero`.

La solución para este problema es ordenar cuidadosamente los manejadores, de forma que los manejadores más específicos vayan primero y los menos específicos después. En este ejemplo específico, si se cambia la ubicación de los manejadores `xCero` y `xMuyChico` se solucionará el problema.

## Acceso a los datos de excepciones mediante la denominación de objetos de excepciones

A menudo necesitará saber más que sólo el tipo de excepción que se produjo para poder responder apropiadamente al error. Las clases de excepciones son iguales que cualquier otra clase. Usted puede proporcionar datos, inicializar los datos en el constructor y leer esos datos en cualquier momento. El listado 20.4 muestra cómo hacer esto.

**ENTRADA** **LISTADO 20.4** Cómo sacar datos de un objeto de excepción

---

```

1: // Listado 20.4: Obtención de los datos de un objeto de excepción
2:
3: #include <iostream.h>
4:
5: const int TamanioPredet = 10;

```

```
6:
7:
8:     class Arreglo
9:     {
10:    public:
11:        // constructores
12:        Arreglo(int suTamanio = TamanioPredet);
13:        Arreglo(const Arreglo & rhs);
14:        ~Arreglo()
15:        { delete [] apTipo; }
16:        // operadores
17:        Arreglo & operator=(const Arreglo &);

18:        int & operator[](int desplazamiento);
19:        const int & operator[](int desplazamiento) const;
20:        // métodos de acceso
21:        int ObtenerSuTamanio() const
22:        { return suTamanio; }
23:        // función amiga
24:        friend ostream & operator<< (ostream &, const Arreglo &);

25:        // definir las clases de excepciones
26:        class xLimite {};
27:        class xTamanio
28:        {
29:            public:
30:                xTamanio(int tamanio) : suTamanio(tamanio) {}
31:                ~xTamanio(){}
32:                int ObtenerTamanio()
33:                { return suTamanio; }
34:            private:
35:                int suTamanio;
36:            };
37:            class xMuyGrande : public xTamanio
38:            {
39:                public:
40:                    xMuyGrande(int tamanio) : xTamanio(tamanio) {}
41:                };
42:            class xMuyChico : public xTamanio
43:            {
44:                public:
45:                    xMuyChico(int tamanio) : xTamanio(tamanio) {}
46:                };
47:            class xCero : public xMuyChico
48:            {
49:                public:
50:                    xCero(int tamanio) : xMuyChico(tamanio) {}
51:                };
52:            class xNegativo : public xTamanio
53:            {
54:                public:
55:                    xNegativo(int tamanio) : xTamanio(tamanio) {}
56:                };
57:            private:
```

**LISTADO 20.4** CONTINUACIÓN

```
58:             int *apTipo;
59:             int suTamanio;
60:         };
61:
62:
63:         Arreglo::Arreglo(int tamanio):
64:             suTamanio(tamanio)
65:         {
66:             if (tamanio == 0)
67:                 throw xCero(tamanio);
68:             if (tamanio > 30000)
69:                 throw xMuyGrande(tamanio);
70:             if (tamanio < 0)
71:                 throw xNegativo(tamanio);
72:             if (tamanio < 10)
73:                 throw xMuyChico(tamanio);
74:             apTipo = new int[ tamanio ];
75:             for (int i = 0; i < tamanio; i++)
76:                 apTipo[ i ] = 0;
77:         }
78:
79:         int & Arreglo::operator[] (int desplazamiento)
80:     {
81:         int tamanio = ObtenerSuTamanio();
82:
83:         if (desplazamiento >= 0 && desplazamiento < ObtenerSuTamanio())
84:             return apTipo[ desplazamiento ];
85:         throw xLimite();
86:         return apTipo[0];
87:     }
88:
89:         const int & Arreglo::operator[] (int desplazamiento) const
90:     {
91:         int tamanio = ObtenerSuTamanio();
92:
93:         if (desplazamiento >= 0 && desplazamiento < ObtenerSuTamanio())
94:             return apTipo[ desplazamiento ];
95:         throw xLimite();
96:         return apTipo[ 0 ];
97:     }
98:
99:         int main()
100:    {
101:        try
102:        {
103:            Arreglo arregloInt(9);
104:
```

```
105:         for (int j = 0; j < 100; j++)
106:         {
107:             arregloInt[ j ] = j;
108:             cout << "arregloInt[" << j << "] está bien..." << endl;
109:         }
110:     }
111:     catch (Arreglo::xLimite)
112:     {
113:         cout << "¡No se pudo procesar su entrada!\n";
114:     }
115:     catch (Arreglo::xCero laExcepcion)
116:     {
117:         cout << "¡Pidió un arreglo de cero objetos!" << endl;
118:         cout << "Se recibieron ";
119:         cout << laExcepcion.ObtenerTamanio() << endl;
120:     }
121:     catch (Arreglo::xMuyGrande laExcepcion)
122:     {
123:         cout << "Este Arreglo es muy grande..." << endl;
124:         cout << "Se recibieron ";
125:         cout << laExcepcion.ObtenerTamanio() << endl;
126:     }
127:     catch (Arreglo::xMuyChico laExcepcion)
128:     {
129:         cout << "Este Arreglo es muy chico..." << endl;
130:         cout << "Se recibieron ";
131:         cout << laExcepcion.ObtenerTamanio() << endl;
132:     }
133:     catch (...)
134:     {
135:         cout << "Algo salió mal,";
136:         cout << " pero no tengo idea de qué fue!\n";
137:     }
138:     cout << "Listo.\n";
139:     return 0;
140: }
```

**SALIDA**

Este arreglo es muy chico...  
Se recibieron 9  
Listo.

**ANÁLISIS**

La declaración de `xTamanio` se ha modificado para incluir una variable miembro llamada `suTamanio` (línea 35) y una función miembro llamada `ObtenerTamanio()` (línea 32). Además, se ha agregado un constructor que toma un entero e inicializa la variable miembro, como se muestra en la línea 30.

Las clases derivadas declaran un constructor que no hace nada excepto inicializar la clase base. No se declararon otras funciones, en parte para ahorrar espacio en el listado.

Las instrucciones `catch` de las líneas 115 a 132 están modificadas para nombrar la excepción que atrapan, `laExcepcion`, y para utilizar este objeto para tener acceso a los datos guardados en `suTamanio`.



### Nota

Tenga en mente que si está construyendo una excepción, es porque se ha producido una: algo ha salido mal, y su excepción debe tener cuidado de no activar el mismo problema. Por lo tanto, si está creando una excepción `NoHayMemoria`, lo más seguro será no asignar memoria en su constructor.

Es un proceso tedioso y propenso a errores hacer que cada una de estas instrucciones `catch` imprima individualmente el mensaje apropiado. Este trabajo pertenece al objeto, el cual sabe qué tipo de objeto es y qué valor recibió. El listado 20.5 utiliza un método más orientado a objetos para este problema, con funciones virtuales para que cada excepción “haga lo correcto”.

#### ENTRADA

#### **LISTADO 20.5** Paso de valores por referencia y uso de funciones virtuales en las excepciones

```
1: // Listado 20.5: Paso de valores por referencia en las excepciones
2:
3: #include <iostream.h>
4:
5: const int TamanioPredet = 10;
6:
7:
8: class Arreglo
9: {
10: public:
11:     // constructores
12:     Arreglo(int suTamanio = TamanioPredet);
13:     Arreglo(const Arreglo & rhs);
14:     ~Arreglo()
15:     { delete [] apTipo; }
16:     // operadores
17:     Arreglo & operator=(const Arreglo &);

18:     int & operator[](int desplazamiento);
19:     const int & operator[](int desplazamiento) const;
20:     // métodos de acceso
21:     int ObtenerSuTamanio() const
22:     { return suTamanio; }
23:     // función amiga
24:     friend ostream & operator<<(ostream &, const Arreglo &);

25:     // definir las clases de excepciones
26:     class xLimite {};
27:     class xTamanio {};
```

```
28:         {
29:     public:
30:         xTamanio(int tamano) : suTamanio(tamano) {}
31:         ~xTamanio() {}
32:         virtual int ObtenerTamano()
33:         { return suTamanio; }
34:         virtual void ImprimirError()
35:         {
36:             cout << "Error en tamaño. Se recibieron: ";
37:             cout << suTamanio << endl;
38:         }
39:     protected:
40:         int suTamanio;
41:     };
42: class xMuyGrande : public xTamanio
43: {
44: public:
45:     xMuyGrande(int tamano) : xTamanio(tamano) {}
46:     virtual void ImprimirError()
47:     {
48:         cout << "¡Muy grande! Se recibieron: ";
49:         cout << xTamanio::suTamanio << endl;
50:     }
51: };
52: class xMuyChico : public xTamanio
53: {
54: public:
55:     xMuyChico(int tamano) : xTamanio(tamano) {}
56:     virtual void ImprimirError()
57:     {
58:         cout << "¡Muy chico! Se recibieron: ";
59:         cout << xTamanio::suTamanio << endl;
60:     }
61: };
62: class xCero : public xMuyChico
63: {
64: public:
65:     xCero(int tamano) : xMuyChico(tamano) {}
66:     virtual void ImprimirError()
67:     {
68:         cout << "¡Cero!. Se recibieron: ";
69:         cout << xTamanio::suTamanio << endl;
70:     }
71: };
72: class xNegativo : public xTamanio
73: {
74: public:
75:     xNegativo(int tamano) : xTamanio(tamano) {}
76:     virtual void ImprimirError()
77:     {
78:         cout << "¡Negativo! Se recibieron: ";
```

20

**LISTADO 20.5** CONTINUACIÓN

```
79:             cout << xTamanio::suTamanio << endl;
80:         }
81:     };
82: private:
83:     int * apTipo;
84:     int suTamanio;
85: };
86:
87: Arreglo::Arreglo(int tamano):
88: suTamanio(tamano)
89: {
90:     if (tamano == 0)
91:         throw xCero(tamano);
92:     if (tamano > 30000)
93:         throw xMuyGrande(tamano);
94:     if (tamano < 0)
95:         throw xNegativo(tamano);
96:     if (tamano < 10)
97:         throw xMuyChico(tamano);
98:     apTipo = new int[ tamano ];
99:     for (int i = 0; i < tamano; i++)
100:         apTipo[ i ] = 0;
101: }
102:
103: int & Arreglo::operator[] (int desplazamiento)
104: {
105:     int tamano = ObtenerSuTamanio();
106:
107:     if (desplazamiento >= 0 && desplazamiento < ObtenerSuTamanio())
108:         return apTipo[ desplazamiento ];
109:     throw xLímite();
110:     return apTipo[0];
111: }
112:
113: const int & Arreglo::operator[] (int desplazamiento) const
114: {
115:     int tamano = ObtenerSuTamanio();
116:
117:     if (desplazamiento >= 0 && desplazamiento < ObtenerSuTamanio())
118:         return apTipo[ desplazamiento ];
119:     throw xLímite();
120:     return apTipo[ 0 ];
121: }
122:
123: int main()
124: {
125:     try
126:     {
```

```
127:         Arreglo arregloInt(9);
128:
129:         for (int j = 0; j < 100; j++)
130:         {
131:             arregloInt[ j ] = j;
132:             cout << "arregloInt[" << j << "] está bien...\n";
133:         }
134:     }
135:     catch (Arreglo::xLimite)
136:     {
137:         cout << "¡No se pudo procesar su entrada!\n";
138:     }
139:     catch (Arreglo::xTamanio & laExcepcion)
140:     {
141:         laExcepcion.ImprimirError();
142:     }
143:     catch (...)
144:     {
145:         cout << "¡Algo salió mal!\n";
146:     }
147:     cout << "Listo.\n";
148:     return 0;
149: }
```

**SALIDA** ¡Muy chico! Se recibieron: 9  
Listo.

**ANÁLISIS** El listado 20.5 declara un método virtual en la clase `xTamanio` llamado `ImprimirError()`, el cual imprime un mensaje de error y el tamaño actual de la clase. Este método se redefine en cada una de las clases derivadas.

En la línea 139, el objeto de excepción se declara como una referencia. Cuando se llama a `ImprimirError()` con una referencia a un objeto, el polimorfismo ocasiona que se invoque la versión correcta de `ImprimirError()`. El código es más limpio, más fácil de entender y de mantener.

## Uso de excepciones y plantillas

Al crear excepciones para trabajar con plantillas, tiene una opción: puede crear una excepción para cada instancia de la plantilla, o puede utilizar clases de excepciones declaradas fuera de la declaración de la plantilla. El listado 20.6 muestra ambos métodos.

**ENTRADA****LISTADO 20.6 Uso de excepciones con plantillas**

```
1: // Listado 20.6: Excepciones con plantillas
2:
3: #include <iostream.h>
4:
5: const int TamanoPredet = 10;
6:
7:
8: class xLimite {};
9: ..
10: template < class T >
11: class Arreglo
12: {
13: public:
14:     // constructores
15:     Arreglo(int suTamano = TamanoPredet);
16:     Arreglo(const Arreglo & rhs);
17:     ~Arreglo()
18:     { delete [] apTipo; }
19:     // operadores
20:     Arreglo & operator=(const Arreglo< T > &);
21:     T & operator[](int desplazamiento);
22:     const T & operator[](int desplazamiento) const;
23:     // métodos de acceso
24:     int ObtenerSuTamano() const
25:     { return suTamano; }
26:     // función amiga
27:     friend ostream & operator<< <> (ostream &, const Arreglo< T > &);
28:     // definir las clases de excepciones
29:     class xTamano {};
30: private:
31:     int * apTipo;
32:     int suTamano;
33: };
34:
35: template < class T >
36: Arreglo< T >::Arreglo(int tamano):
37: suTamano(tamano)
38: {
39:     if (tamano < 10 || tamano > 30000)
40:         throw xTamano();
41:     apTipo = new T[ tamano ];
42:     for (int i = 0; i < tamano; i++)
43:         apTipo[ i ] = 0;
44: }
45:
46: template < class T >
47: Arreglo< T > & Arreglo< T >::operator=(const Arreglo< T > & rhs)
48: {
49:     if (this == &rhs)
```

```

50:         return *this;
51:     delete [] apTipo;
52:     suTamanio = rhs.ObtenersuTamanio();
53:     apTipo = new T[ suTamanio ];
54:     for (int i = 0; i < suTamanio; i++)
55:         apTipo[ i ] = rhs[ i ];
56: }
57:
58: template < class T >
59: Arreglo< T >::Arreglo(const Arreglo< T > & rhs)
60: {
61:     suTamanio = rhs.ObtenersuTamanio();
62:     apTipo = new T[ suTamanio ];
63:     for (int i = 0; i < suTamanio; i++)
64:         apTipo[ i ] = rhs[ i ];
65: }
66:
67: template < class T >
68: T & Arreglo< T >::operator[](int desplazamiento)
69: {
70:     int tamanio = ObtenersuTamanio();
71:
72:     if (desplazamiento >= 0 && desplazamiento < ObtenerTamano())
73:         return apTipo[ desplazamiento ];
74:     throw xLmite();
75:     return apTipo[ 0 ];
76: }
77:
78: template < class T >
79: const T & Arreglo< T >::operator[](int desplazamiento) const
80: {
81:     int mitamanio = ObtenersuTamanio();
82:
83:     if (desplazamiento >= 0 && desplazamiento < ObtenerTamano())
84:         return apTipo[ desplazamiento ];
85:     throw xLmite();
86: }
87:
88: template < class T >
89: ostream & operator<< (ostream & salida, const Arreglo<
90: =>T > & elArreglo)
91: {
92:     for (int i = 0; i < elArreglo.ObtenersuTamanio(); i++)
93:         salida << "[" << i << "] " << elArreglo[ i ] << endl;
94:     return salida;
95: }
96: int main()
97: {
98:     try
99:     {
100:         Arreglo< int > arregloInt(9);

```

20

*continúa*

**LISTADO 20.6** CONTINUACIÓN

```

101:
102:         for (int j = 0; j < 100; j++)
103:         {
104:             arregloInt[ j ] = j;
105:             cout << "arregloInt[" << j << "] está bien..." << endl;
106:         }
107:     }
108:     catch (xLimite)
109:     {
110:         cout << "¡No se pudo procesar su entrada!\n";
111:     }
112:     catch (Arreglo< int >::xTamanio)
113:     {
114:         cout << "¡Tamaño incorrecto!\n";
115:     }
116:     cout << "Listo.\n";
117:     return 0;
118: }
```

**SALIDA**

iTamaño incorrecto!  
Listo.

**ANÁLISIS**

La primera excepción, `xLimite`, se declara fuera de la definición de la plantilla, en la línea 8. La segunda excepción, `xTamanio`, se declara desde el interior de la definición de la plantilla, en la línea 29.

La excepción `xLimite` no está atada a la clase de plantilla, pero se puede utilizar de la misma forma que cualquier otra clase. `xTamanio` está atada a la plantilla y debe ser llamada con base en el `Arreglo` instanciado. Puede ver la diferencia en la sintaxis de las dos instrucciones `catch`. La línea 108 muestra `catch (xLimite)`, pero la línea 112 muestra `catch (Arreglo< int >::xTamanio)`. La segunda está atada a la instanciación de un `Arreglo` de enteros.

## Cómo activar excepciones sin errores

Cuando los programadores de C++ se reúnen después del trabajo para tomarse una cerveza virtual en el bar del ciberespacio, la charla, por lo general, gira en torno a si las excepciones se deben utilizar para condiciones de rutina. Algunos afirman que, debido a su naturaleza, las excepciones se deben reservar para aquellas circunstancias predecibles pero excepcionales (¡de ahí el nombre!) que un programador debe anticipar, pero que no son parte del procesamiento de rutina del código.

Otros recalcan que las excepciones ofrecen una forma poderosa y limpia de regresar a través de muchos niveles de llamadas a funciones sin peligro de fugas de memoria. Un ejemplo muy común es el siguiente: el usuario solicita una acción en un entorno GUI. La parte del código que atrapa la solicitud debe llamar a un método en un administrador

de diálogos, el cual, a su vez, llama al código que procesa la solicitud, el cual llama al código que decide cuál cuadro de diálogo utilizar, y éste llama al código que se encarga de colocar el cuadro de diálogo, el cual, finalmente, llama al código que procesa la entrada del usuario. Si el usuario oprime el botón Cancelar, el código debe regresar al primer método llamador en donde se manejó la solicitud original.

Una forma de solucionar este problema es colocar un bloque `try` alrededor de la llamada original y atrapar `DialogoCancelar` como una excepción, la cual puede ser producida por el manejador del botón Cancelar. Esto es seguro y efectivo, pero oprimir Cancelar es una circunstancia de rutina, no una excepcional.

Con frecuencia, esto se convierte en algo así como un argumento religioso, pero una forma razonable de decidir la cuestión es preguntar lo siguiente: ¿Usar de esta forma las excepciones hace que el código sea más fácil o más difícil de entender? ¿Hay menos riesgos de errores y fugas de memoria, o hay más? ¿Será más difícil o más fácil mantener este código? Estas decisiones, como muchas otras, requerirán de un análisis de las concepciones que haya que hacer; no existe una sola respuesta correcta que sea obvia.

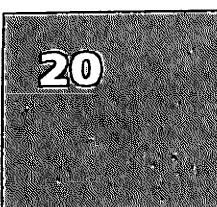
## Cómo tratar con los errores y la depuración

Casi todos los entornos de desarrollo modernos incluyen uno o más depuradores de alto poder. La idea esencial acerca del uso de un depurador es la siguiente: usted ejecuta su depurador, el cual carga el código fuente, y luego ejecuta su programa desde el depurador. Esto le permite ver cada instrucción de su programa a medida que se va ejecutando, y examinar sus variables a medida que van cambiando durante la vida de su programa.

Todos los compiladores le permitirán compilar con o sin símbolos. Compilar con símbolos le indica al compilador que cree el mapeo necesario entre su código fuente y el programa generado; el depurador utiliza esto para apuntar a la línea de código fuente que corresponde a la siguiente acción del programa.

Los depuradores simbólicos facilitan esta tarea. El depurador GNU, `gdb`, es un depurador simbólico que se ejecuta en modo de texto; algunos otros depuradores simbólicos soportan un modo de pantalla completa con ventanas. Al cargar el depurador, éste leerá todo el código fuente. Puede pasar por las llamadas a funciones sin entrar, o hacer que el depurador entre a la función, línea por línea.

Con la mayoría de los depuradores puede alternar entre el código fuente y la salida para ver los resultados de cada instrucción ejecutada. Algo que ofrece más poder es examinar el estado actual de cada variable, examinar estructuras de datos complejas, examinar el valor de los datos miembro dentro de las clases y ver los valores actuales en memoria de varios apuntadores y de otras ubicaciones de memoria. Puede ejecutar varios tipos de control dentro de un depurador que incluyan establecer puntos de interrupción, establecer puntos de observación, examinar la memoria y analizar el código en lenguaje ensamblador.



## Uso de gdb o depurador GNU

El primer paso para utilizar el depurador GNU (o casi cualquier otro depurador) es asegurarse de que el compilador haya incluido el código requerido en el archivo ejecutable. Con esta información, el depurador sabe dónde se guardan las variables, los nombres de éstas y de las funciones, y cómo relacionar el código fuente y el código binario.

La primera pregunta que podría venir a la mente sería: “¿Por qué no incluir siempre el código especial de depuración en el archivo ejecutable?”. La respuesta es simple; ese código reduce la velocidad de ejecución del programa y ocasiona que ocupe memoria adicional. Todos queremos que nuestros programas se ejecuten con rapidez y que utilicen la menor cantidad posible de recursos del sistema. Así que, a menos que se necesite, no le indicamos al compilador que incluya esta información.

Para indicarle al compilador GNU que incluya información de depuración en el código binario, se utiliza la opción `-g`:

```
g++ -g suprograma.cxx -o suprograma
```

De manera predeterminada, la información de depuración no se incluye.

Después de haber compilado todos sus módulos con la opción `-g` y de haberlos enlazado en un archivo ejecutable, está listo para utilizar gdb para depurar su programa.

La tabla 20.1 muestra algunos de los comandos comunes que se usan en gdb.

**TABLA 20.1** Comandos comunes de gdb

| Comando                              | Propósito                                                                                                                                                                      |
|--------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>break [archivo:]funcion</code> | Establecer un punto de interrupción al entrar a <i>funcion</i> en <i>archivo</i> . <i>archivo</i> es opcional.                                                                 |
| <code>bt</code>                      | Forma abreviada de “backtrace”; mostrar la pila del programa.                                                                                                                  |
| <code>c</code>                       | Continuar la ejecución del programa después del punto de interrupción.                                                                                                         |
| <code>disassemble</code>             | Mostrar el código binario como código en lenguaje ensamblador en vez de lenguaje fuente (es decir, C++).                                                                       |
| <code>display exp</code>             | Mostrar el valor de la variable <i>exp</i> cada vez que el programa se detenga.                                                                                                |
| <code>help</code>                    | Obtener ayuda en general o ayuda en categorías de comandos, comandos específicos y opciones para comandos específicos. <code>help</code> es casi el comando más útil de todos. |
| <code>list [[archivo:]]linea]</code> | Muestra ±5 líneas de código fuente. <i>archivo</i> y <i>linea</i> especifican el código fuente a desplegar y son opcionales.                                                   |
| <code>next</code>                    | Ejecutar el siguiente paso del programa sin entrar en las funciones llamadas.                                                                                                  |

| Comando                         | Propósito                                                                                                                                                                                                                                                                                                                                                     |
|---------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>print exp</code>          | Imprimir <code>exp</code> ; <code>exp</code> puede ser variable, nombre de función o expresión compleja, como el comienzo de un arreglo (nombre del arreglo) más un valor para mostrar un elemento especificado. Esto le permite examinar la memoria.                                                                                                         |
| <code>quit</code>               | Salir de gdb. Si tiene un programa en ejecución, se le pedirá que confirme.                                                                                                                                                                                                                                                                                   |
| <code>run arglista</code>       | Ejecutar el programa desde el principio con la lista opcional de argumentos de línea de comandos <code>arglista</code> .                                                                                                                                                                                                                                      |
| <code>set variable = exp</code> | Asignar a la <code>variable</code> del código fuente la expresión <code>exp</code> . Igual que con <code>print</code> , <code>exp</code> puede ser una variable, nombre de función o expresión compleja. La variable sigue la sintaxis del lenguaje fuente y reconoce cosas tales como notación de arreglos. Esto le permite alterar el estado de la memoria. |
| <code>set</code>                | Modificar las variables de entorno de gdb.                                                                                                                                                                                                                                                                                                                    |
| <code>step</code>               | Ejecutar el siguiente paso del programa (entrar en cualquier función llamada).                                                                                                                                                                                                                                                                                |
| <code>undisplay</code>          | Cancelar el despliegue.                                                                                                                                                                                                                                                                                                                                       |
| <code>watch</code>              | Crear un punto de observación.                                                                                                                                                                                                                                                                                                                                |
| <code>whatis exp</code>         | Desplegar el tipo de datos de <code>exp</code> .                                                                                                                                                                                                                                                                                                              |

El depurador GNU también acepta opciones de línea de comandos. Al ejecutar gdb, por lo general se escribe un comando como el siguiente:

`gdb suprograma`

También puede utilizar gdb para depurar un programa que falle al ejecutarlo (cuando se produzca un archivo core). En este caso, su comando se verá así:

`gdb suprograma core`

Debe tener en cuenta que el archivo core que utilizará para la depuración es el mismo que se creó debido a la “caída” de su programa. Al depurar un archivo core, puede examinar la memoria y la pila de llamadas, pero no puede ejecutar pasos porque el programa no está en ejecución.

20

### Tip

Dos tips rápidos para trabajar con gdb:

Debe establecer un punto de interrupción antes de emitir el comando `run`. De no ser así, el programa se ejecutará hasta completarse o fallar. Un buen hábito es establecer un punto de interrupción dentro de `main` para que se pueda realizar la inicialización de la biblioteca estándar en tiempo de ejecución, y luego introducir los otros puntos de interrupción, puntos de observación y despliegues.

El otro tip es acerca de un mensaje de error confuso:

`prueba1.cxx:6: No such file or directory (ENOENT).`

Si recibe este mensaje, se debe a que no se puede encontrar el archivo fuente (`prueba1.cxx`), o a que su archivo fuente tiene una extensión diferente (`prueba1.cpp`) pero el compilador confundió las cosas. La solución para la primer situación es ejecutar el programa en el directorio apropiado o utilizar la opción `-d` para especificar el directorio que contiene los archivos fuente. La segunda es un poco rara: copiar su archivo fuente (`prueba1.cpp`) en el nombre que gdb está buscando (`prueba1.cxx`), y el problema deberá desaparecer.

Para obtener mayor información acerca de las opciones de línea de comandos y los comandos internos de gdb, debe revisar el manual (`man gdb`) o el archivo de información (`info gdb`).

Las siguientes secciones describen el significado de los términos *punto de interrupción*, *punto de observación*, *examen* y *modificación del estado de la memoria* y *desensamblaje*.

## Uso de los puntos de interrupción

Los puntos de interrupción son instrucciones que indican al depurador que cuando una línea específica de código esté lista para ejecutarse, el programa debe detenerse. Esto le permite ejecutar su programa sin impedimentos hasta llegar a la línea en cuestión. Los puntos de interrupción le permiten analizar la condición actual de las variables justo antes y después de una línea crítica de código.

## Uso de los puntos de observación

Es posible pedir al depurador que le muestre el valor de una variable específica o que se detenga cuando se lea o se escriba a una variable específica. Los puntos de observación le permiten establecer estas condiciones, y algunas veces le permiten modificar el valor de una variable mientras el programa está en ejecución.

## Examen y modificación del estado de la memoria

A veces es importante ver los valores actuales que se guardan en memoria. Los depuradores modernos pueden mostrar valores en el formato de la variable actual; es decir, cadenas que se pueden mostrar como caracteres, enteros largos como números y no como 4 bytes, y así sucesivamente. El sofisticado depurador gdb puede incluso mostrar clases completas y proporcionar el valor actual de todas las variables miembro, incluyendo el apuntador `this`. Y lo que es igual de importante, gdb le permite cambiar los valores de las variables cuando el programa se está ejecutando.

## Desensamblaje

Aunque leer el código puede ser todo lo que se requiere para encontrar un bug (error), cuando todo lo demás falla, es posible que el depurador le muestre el código en lenguaje

ensamblador generado para cada línea del código fuente. Puede examinar el estado de los registros, de la memoria y de los indicadores, y por lo general ahondar en el funcionamiento interno de su programa tanto como lo requiera.

Aprenda a utilizar su depurador. Puede ser el arma más poderosa en su guerra santa contra los bugs. Los bugs en tiempo de ejecución son los más duros de encontrar y de eliminar, y un depurador poderoso puede hacer posible, si no es que fácil, encontrarlos casi todos.

## Resumen

Hoy aprendió cómo crear y utilizar las excepciones. Éstas son objetos que se crean cuando el código que se está ejecutando no puede manejar el error o cualquier otra condición excepcional que haya surgido. Otras partes del programa, que se encuentran más abajo en la pila de llamadas, implementan bloques `catch` que atrapan la excepción y realizan la acción apropiada.

Las excepciones son objetos normales creados por el usuario, y como tales se pueden pasar por valor o por referencia. Pueden contener datos y métodos, y el bloque `catch` puede utilizar esos datos para decidir cómo tratar la excepción.

Es posible crear múltiples bloques `catch`, pero después de que una excepción concuerda con la firma de un bloque `catch`, se considera que ya fue manejada y no se proporciona a los bloques `catch` subsecuentes. Es importante ordenar apropiadamente los bloques `catch`, de forma que los bloques `catch` específicos tengan la primera oportunidad, y que los bloques `catch` más generales se encarguen de los que no se manejan en forma específica.

Esta lección también examinó algunos de los fundamentos de los depuradores simbólicos, incluyendo el uso de puntos de observación, puntos de interrupción, etc. Estas herramientas pueden ayudarle a centrarse en la parte de su programa que está ocasionando el error y le permiten ver el valor de las variables a medida que cambian durante la ejecución del programa.

## Preguntas y respuestas

20

**P ¿Por qué preocuparse por producir excepciones? ¿Por qué no manejar el error donde ocurre?**

**R** A menudo, el mismo error se puede generar en distintas partes del código. Las excepciones le permiten centralizar el manejo de errores. Además, la parte del código que genera el error tal vez no sea el mejor lugar para determinar la forma de manejar el error.

**P ¿Por qué generar un objeto? ¿Por qué no sólo pasar un código de error?**

**R** Los objetos son más flexibles y poderosos que los códigos de error. Pueden transmitir más información, y los mecanismos constructores/destructores se pueden utilizar para la creación y remoción de recursos que se puedan necesitar para manejar apropiadamente la condición excepcional.

- P** ¿Por qué no utilizar excepciones para condiciones que no sean de error? ¿No sería conveniente poder regresar a las áreas de código de más atrás, incluso cuando existan condiciones que no sean excepcionales?
- R** Sí, algunos programadores de C++ utilizan excepciones para ese propósito. El peligro es que las excepciones podrían crear fugas de memoria al eliminar los elementos de la pila y algunos objetos se queden accidentalmente en el heap. Con las técnicas de programación cuidadosas y un buen compilador, esto se puede evitar. De otra manera, es cuestión de estética personal; algunos programadores sienten que, por sus excepciones naturales, no se deben utilizar para condiciones de rutina.
- P** ¿Se tiene que atrapar una excepción en el mismo lugar en el que el bloque try la creó?
- R** No, es posible atraparla en cualquier lugar de la pila de llamadas. Conforme se eliminan los elementos de la pila, la excepción se pasa a la parte baja de la pila hasta que es manejada.
- P** ¿Por qué utilizar un depurador si se puede utilizar cout con compilación condicional (#ifdef debug)?
- R** El depurador proporciona un mecanismo mucho más poderoso para avanzar paso a paso por su código y observar que los valores cambien sin tener que atestar el código con instrucciones de depuración.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Qué es una excepción?
2. ¿Qué es un bloque try?
3. ¿Qué es una instrucción catch?
4. ¿Qué información puede contener una excepción?
5. ¿Cuándo se crean los objetos de excepción?
6. ¿Se deben pasar las excepciones por valor o por referencia?
7. ¿Atrapará una instrucción catch una excepción derivada si está buscando la clase base?
8. Si se utilizan dos instrucciones catch, una para la clase base y una para la derivada, ¿cuál debe ir primero?

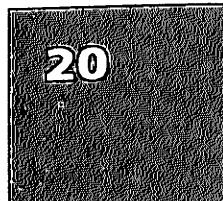
9. ¿Qué significa `catch(...)`?
10. ¿Qué es un punto de interrupción?

## Ejercicios

1. Cree un bloque `try`, una instrucción `catch` y una excepción simple.
2. Modifique la respuesta del ejercicio 1, coloque datos en la excepción junto con una función de acceso, y utilícela en el bloque `catch`.
3. Modifique la clase del ejercicio 2 para que sea una jerarquía de excepciones. Cambie el bloque `catch` para utilizar los objetos derivados y los objetos base.
4. Modifique el programa del ejercicio 3 para que tenga tres niveles de llamadas a funciones.
5. CAZA ERRORES: ¿Qué está mal en el siguiente código?

```
class xNoHayMemoria
{
public:
    xNoHayMemoria()
    {
        elMsje = new char[ 20 ];
        strcpy(elMsje, "error en memoria");
    }
    ~xNoHayMemoria()
    {
        delete [] elMsje;
        cout << "Memoria restablecida." << endl;
    }
    char * Mensaje()
    {
        return elMsje;
    }
private:
    char * elMsje;
};

main()
{
    try
    {
        char * var = new char;
        if (var == 0)
        {
            xNoHayMemoria * apx =           throw apx;
        }
    }
    catch(xNoHayMemoria * laExpcion)
    {
        cout << laExpcion->Mensaje() << endl;
        delete laExpcion;
    }
    return 0;
}
```



Este ejercicio muestra el bug planeado: Está asignando memoria para presentar el mensaje de error, pero lo hace precisamente cuando no hay memoria disponible para asignar (después de todo, ¡es lo que esta excepción maneja!). Puede probar este programa cambiando la línea `if (var == 0)` a `if (1)`, la cual obligará a que se produzca la excepción.

# SEMANA 3

DÍA 21

## Qué sigue

¡Felicitaciones! Ya casi termina una introducción intensiva de tres semanas a C++. Para estos momentos debe tener una comprensión sólida de C++, pero en la programación moderna siempre hay algo más que aprender. Este día cubre algunos detalles faltantes y luego lo prepara para continuar el estudio.

La mayor parte de lo que escribe en sus archivos de código fuente es C++. Su compilador interpreta esto y lo convierte en un programa. Sin embargo, antes de que el compilador se ejecute, se ejecuta el preprocesador, y esto proporciona una oportunidad para la compilación condicional. Hoy aprenderá lo siguiente:

- Qué es la compilación condicional y cómo manejarla
- Cómo escribir macros mediante el preprocesador
- Cómo utilizar el preprocesador para encontrar bugs
- Cómo manipular bits individuales y utilizarlos como indicadores
- Cuáles son los siguientes pasos para aprender a utilizar C++ en forma efectiva

## El preprocessador y el compilador

Cada vez que se ejecuta su compilador, primero se tiene que ejecutar su preprocessador. El preprocessador busca instrucciones de preprocessador, las cuales empiezan con un signo de numeral (#). El efecto de cada una de estas instrucciones es un cambio en el texto del código fuente. El resultado es un nuevo archivo de código fuente (un archivo temporal que por lo general no se ve, pero puede indicarle al compilador que lo guarde para que pueda examinarlo, si lo desea).

El compilador no lee su código fuente original; lee la salida del preprocessador y compila ese archivo. Ya ha visto el efecto de esto con la directiva `#include`. Esta directiva le indica al preprocessador que encuentre el archivo cuyo nombre sigue después de la directiva `#include` y que lo escriba en un archivo intermedio en esa ubicación. Es como si usted hubiera escrito todo ese archivo dentro de su código fuente, y cuando el compilador ve el código fuente, el archivo de encabezado está ahí.

### Cómo ver el formato del archivo intermedio

Casi todos los compiladores tienen un parámetro que se puede establecer ya sea en el entorno de desarrollo integrado (IDE) o en la línea de comandos, el cual le indica al compilador que guarde el archivo intermedio.

Para el compilador GNU, esa opción es `-E`. Para otros compiladores, revise la documentación para establecer los parámetros adecuados si quiere examinar este archivo.

## La directiva de preprocessador `#define`

El comando `#define` define una sustitución de cadenas. Si escribe

```
#define GRANDE 512
```

esto le indica al precompilador que sustituya la cadena 512 en cualquier lugar donde vea la cadena GRANDE. Ésta no es una cadena común de C++. Los caracteres 512 se sustituyen en el código fuente en cualquier lugar donde se vea el token GRANDE. Un token es una cadena de caracteres que se pueden utilizar en cualquier lugar donde se pueda utilizar una cadena o constante u otro conjunto de letras. Por lo tanto, si escribe

```
#define GRANDE 512
int miArreglo[GRANDE];
```

El archivo intermedio producido por el precompilador se verá así:

```
int miArreglo[512];
```

Observe que la directiva `#define` ha desaparecido. Las instrucciones del precompilador se quitan del archivo intermedio; no aparecen en el código fuente final.

## Uso de #define como alternativa para constantes

Una manera de utilizar `#define` es como sustituto para constantes. Sin embargo, ésta casi nunca es una buena idea, ya que `#define` simplemente hace una sustitución de cadenas y no comprueba los tipos. Como se explicó en la sección que trata sobre las constantes, existen muchas ventajas al usar la palabra reservada `const` en lugar de `#define`.

## Uso de #define para probar el código

Una segunda manera de utilizar `#define` es para declarar que se define una cadena de caracteres específica. Por lo tanto, podría escribir

```
#define GRANDE
```

Después, puede probar si `GRANDE` ha sido definida y realizar las acciones apropiadas. Los comandos del precompilador para probar si una cadena ha sido definida son `#ifdef` (si está definida) e `#ifndef` (si no está definida). El comando `#endif` debe estar después de ambos comandos antes de que el bloque termine (antes de la siguiente llave de cierre).

`#ifdef` se evalúa como `true` si la cadena que prueba ya ha sido definida. Por lo tanto, puede escribir

```
#ifdef DEPURACION  
cout << "Depuración está definida";  
#endif
```

Cuando el precompilador lee la directiva `#ifdef`, revisa una tabla que ha creado para ver si ya definió `DEPURACION`. Si ya la definió, `#ifdef` se evalúa como `true`, y todo lo que se encuentre antes de la siguiente directiva `#else` o `#endif` se escribe en el archivo intermedio para compilarlo. Si se evalúa como `false`, nada de lo que haya entre `#ifdef DEPURACION` y `#endif` se escribirá en el archivo intermedio; será como si nunca hubiera estado en el código fuente.

Observe que `#ifndef` es el inverso lógico de `#ifdef`. `#ifndef` se evalúa como `true` si la cadena no ha sido definida hasta ese punto del archivo.

## Uso de la directiva de precompilador #else

Como podría imaginar, el término `#else` se puede insertar entre `#ifdef` o entre `#ifndef` y el `#endif` de cierre. El listado 21.1 muestra cómo se utilizan estos términos.

**ENTRADA****LISTADO 21.1 Uso de #define**

```
1: // Listado 21.1 - Uso de #define  
2:  
3:     #define VersionDemo  
4:     #define VERSION_NT 5
```

*continúa*

**LISTADO 21.1** CONTINUACIÓN

```

5:      #include <iostream.h>
6:
7:      int main()
8:      {
9:          cout << "Comprobando las definiciones de VersionDemo, ";
10:         cout << "VERSION_NT y VERSION_WINDOWS...\n";
11:
12:         #ifdef VersionDemo
13:             cout << "VersionDemo definida.\n";
14:         #else
15:             cout << "VersionDemo no definida.\n";
16:         #endif
17:
18:         #ifndef VERSION_NT
19:             cout << "iVERSION_NT no definida!\n";
20:         #else
21:             cout << "VERSION_NT definida como: " << VERSION_NT << endl;
22:         #endif
23:
24:         #ifdef VERSION_WINDOWS
25:             cout << "iVERSION_WINDOWS definida!\n";
26:         #else
27:             cout << "VERSION_WINDOWS no fue definida.\n";
28:         #endif
29:
30:         cout << "Listo.\n";
31:         return 0;
32:     }

```

**SALIDA**

Comprobando las definiciones de VersionDemo,  
 ↵VERSION\_NT y VERSION\_WINDOWS...  
 VersionDemo definida.  
 VERSION\_NT definida como: 5  
 VERSION\_WINDOWS no fue definida.  
 Listo.

**ANÁLISIS**

En las líneas 3 y 4 se definen VersionDemo y VERSION\_NT, y esta última se define con la cadena 5. En la línea 12 se prueba la definición de VersionDemo, y como VersionDemo sí está definida (aunque sin valor), la prueba es true y se imprime la cadena de la línea 13.

En la línea 18 se prueba si VERSION\_NT no está definida. Como sí está definida, esta prueba falla y la ejecución del programa salta hacia la línea 21. Aquí, la cadena 5 sustituye a la cadena VERSION\_NT; el compilador ve esto como

```
cout << " VERSION_NT definida como: " << 5 << endl;
```

Observe que la primera cadena `VERSION_NT` no se sustituye ya que se encuentra en una cadena encerrada entre comillas. Sin embargo, la segunda cadena `VERSION_NT` sí se sustituye, y por consecuencia el compilador ve un 5 como si usted lo hubiera escrito ahí.

Por último, en la línea 24 el programa prueba la definición de `VERSION_WINDOWS`. Como esta cadena no se definió, la prueba falla y se imprime el mensaje de la línea 27.

## Inclusión y guardias de inclusión

Usted creará proyectos con muchos archivos diferentes. Probablemente organizará sus directorios de forma que cada clase tenga su propio archivo de encabezado (por ejemplo, `.hpp`) con la declaración de la clase y su propio archivo de implementación (por ejemplo, `.cxx`) con el código fuente para los métodos de las clases.

Su función `main()` estará en su propio archivo `.cxx`, y todos los archivos `.cxx` se compilarán en archivos `.o` (`.obj` en Windows y DOS), los cuales serán luego enlazados en un solo programa por el enlazador.

Ya que sus programas utilizarán métodos de muchas clases, se incluirán muchos archivos de encabezado en cada archivo. Además, los archivos de encabezado a menudo necesitan incluir otros archivos de encabezados. Por ejemplo, el archivo de encabezado para la declaración de una clase derivada debe incluir el archivo de encabezado para su clase base.

Imagine que la clase `Animal` está declarada en el archivo `ANIMAL.hpp`. La clase `Perro` (que se deriva de `Animal`) debe incluir el archivo `ANIMAL.hpp` en `PERRO.hpp`, o de lo contrario `Perro` no se podrá derivar de `Animal`. El archivo de encabezado de `Gato` también incluye a `ANIMAL.hpp` por la misma razón.

Si crea un método que utilice tanto un `Gato` como un `Perro`, correrá el peligro de incluir dos veces a `ANIMAL.hpp`. Esto generará un error en tiempo de compilación, pues no es válido declarar una clase (`Animal`) dos veces, aunque las declaraciones sean idénticas. Puede resolver este problema con los guardias de inclusión. Al principio de su archivo de encabezado de `ANIMAL`, escriba estas líneas:

```
#ifndef ANIMAL_HPP
#define ANIMAL_HPP
...
// todo el archivo va aquí
#endif
```

Esto dice: si no ha definido el término `ANIMAL_HPP`, défínalohora. Todo el contenido del archivo va entre la directiva `#define` y la directiva `#endif` de cierre.

La primera vez que su programa incluya este archivo, leerá la primera línea y la prueba se evaluará como `true`; es decir, aún no ha definido a `ANIMAL_HPP`. Por lo tanto, el programa define este término e incluye el archivo completo.

La segunda vez que su programa incluya el archivo ANIMAL.hpp, leerá la primera línea y la prueba se evaluará como `false`; ya se ha definido a `ANIMAL.hpp`. Por lo tanto, la ejecución del programa salta hasta la siguiente directiva `#else` (en este caso no hay ninguna) o hasta la siguiente `#endif` (al final del archivo). Por consecuencia, se salta todo el contenido del archivo, y así la clase no se declara dos veces.

El nombre actual del símbolo definido (`ANIMAL_HPP`) no es importante, aunque se acostumbra utilizar el nombre de archivo en mayúsculas, y cambiar el punto (.) por un guion bajo. Sin embargo, esto es sólo una convención.

### Nota

Nunca está de más utilizar guardias de inclusión. Con frecuencia le ahorrarán horas de tiempo de depuración.

## Funciones de macros

La directiva `#define` también se puede utilizar para crear funciones de macros. Una función de macro es un símbolo creado por medio de `#define`; toma un argumento en forma muy parecida a una función común. El preprocesador sustituirá la cadena de sustitución por cualquier argumento que se le dé. Por ejemplo, puede definir la macro DOBLE de la siguiente manera:

```
#define DOBLE(x) ((x) * 2)
```

y luego escribir en su código

```
DOBLE(4)
```

Se quitará toda la cadena `DOBLE(4)`, ¡y se sustituirá el valor 8! Cuando el precompilador vea el 4, lo sustituirá por `((4) * 2)`, lo que se evaluará como `4 * 2`, o sea 8.

Una macro puede tener más de un parámetro, y cada parámetro se puede utilizar en forma repetida en el texto de reemplazo. Dos macros comunes son `MAX` y `MIN`:

```
#define MAX(x,y) ((x) > (y) ? (x) : (y))  
#define MIN(x,y) ((x) < (y) ? (x) : (y))
```

Observe que en la definición de una función de macro, los paréntesis de apertura para la lista de parámetros deben estar inmediatamente después del nombre de la macro, sin espacios. El preprocesador no es tan considerado con los espacios en blanco como el compilador.

Si escribiera

```
#define MAX (x,y) ((x) > (y) ? (x) : (y))
```

y luego tratará de utilizar `MAX` de la siguiente manera:

```
int x = 5, y = 7, z;
z = MAX(x,y);
```

el código intermedio sería

```
int x = 5, y = 7, z;
z = (x,y) ((x) > (y) ? (x) : (y))(x,y)
```

Se realizaría una simple sustitución de texto, en lugar de invocar a la macro tipo función. Por lo tanto, el token MAX sería sustituido por (x,y) ((x) > (y) ? (x) : (y)), y después de esto seguiría (x,y), que estaba después de MAX.

No obstante, si quita el espacio entre MAX y (x,y), el código intermedio sería

```
int x = 5, y = 7, z;
z =7;
```

## ¿Para qué son todos esos paréntesis?

Tal vez se esté preguntando por qué hay tantos paréntesis en muchas de las macros que se han presentado hasta ahora. El preprocesador no requiere que se coloquen paréntesis alrededor de los argumentos de la cadena de sustitución, pero los paréntesis le ayudan a evitar efectos secundarios no deseados al pasar valores complicados a una macro. Por ejemplo, si define MAX como

```
#define MAX(x,y) x > y ? x : y
```

y pasa los valores 5 y 7, la macro funciona como se espera. Pero si pasa una expresión más complicada, obtendrá resultados inesperados, como se muestra en el listado 21.2.

### ENTRADA

### LISTADO 21.2 Uso de paréntesis en macros

```
1: // Listado 21.2 Expansión de macros
2:
3: #include <iostream.h>
4: #define CUBO(a)    ((a) * (a) * (a))
5: #define TRIPLE(a)  a * a * a
6:
7:
8: int main()
9: {
10:     long x = 5;
11:     long y = CUBO(x);
12:     long z = TRIPLE(x);
13:
14:     cout << "y: " << y << endl;
15:     cout << "z: " << z << endl;
16:
17:     long a = 5, b = 7;
```



**LISTADO 21.2** CONTINUACIÓN

---

```

18:         y = CUBO(a + b);
19:         z = TRIPLE(a + b);
20:
21:         cout << "y: " << y << endl;
22:         cout << "z: " << z << endl;
23:         return 0;
24:     }

```

---

**SALIDA**

```

y: 125
z: 125
y: 1728
z: 82

```

**ANÁLISIS** En la línea 4 se define la macro CUBO, y se coloca el argumento x entre paréntesis cada vez que se utiliza. En la línea 5 se define la macro TRIPLE, sin los paréntesis.

En el primer uso de estas macros, se da el valor 5 como parámetro, y ambas macros funcionan bien. CUBO(5) se expande a  $((5) * (5) * (5))$ , lo que se evalúa como 125, y TRIPLE(5) se expande a  $5 * 5 * 5$ , lo cual también se evalúa como 125.

En el segundo uso, en las líneas 17 a 19, el parámetro es  $5 + 7$ . En este caso, CUBO( $5+7$ ) se evalúa como

$$((5+7) * (5+7) * (5+7))$$

lo que se evalúa como

$$((12) * (12) * (12))$$

lo que, a su vez, se evalúa como 1728. Sin embargo, TRIPLE( $5+7$ ) se evalúa como

$$5 + 7 * 5 + 7 * 5 + 7$$

Ya que la multiplicación tiene mayor precedencia que la adición, esto se convierte en

$$5 + (7 * 5) + (7 * 5) + 7$$

lo que se evalúa como

$$5 + (35) + (35) + 7$$

lo cual finalmente se evalúa como 82.

## Macros en comparación con funciones y plantillas

En C++, las macros sufren de cuatro problemas. El primero es que pueden ser confusas si se hacen grandes, ya que todas las macros se deben definir en una línea. En la mayoría de los compiladores se puede extender esa línea por medio del carácter de barra diagonal inversa (\), lo que se conoce como *empalme de líneas*.

El preprocesador distribuido con los compiladores GNU no maneja de manera apropiada el empalme de líneas, por lo que todas las macros deben permanecer en una línea. Las macros grandes se vuelven rápidamente difíciles de manejar.

El segundo problema es que las macros se expanden en línea cada vez que se utilizan. Esto significa que si una macro se utiliza una docena de veces, la sustitución aparecerá 12 veces en el programa, en lugar de aparecer una vez, como ocurre con la llamada a una función. Por otro lado, la mayoría de las veces son más rápidas que la llamada a una función ya que se evita la sobrecarga de una llamada a función.

El hecho de que se expanden en línea conduce al tercer problema: la macro no aparece en el código fuente intermedio utilizado por los compiladores; por lo tanto, no está disponible en la mayoría de los depuradores. Esto dificulta la depuración de las macros.

Sin embargo, el último problema es el más grande: las macros no tienen seguridad de tipos. Aunque es conveniente que cualquier argumento se pueda utilizar con una macro, estomina completamente la fuerte tipificación de C++, por lo que es una maldición para los programadores de C++. Desde luego, la manera correcta de resolver esto es con las plantillas, como vio en el día 19, "Plantillas".

## Funciones en línea

A menudo es posible declarar una función en línea en lugar de una macro. Por ejemplo, el listado 21.3 crea una función CUBO, la cual logra lo mismo que la macro CUBO del listado 21.2, pero lo hace ofreciendo seguridad de tipos.

**ENTRADA****LISTADO 21.3** Uso de una función en línea en lugar de una macro

```
1: // Listado 21.3: Uso de una función en linea en lugar de una macro
2:
3: #include <iostream.h>
4:
5: inline unsigned long Cuadrado(unsigned long a)
6: {
7:     return a * a;
8: }
9:
10: inline unsigned long Cubo(unsigned long a)
11: {
12:     return a * a * a;
13: }
14:
15: int main()
16: {
17:     unsigned long x = 1;
18:     for (;;)
19:     {
20:         cout << "Escriba un número (0 para salir): ";
21:         cin >> x;
22:         if (x == 0)
```

**LISTADO 21.3** CONTINUACIÓN

```

20:           break;
21:           cout << "Usted escribió: " << x;
22:           cout << ". Cuadrado(" << x << "): ";
23:           cout << Cuadrado(x);
24:           cout << ". Cubo(" << x << "): ";
25:           cout << Cubo(x) << "." << endl;
26:       }
27:   return 0;
28: }
```

**SALIDA**

Escriba un número (0 para salir): 1  
 Usted escribió: 1. Cuadrado(1): 1. Cubo(1): 1.  
 Escriba un número (0 para salir): 2  
 Usted escribió: 2. Cuadrado(2): 4. Cubo(2): 8.  
 Escriba un número (0 para salir): 3  
 Usted escribió: 3. Cuadrado(3): 9. Cubo(3): 27.  
 Escriba un número (0 para salir): 4  
 Usted escribió: 4. Cuadrado(4): 16. Cubo(4): 64.  
 Escriba un número (0 para salir): 5  
 Usted escribió: 5. Cuadrado(5): 25. Cubo(5): 125.  
 Escriba un número (0 para salir): 6  
 Usted escribió: 6. Cuadrado(6): 36. Cubo(6): 216.  
 Escriba un número (0 para salir): 0

**ANÁLISIS**

En las líneas 5 y 7 se definen dos funciones en línea: Cuadrado() y Cubo(). Cada una se declara en línea, por lo que al igual que una función de macro, éstas se expandirán en el lugar adecuado para cada llamada, y no ocurrirá una sobrecarga en las llamadas a funciones.

Como recordatorio, una función expandida en línea significa que el contenido de la función se colocará en el código en cualquier lugar en el que se haga la llamada a la función (por ejemplo, en la línea 23). Como la llamada a la función nunca se hace, no hay sobrecarga por colocar en la pila la dirección de retorno y los parámetros.

En la línea 23 se llama a la función Cuadrado, y en la línea 25 se llama a la función Cubo. De nuevo, como éstas son funciones en línea, es exactamente como si esta línea se hubiera escrito así:

```

22:           cout << ". Cuadrado(" << x << "): ";
23:           cout << x * x;
24:           cout << ". Cubo(" << x << "): ";
25:           cout << x*x*x << endl;
```

## Manipulación de cadenas

El preprocesador proporciona dos operadores especiales para manipular cadenas en las macros. El operador de cadena (#) sustituye cualquier cosa que le siga por una cadena entre comillas. El operador de concatenación une dos cadenas en una.

## Uso de cadenas con la directiva #define

El operador de cadena coloca comillas alrededor de cualquier carácter que le siga, hasta el siguiente espacio en blanco. Por ejemplo, si escribe

```
#define ESCRIBECADENA(x) cout << #x
```

y luego hace la siguiente llamada:

```
ESCRIBECADENA(Esta es una cadena);
```

El precompilador la convertirá en

```
cout << "Ésta es una cadena";
```

Observe que la cadena Ésta es una cadena se coloca entre comillas, como lo requiere cout.

## Concatenación

El operador de concatenación le permite unir más de un término para formar una nueva palabra. La nueva palabra es en realidad un token que se puede usar como nombre de una clase, nombre de una variable, desplazamiento en un arreglo, o en cualquier lugar en el que pueda aparecer una serie de letras.

Suponga por un momento que tiene cinco funciones llamadas fUnoImprimir, fDosImprimir, fTresImprimir, fCuatroImprimir y fCincoImprimir. Entonces puede declarar

```
#define FIMPRIMIR(x) f ## x ## Imprimir
```

y luego utilizarla con FIMPRIMIR(Dos) para generar fDosImprimir y con FIMPRIMIR(Tres) para generar fTresImprimir.

Al finalizar la semana 2 se desarrolló una clase llamada ListaPiezas. Esta lista sólo podía manejar objetos de tipo Lista. Suponga que esta lista funciona bien, y que le gustaría crear listas de animales, autos, computadoras, etc.

Un método sería crear ListaAnimales, ListaAutos, ListaComputadoras, y así sucesivamente, cortando y pegando el código en los lugares adecuados. Esto se convertiría rápidamente en una pesadilla, pues cualquier cambio en una lista se debe escribir en todas las demás.

Una alternativa es utilizar macros y el operador de concatenación. Por ejemplo, en la mayoría de los compiladores, podría definir

```
#define Listade(Tipo) class Tipo##Lista \
{ \
public: \
Tipo##Lista(){} \
private: \
int suLongitud; \
};
```

Para GNU, se definiría de la siguiente manera:

```
#define Listade(Tipo) class Tipo##Lista { public: Tipo##Lista(){}  
private: int suLongitud; };
```

Claro que la página no es lo suficientemente amplia para mostrar toda la macro como una línea. La mayoría de las ventanas de los editores tampoco son lo suficientemente amplias, pero por lo menos se da una idea.

Este ejemplo es demasiado escaso, pero la idea sería colocar todos los métodos y datos necesarios. Cuando estuviera listo para crear una `ListaAnimales`, escribiría

`Listade(Animal)`

y esto se convertiría en la declaración de la clase `ListaAnimales`. Hay algunos problemas con este método, los cuales se tratan en el día 19, "Plantillas".

## Macros predefinidas

Muchos compiladores predefinen una variedad de macros útiles, incluyendo `_DATE_`, `_TIME_`, `_LINE_` y `_FILE_`. Cada uno de estos nombres está rodeado por dos caracteres de guión bajo para reducir la probabilidad de que los nombres tengan conflicto con los que usted utilice en su programa.

Cuando el precompilador ve una de estas macros, hace las sustituciones apropiadas. Para `_DATE_`, se sustituye la fecha actual. Para `_TIME_`, se sustituye la hora actual. `_LINE_` y `_FILE_` se reemplazan con el número de línea y el nombre de archivo del código fuente, respectivamente. Debe tener en cuenta que esta sustitución se hace cuando se precompila el código fuente, no cuando se ejecuta el programa. Si pide al programa que imprima `_DATE_`, no obtendrá la fecha actual, sino la fecha en la que se compiló el programa. Estas macros definidas son muy útiles en la depuración.

## ASSERT()

GNU y muchos otros compiladores ofrecen una macro llamada `ASSERT()`. Esta macro regresa `true` si su parámetro se evalúa como `true` y realiza algún tipo de acción si se evalúa como `false`. GNU y muchos otros compiladores abortarán el programa en caso de que una macro `ASSERT()` falle; otros producirán una excepción (vea el día 20, "Excepciones y manejo de errores").

Una característica poderosa de la macro `ASSERT()` es que el preprocesador la comprime sin código alguno si `DEPURAR` no está definida. Es una gran ayuda durante el desarrollo, y cuando el producto final está terminado, no se castiga el rendimiento ni se incrementa el tamaño de la versión ejecutable del programa.

En vez de depender de la macro `ASSERT()` proporcionada por el compilador, usted puede escribir su propia macro `ASSERT()`. El listado 21.4 proporciona una macro `ASSERT()` sencilla y muestra su uso.

**ENTRADA** **LISTADO 21.4** Una macro `ASSERT()` sencilla

```
1: // Listado 21.4 ASSERT
2:
3: #define DEPURAR
4: #include <iostream.h>
5:
6: #ifndef DEPURAR
7: #define ASSERT(x)
8: #else
9: #define ASSERT(x) if (! (x)) { cout << "!!ERROR!! Assert ";
  => cout << "#x << \" falló\n\"; cout << \" en la linea \" << __LINE__ << "\n";
  => cout << \" del archivo \" << __FILE__ << "\n"; }
10: //
11: //
12: //
13: //
14: //
15: //
16: #endif
17:
18:
19: int main()
20: {
21:     int x = 5;
22:
23:     cout << "Primera Assert: \n";
24:     ASSERT(x == 5);
25:     cout << "\nSegunda Assert: \n";
26:     ASSERT(x != 5);
27:     cout << "\nListo.\n";
28:
29: }
```

**SALIDA** Primera Assert:

Segunda Assert:

```
ERROR!! Assert x !=5 falló
en la linea 24
del archivo lst21-04.cxx
Listo.
```

**ANÁLISIS** En la línea 3 se define el término `DEPURAR`. Por lo general, esto se haría desde la línea de comandos (por medio del argumento de línea de comandos `-D` para `g++`) en tiempo de compilación, para que pueda activar y desactivar esto según lo requiera. En la línea 9 se define la macro `ASSERT()`. Por lo general, esto se haría en un archivo de encabezado, y ese archivo de encabezado (`ASSERT.hpp`) se incluiría en todos sus archivos de implementación.

En la línea 6 se prueba el término DEPURAR. Si no está definido, ASSERT() se define para no crear ningún código. Si DEPURAR está definido, se aplica la funcionalidad definida en la línea 9.

La instrucción ASSERT() en sí es larga debido a que el preprocesador GNU no soporta la división de líneas.

Muchos otros preprocesadores sí soportan la división de líneas, en donde ASSERT() se divide entre siete líneas de código fuente en lo que al precompilador concierne. Un ejemplo de este tipo de código sería reemplazar las líneas 9 a 15 del listado 21.4

```
9: #define ASSERT(x) \
10: if (! (x)) \
11: { \
12:     cout << "!!ERROR!! Assert " << #x << " falló\n"; \
13:     cout << " en la línea " << __LINE__ << "\n"; \
14:     cout << " del archivo " << __FILE__ << "\n"; \
15: }
```

En la línea 10 se prueba el valor que se pasa como parámetro; si se evalúa como false, se invocan las instrucciones de las líneas 12 a 14, y se imprime un mensaje de error. Si el valor que se pasa se evalúa como true, no se realiza ninguna acción.

## Depuración con ASSERT()

Al escribir su programa, a menudo sabrá muy dentro de su ser que algo es verdadero: una función tiene cierto valor, un apuntador es válido, y así sucesivamente. La naturaleza de los bugs (errores) es que, bajo ciertas condiciones, lo que usted sabe que es cierto, podría no serlo. Por ejemplo, sabe que un apuntador es válido, a pesar de que el programa falle. ASSERT() puede ayudarlo a encontrar este tipo de bugs, pero sólo si lo utiliza con frecuencia y libremente en su código. Cada vez que asigne o que pase un apuntador como parámetro o como valor de retorno de una función, asegúrese de afirmar (ASSERT) que el apuntador es válido. Cada vez que su código dependa de que se encuentre un valor específico en una variable, afirme (ASSERT()) que eso es cierto.

No hay ningún castigo por el uso frecuente de ASSERT(); se quita del código cuando se quita la definición de la depuración. También proporciona una buena documentación interna, recordando al lector lo que usted cree que es cierto en cualquier momento dado en el flujo del código.

## ASSERT() en comparación con las excepciones

Ayer vio cómo trabajar con las excepciones para manejar condiciones de error. Es importante observar que ASSERT() no tiene el propósito de manejar condiciones de error en tiempo de ejecución, como datos incorrectos, condiciones de agotamiento de memoria, incapacidad para abrir un archivo, etc. ASSERT() fue creada sólo para atrapar errores de programación. Es decir, si “se dispara” una macro ASSERT(), usted sabrá que tiene un bug en su código.

Esto es crítico pues cuando envíe su código a sus clientes, se quitarán instancias de ASSERT(). No puede depender de una macro ASSERT() para manejar un problema en tiempo de ejecución ya que ASSERT() no estará ahí.

Es un error común utilizar a ASSERT() para probar el valor de retorno de una asignación de memoria:

```
Animal *apGato = new Gato;  
ASSERT(apGato); // mal uso de Assert  
apGato->UnaFuncion();
```

Éste es un clásico error de programación; cada vez que el programador ejecute el programa, habrá suficiente memoria disponible y ASSERT() nunca se disparará. Después de todo, el programador está ejecutando el programa con bastante RAM adicional para acelerar la velocidad del compilador, del depurador, etc. Luego, el programador envía el ejecutable, y el pobre usuario, que tiene menos memoria, llega a esta parte del programa y la llamada a new falla, y regresa NULL. Sin embargo, la macro ASSERT() ya no se encuentra en el código y nada indica que el apuntador está apuntando a NULL. Tan pronto como llegue a la instrucción apGato->UnaFuncion(), el programa fallará.

Recibir NULL de una asignación de memoria no es un error de programación, aunque sí es una situación excepcional. Su programa debe ser capaz de recuperarse de esta condición, aunque sea sólo para producir una excepción. Recuerde: toda la instrucción ASSERT() desaparece cuando no se define DEPURAR. Las excepciones se trataron con detalle en el día 20.

## Efectos secundarios

Es común encontrar que un bug aparece sólo después de que se quitan las instancias de ASSERT(). Esto casi siempre se debe a que el programa depende sin querer de los efectos secundarios de las cosas que se realizan en el código contenido en las instancias de ASSERT() y demás código de sólo depuración. Por ejemplo, si escribe

```
ASSERT (x = 5)
```

cuando lo que quiere es probar si  $x == 5$ , creará un bug especialmente terrible.

Suponga que justo antes de esta macro ASSERT() llamó a una función que establecía el valor de  $x$  en 0. Con esta macro ASSERT(), usted piensa que está probando si  $x$  es igual a 5; de hecho, está asignando el valor 5 a  $x$ . La prueba regresa true debido a que  $x = 5$  no sólo asigna el valor 5 a  $x$ , sino que también regresa el valor 5, y como 5 no es igual a cero, se evalúa como true.

Al pasar la instrucción ASSERT(),  $x$  realmente es igual a 5 (¡le acaba de asignar ese valor!). Su programa funciona a la perfección. Está listo para enviarlo al cliente, así que desactiva la depuración. Ahora desaparece la instrucción ASSERT(), y a  $x$  ya no se le asigna el valor 5. Como  $x$  valía 0 antes de esto, sigue siendo 0 y su programa falla.

Motivado por la frustración, vuelve a activar la depuración, y ¡listo! El bug desaparece. De nuevo, esto parece gracioso, pero no lo es, así que debe ser muy cuidadoso con los efectos secundarios producidos en el código de depuración. Si ve un bug que aparece sólo cuando la depuración está desactivada, analice cuidadosamente su código de depuración para buscar efectos secundarios desagradables.

## Constantes de clases

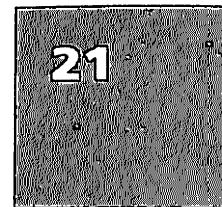
La mayoría de las clases tiene algunas condiciones que siempre deben ser verdaderas al dejar de utilizar una función miembro de la clase. Estas constantes de clase son el *sine qua non* (es decir, la condición indispensable) de su clase. Por ejemplo, puede ser cierto que su objeto CIRCULO nunca debe tener un radio de cero, o que su objeto ANIMAL siempre debe tener una edad mayor de cero y menor de 100.

Puede ser muy útil declarar un método Constantes() que regrese true sólo si cada una de estas condiciones sigue siendo true. Luego puede utilizar ASSERT(Constantes()) al inicio y al término de cada método de la clase. La excepción sería que sus Constantes() no esperen regresar true antes de que su constructor se ejecute o después de que su destructor termine. El listado 21.5 muestra el uso del método Constantes() en una clase trivial.

**ENTRADA****LISTADO 21.5 Uso de Constantes()**

```
1: // Listado 21.5: Uso de Constantes()
2:
3: #include <iostream.h>
4: #include <string.h>
5: #define DEPURAR
6: #define MOSTRAR_CONSTANTES
7:
8: #ifndef DEPURAR
9: #define ASSERT(x)
10: #else
11:     #define ASSERT(x) if (! (x)) { cout << "¡¡ERROR!! Assert "
12:         << #x << " falló\n"; cout << " en la línea " << __LINE__ << "\n";
13:         << " del archivo " << __FILE__ << "\n"; }
14: #endif
15:
16: const int FALSE = 0;
17: const int TRUE = 1;
18: // typedef int bool;
19:
20: class Cadena
21: {
22:     public:
23:         // constructores
24:         Cadena();
25:         Cadena(const char * const);
26:         Cadena(const Cadena &);
```

```
26:         -Cadena();
27:         char & operator[](int offset);
28:         char operator[](int offset) const;
29:         Cadena & operator=(const Cadena &);
30:         int ObtenerLongitud() const
31:         { return suLongitud; }
32:         const char * ObtenerCadena() const
33:         { return suCadena; }
34:         bool Constantes() const;
35:     private:
36:         // constructor privado
37:         Cadena (int);
38:         char * suCadena;
39:         // unsigned short suLongitud;
40:         int suLongitud;
41:     };
42:
43: // constructor predeterminado crea una cadena de 0 bytes
44: Cadena::Cadena()
45: {
46:     suCadena = new char[ 1 ];
47:     suCadena[ 0 ] = '\0';
48:     suLongitud = 0;
49:     ASSERT(Constantes());
50: }
51:
52: // constructor privado (auxiliar), lo utilizan sólo los
53: // métodos de la clase para crear una nueva cadena del
54: // tamaño requerido. Se llena con caracteres nulos.
55: Cadena::Cadena(int longitud)
56: {
57:     suCadena = new char[ longitud + 1 ];
58:
59:     for (int i = 0; i <= longitud; i++)
60:         suCadena[ i ] = '\0';
61:     suLongitud = longitud;
62:     ASSERT(Constantes());
63: }
64:
65: // Convierte un arreglo de caracteres en una Cadena
66: Cadena::Cadena(const char * const cCadena)
67: {
68:     suLongitud = strlen(cCadena);
69:     suCadena = new char[ suLongitud + 1 ];
70:
71:     for (int i = 0; i < suLongitud; i++)
72:         suCadena[ i ] = cCadena[ i ];
73:     suCadena[ suLongitud ] = '\0';
74:     ASSERT(Constantes());
75: }
76:
77: // constructor de copia
```



**LISTADO 21.5 CONTINUACIÓN**

```
78:     Cadena::Cadena(const Cadena & rhs)
79:     {
80:         suLongitud = rhs.ObtenerLongitud();
81:         suCadena = new char[ suLongitud + 1 ];
82:
83:         for (int i = 0; i < suLongitud;i++)
84:             suCadena[ i ] = rhs[ i ];
85:         suCadena[ suLongitud ] = '\0';
86:         ASSERT(Constantes());
87:     }
88:
89:     // destructor, libera la memoria asignada
90:     Cadena::~Cadena ()
91:     {
92:         ASSERT(Constantes());
93:         delete [] suCadena;
94:         suLongitud = 0;
95:     }
96:
97:     // operador igual a, libera la memoria existente
98:     // y luego copia la cadena y el tamaño
99:     Cadena & Cadena::operator=(const Cadena & rhs)
100:    {
101:        ASSERT(Constantes());
102:        if (this == &rhs)
103:            return *this;
104:        delete [] suCadena;
105:        suLongitud = rhs.ObtenerLongitud();
106:        suCadena = new char[ suLongitud + 1 ];
107:        for (int i = 0; i < suLongitud; i++)
108:            suCadena[ i ] = rhs[ i ];
109:        suCadena[ suLongitud ] = '\0';
110:        ASSERT(Constantes());
111:        return *this;
112:    }
113:
114: //operador de desplazamiento no constante
115:     char & Cadena::operator[](int offset)
116:    {
117:        ASSERT(Constantes());
118:        if (offset > suLongitud)
119:        {
120:            ASSERT(Constantes());
121:            return suCadena[ suLongitud - 1 ];
122:        }
123:        else
124:        {
125:            ASSERT(Constantes());
126:            return suCadena[ offset ];
127:        }
128:    }
```

```
129:
130: // operador de desplazamiento constante
131: char Cadena::operator[](int offset) const
132: {
133:     ASSERT(Constantes());
134:     char valRet;
135:     if (offset > suLongitud)
136:         retVal = suCadena[ suLongitud - 1 ];
137:     else
138:         retVal = suCadena[ offset ];
139:     ASSERT(Constantes());
140:     return retVal;
141: }
142:
143: bool Cadena::Constantes() const
144: {
145: #ifdef MOSTRAR_CONSTANTES
146:     cout << "Constantes probadas";
147: #endif
148:     return ((suLongitud && suCadena) || (!suLongitud
149: -&& !suCadena));
150: }
151: class Animal
152: {
153: public:
154:     Animal():suEdad(1),suNombre("John Q. Animal")
155:     { ASSERT(Constantes()); }
156:     Animal(int, const Cadena &);
157:     ~Animal() {}
158:     int ObtenerEdad()
159:     { ASSERT(Constantes()); return suEdad; }
160:     void AsignarEdad(int Edad)
161:     {
162:         ASSERT(Constantes());
163:         suEdad = Edad;
164:         ASSERT(Constantes());
165:     }
166:     Cadena& ObtenerNombre()
167:     {
168:         ASSERT(Constantes());
169:         return suNombre;
170:     }
171:     void AsignarNombre(const Cadena & nombre)
172:     {
173:         ASSERT(Constantes());
174:         suNombre = nombre;
175:         ASSERT(Constantes());
176:     }
177:     bool Constantes();
178: private:
179:     int suEdad;
180:     Cadena suNombre;
```

**LISTADO 21.5** CONTINUACIÓN

```

181:     };
182:
183:     Animal::Animal(int edad, const Cadena & nombre):
184:         suEdad(edad),
185:         suNombre(nombre)
186:     {
187:         ASSERT(Constantes());
188:     }
189:
190:     bool Animal::Constantes()
191:     {
192:         #ifdef MOSTRAR_CONSTANTES
193:             cout << "Constantes probadas";
194:         #endif
195:         return (suEdad > 0 && suNombre.ObtenerLongitud());
196:     }
197:
198:     int main()
199:     {
200:         Animal sparky(5, "Sparky");
201:
202:         cout << "\n" << sparky.ObtenerNombre().ObtenerCadena();
203:         cout << " tiene " << sparky.ObtenerEdad();
204:         cout << " años de edad.\n";
205:         sparky.AsignarEdad(8);
206:         cout << "\n" << sparky.ObtenerNombre().ObtenerCadena();
207:         cout << " tiene " << sparky.ObtenerEdad();
208:         cout << " años de edad.\n";
209:         return 0;
210:     }

```

**SALIDA****MOSTRAR\_CONSTANTES definida**

Constantes probadasConstantes probadasConstantes probadasConstantes  
 ➔probadasConstantes  
 ➔probadasConstantes probadasConstantes probadasConstantes  
 probadasConstantes  
 ➔probadasConstantes probadasConstantes probadasConstantes  
 probadasConstantes  
 ➔ProbadasConstantes probadasConstantes probadasConstantes  
 probadasConstantes  
 ➔probadas  
 SparkyConstantes probadas tiene 5 años de edad.Constantes  
 probadasConstantes  
 ➔probadasConstantes probadas  
 SparkyConstantes probadas tiene 8 años de edad.Constantes probadas

**MOSTRAR\_CONSTANTES indefinida**

Sparky tiene 5 años de edad.  
 Sparky tiene 8 años de edad.

**ANÁLISIS** En la línea 11 se define la macro ASSERT(). Si se define DEPURAR, esto provocará que se escriba un mensaje de error cuando la macro ASSERT() se evalúe como false.

En la línea 34 se declara la función miembro Constantes() de la clase Cadena, y se define en las líneas 143 a 149. El constructor se declara en las líneas 44 a 50. En la línea 49, después de que el objeto está completamente construido, se llama a la función Constantes() para confirmar que la construcción sea apropiada.

Este patrón se repite para los otros constructores, y el destructor llama a Constantes() sólo antes de prepararse para destruir el objeto. Las funciones restantes de la clase llaman a Constantes() antes de realizar cualquier acción y luego otra vez antes de terminar. Esto afirma y valida un principio fundamental de C++: las funciones miembro que no sean constructores ni destructores deben funcionar sobre objetos válidos y deben dejarlos en un estado válido.

En la línea 177, la clase Animal declara su propio método Constantes(), el cual se implementa en las líneas 190 a 196. Observe que las funciones en línea pueden llamar al método Constantes() (líneas 155, 159, 162 y 164).

## Impresión de valores interinos

Además de utilizar la macro ASSERT() para afirmar que algo es cierto, tal vez necesite imprimir el valor actual de los apuntadores, variables y cadenas. Esto puede ser útil para ayudarlo a comprobar sus suposiciones acerca del progreso de su programa, y para localizar bugs en ciclos de tipo “se pasó por uno”. El listado 21.6 muestra esta idea.

### ENTRADA

### LISTADO 21.6 Impresión de valores en modo DEPURAR

```
1: // Listado 21.6 - Impresión de valores en modo DEPURAR
2:
3: #include <iostream.h>
4: #define DEPURAR
5:
6: #ifndef DEPURAR
7: #define IMPRIMIR(x)
8: #else
9: #define IMPRIMIR(x) cout << #x << ":\t" << x << endl;
10: #endif
11:
12: // enum bool { FALSE, TRUE } ;
13:
14:
15: int main()
16: {
17:     int x = 5;
18:     long y = 738981;
19:
```

21

**LISTADO 21.6** CONTINUACIÓN

```

20:      IMPRIMIR(x);
21:      for (int i = 0; i < x; i++)
22:      {
23:          IMPRIMIR(i);
24:      }
25:      IMPRIMIR(y);
26:      IMPRIMIR("Hola.");
27:      int *apx = &x;
28:      IMPRIMIR(apx);
29:      IMPRIMIR(*apx);
30:      return 0;
31:  }

```

**SALIDA**

```

x:      5
i:      0
i:      1
i:      2
i:      3
i:      4
y:      73898
"Hola.": Hola.
apx:      0x7fffc64
*apx:      5

```

**ANÁLISIS**

La macro que se encuentra en las líneas 6 a 9 proporciona la impresión del valor actual del parámetro proporcionado. Observe que lo primero que se proporciona a cout es la versión de cadena del parámetro; es decir, se pasa x y cout recibe "x".

A continuación, cout recibe la cadena entre comillas ":\t", la cual imprime el signo de dos puntos (:) y luego un tabulador. Luego, cout recibe el valor del parámetro (x), y finalmente recibe endl, el cual escribe una nueva línea y vacía el búfer.

Observe que puede recibir un valor distinto de 0x7fffc64.

## Niveles de depuración

En proyectos grandes y complejos, podría necesitar un mayor control que sólo activar y desactivar el modo DEPURAR. Puede definir niveles de depuración y probar esos niveles cuando decida qué macros utilizar y cuáles quitar.

Para definir un nivel, simplemente coloque un número después de la instrucción #define DEPURAR. Aunque puede tener cualquier número de niveles, un sistema común es tener cuatro: ALTO, MEDIO, BAJO y NINGUNO. El listado 21.7 muestra cómo se podría realizar esto, usando las clases Cadena y Animal del listado 21.5.

**ENTRADA****LISTADO 21.7 Niveles de depuración**

```

1: // Listado 21.7: Niveles de depuración
2:
3: #define ALTO 3
4: #define MEDIO 2
5: #define BAJO 1
6: #define NINGUNO 0
7: #define NIVELDEPURAR ALTO
8:
9: #include <iostream.h>
10: #include <string.h>
11:
12: #if NIVELDEPURAR < BAJO // debe ser medio o alto para definir ASSERT()
13: #define ASSERT(x)
14: #else
15: #define ASSERT(x) if (! (x)) { cout << "!!ERROR!! Assert "
   =><< #x << "\n"; cout << " en la linea " << __LINE__ << "\n";
   =>cout << " del archivo " << __FILE__ << "\n"; }
16: #endif
17:
18: #if NIVELDEPURAR < MEDIO
19: #define EVAL(x)
20: #else
21: #define EVAL(x) cout << #x << ":\t" << x << endl;
22: #endif
23:
24: #if NIVELDEPURAR < ALTO
25: #define IMPRIMIR(x)
26: #else
27: #define IMPRIMIR(x) cout << x << endl;
28: #endif
29:
30:
31: class Cadena
32: {
33:     public:
34:         // constructores
35:         Cadena();
36:         Cadena(const char * const);
37:         Cadena(const Cadena &);
38:         ~Cadena();
39:         char & operator[](int offset);
40:         char operator[](int offset) const;
41:         Cadena & operator=(const Cadena &);
42:         int ObtenerLongitud() const
43:             { return suLongitud; }
44:         const char * ObtenerCadena() const
45:             { return suCadena; }

```

**LISTADO 21.7** CONTINUACIÓN

```
46:         bool Constantes() const;
47:     private:
48:         // constructor privado
49:         Cadena (int);
50:         char * suCadena;
51:         unsigned short suLongitud;
52:     };
53:
54:     // constructor predeterminado crea una cadena de 0 bytes
55:     Cadena::Cadena()
56:     {
57:         suCadena = new char[ 1 ];
58:         suCadena[ 0 ] = '\0';
59:         suLongitud = 0;
60:         ASSERT(Constantes());
61:     }
62:
63:     // constructor privado (auxiliar), lo utilizan sólo
64:     // los métodos de la clase para crear una nueva cadena del
65:     // tamaño requerido. Se llena de caracteres nulos.
66:     Cadena::Cadena(int longitud)
67:     {
68:         suCadena = new char[ longitud + 1 ];
69:
70:         for (int i = 0; i <= longitud; i++)
71:             suCadena[ i ] = '\0';
72:         suLongitud = longitud;
73:         ASSERT(Constantes());
74:     }
75:
76:     // Convierte un arreglo de caracteres en una Cadena
77:     Cadena::Cadena(const char * const cCadena)
78:     {
79:         suLongitud = strlen(cCadena);
80:         suCadena = new char[ suLongitud + 1 ];
81:         for (int i = 0; i < suLongitud; i++)
82:             suCadena[ i ] = cCadena[ i ];
83:         suCadena[ suLongitud ] = '\0';
84:         ASSERT(Constantes());
85:     }
86:
87:     // constructor de copia
88:     Cadena::Cadena(const Cadena & rhs)
89:     {
90:         suLongitud = rhs.ObtenerLongitud();
91:         suCadena = new char[ suLongitud + 1 ];
92:
93:         for (int i = 0; i < suLongitud; i++)
94:             suCadena[ i ] = rhs[ i ];
```

```
95:         suCadena[ suLongitud ] = '\0';
96:         ASSERT(Constantes());
97:     }
98:
99:     // destructor, libera la memoria asignada
100:    Cadena::~Cadena ()
101:    {
102:        ASSERT(Constantes());
103:        delete [] suCadena;
104:        suLongitud = 0;
105:    }
106:
107:    // operador igual a, libera la memoria existente
108:    // luego copia la cadena y el tamaño
109:    Cadena & Cadena::operator=(const Cadena & rhs)
110:    {
111:        ASSERT(Constantes());
112:        if (this == &rhs)
113:            return *this;
114:        delete [] suCadena;
115:        suLongitud = rhs.ObtenerLongitud();
116:        suCadena = new char[ suLongitud + 1 ];
117:        for (int i = 0; i < suLongitud; i++)
118:            suCadena[ i ] = rhs[ i ];
119:        suCadena[ suLongitud ] = '\0';
120:        ASSERT(Constantes());
121:        return *this;
122:    }
123:
124:    //operador de desplazamiento no constante
125:    char & Cadena::operator[](int offset)
126:    {
127:        ASSERT(Constantes());
128:        if (offset > suLongitud)
129:        {
130:            ASSERT(Constantes());
131:            return suCadena[ suLongitud - 1 ];
132:        }
133:        else
134:        {
135:            ASSERT(Constantes());
136:            return suCadena[ offset ];
137:        }
138:    }
139:
140:    // operador de desplazamiento constante
141:    char Cadena::operator[](int offset) const
142:    {
143:        ASSERT(Constantes());
```

**LISTADO 21.7** CONTINUACIÓN

```
144:     char retval;
145:
146:     if (offset > suLongitud)
147:         retVal = suCadena[ suLongitud - 1 ];
148:     else
149:         retVal = suCadena[ offset ];
150:     ASSERT(Constantes());
151:     return retVal;
152: }
153:
154: bool Cadena::Constantes() const
155: {
156:     IMPRIMIR("(Constantes de Cadena probadas)");
157:     return ((bool) (suLongitud && suCadena) ||
158:             (!suLongitud && !suCadena));
159: }
160:
161: class Animal
162: {
163: public:
164:     Animal() : suEdad(1), suNombre("John Q. Animal")
165:     { ASSERT(Constantes()); }
166:     Animal(int, const Cadena &);
167:     ~Animal() {}
168:     int ObtenerEdad()
169:     {
170:         ASSERT(Constantes());
171:         return suEdad;
172:     }
173:     void AsignarEdad(int Edad)
174:     {
175:         ASSERT(Constantes());
176:         suEdad = Edad;
177:         ASSERT(Constantes());
178:     }
179:     Cadena& ObtenerNombre()
180:     {
181:         ASSERT(Constantes());
182:         return suNombre;
183:     }
184:     void AsignarNombre(const Cadena & nombre)
185:     {
186:         ASSERT(Constantes());
187:         suNombre = nombre;
188:         ASSERT(Constantes());
189:     }
190:     bool Constantes();
191: private:
192:     int suEdad;
```

```
193:     Cadena suNombre;
194: };
195:
196: Animal::Animal(int edad, const Cadena & nombre):
197:     suEdad(edad),
198:     suNombre(nombre)
199: {
200:     ASSERT(Constantes());
201: }
202:
203: bool Animal::Constantes()
204: {
205:     IMPRIMIR("(Constantes de Animal probadas)");
206:     return (suEdad > 0 && suNombre.ObtenerLongitud());
207: }
208:
209: int main()
210: {
211:     const int EDAD = 5;
212:
213:     EVAL(EDAD);
214:     Animal sparky(EDAD, "Sparky");
215:     cout << "\n" << sparky.ObtenerNombre().ObtenerCadena();
216:     cout << " tiene " << sparky.ObtenerEdad();
217:     cout << " años de edad.\n";
218:     sparky.AsignarEdad(8);
219:     cout << "\n" << sparky.ObtenerNombre().ObtenerCadena();
220:     cout << " tiene " << sparky.ObtenerEdad();
221:     cout << " años de edad.\n";
222:     return 0;
223: }
```

**SALIDA**

```
EDAD:      5
(Constantes de Cadena probadas)
```

(Constantes de Cadena probadas)  
 (Constantes de Cadena probadas)  
 (Constantes de Animal probadas)  
 (Constantes de Cadena probadas)  
 (Constantes de Animal probadas)

Sparky(Constantes de Animal probadas)  
 tiene 5 años de edad.  
 (Constantes de Animal probadas)  
 (Constantes de Animal probadas)  
 (Constantes de Animal probadas)

Sparky(Constantes de Animal probadas)  
 tiene 8 años de edad.  
 (Constantes de Cadena probadas)

Ejecutado de nuevo con DEPURAR = MEDIO

EDAD: 5  
 Sparky tiene 5 años de edad.  
 Sparky tiene 8 años de edad.

### ANÁLISIS

En las líneas 13 y 15 se define la macro ASSERT() para ser eliminada si NIVELDEPURAR es menor que BAJO (es decir, que NIVELDEPURAR sea NINGUNO). Si se permite cualquier nivel de depuración, la macro ASSERT() funcionará. En la línea 19 se declara EVAL() para ser eliminada si DEPURAR es menor que MEDIO; si NIVELDEPURAR es NINGUNO o BAJO, EVAL() se elimina.

Por último, en las líneas 24 a 28 se declara la macro IMPRIMIR para ser eliminada si NIVELDEPURAR es menor que ALTO. IMPRIMIR se utiliza sólo cuando NIVELDEPURAR sea ALTO; puede eliminar esta macro asignando el valor MEDIO a NIVELDEPURAR y seguir utilizando a EVAL() y a ASSERT().

IMPRIMIR se utiliza dentro de los métodos Constantes() para imprimir un mensaje informativo. EVAL() se utiliza en la línea 213 para evaluar el valor actual de la constante entera EDAD.

| DEBE                                                                                                                                               | No DEBE                                                                                                                                   |
|----------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| DEBE utilizar MAYÚSCULAS para sus nombres de macros. Ésta es una convención muy arraigada, y otros programadores se confundirán si no lo hace así. | NO DEBE permitir que sus macros tengan efectos secundarios. No incremente las variables ni asigne valores desde el interior de una macro. |
| DEBE encerrar entre paréntesis todos los argumentos de las funciones de macros.                                                                    |                                                                                                                                           |

## Manipulación de bits

A menudo es necesario establecer indicadores en sus objetos para mantener el registro del estado de su objeto. (¿Se encuentra en EstadoDeAlarma? ¿Se ha inicializado ya? ¿Va o viene?)

Puede hacer esto con valores booleanos definidos por el usuario, pero cuando tiene muchos indicadores, y cuando el espacio de almacenamiento es importante, es conveniente poder utilizar los bits individuales como indicadores.

Cada byte tiene ocho bits, por lo que en un valor de tipo `long` de 4 bytes se pueden guardar 32 indicadores individuales. Se dice que un bit está “encendido” si su valor es 1 y apagado si su valor es 0. Cuando enciende un bit, su valor se establece en 1, y cuando lo apaga se establece en 0. Puede encender y apagar bits cambiando el valor del tipo `long`, pero eso puede ser tedioso y confuso.

### Nota

El apéndice C, “Números binarios, octales, hexadecimales y una tabla de valores ASCII”, proporciona información adicional valiosa acerca de la manipulación de números binarios, octales y hexadecimales.

C++ proporciona operadores a nivel de bits que actúan sobre los bits individuales. Éstos se parecen a los operadores lógicos, pero son distintos de ellos, por lo que muchos programadores novatos los confunden. Los operadores a nivel de bits se presentan en la tabla 21.1.

**TABLA 21.1** Los operadores a nivel de bits

| Símbolo | Operador           |
|---------|--------------------|
| &       | AND                |
|         | OR                 |
| ^       | OR exclusivo (XOR) |
| -       | De complemento     |

### Operador AND

El operador AND (`&`) es un solo signo `&`, en contraste con el operador lógico AND, que está formado por dos (`&&`). Cuando se utiliza la operación AND sobre dos bits, el resultado es 1 si ambos bits son 1, y es 0 si uno de ellos o ambos son 0. Piense en esto de la siguiente manera: el resultado es 1 si el bit 1 y el bit 2 están encendidos.

## Operador OR

El segundo operador a nivel de bits es OR (|). De nuevo, es una sola barra vertical, en contraste con el operador lógico OR, que está formado por dos barras verticales (||). Cuando se utiliza la operación OR sobre dos bits, el resultado es 1 si cualquiera de los dos bits está encendido, o si los dos están encendidos.

## Operador OR exclusivo

El tercer operador a nivel de bits es el operador OR exclusivo (^), o XOR. Cuando se utiliza este operador sobre dos bits, el resultado es 1 si los dos bits son distintos.

## El operador de complemento

El operador de complemento (-) apaga todos los bits de un número que estén encendidos y enciende cada bit que esté apagado. Si el valor actual del número es 1010 0011, el complemento de ese número es 0101 1100.

## Cómo encender bits

Cuando se quiere encender o apagar un bit específico, se utilizan operaciones de enmascaramiento. Si tiene un indicador de 4 bytes y quiere que el bit 8 sea true, es decir, encenderlo, necesita poner el operador OR a nivel de bits entre el indicador y el valor 128. ¿Por qué? El valor 128 es 1000 0000 en binario; por lo tanto, el valor del octavo bit es 128. Cualquiera que sea el valor de ese bit (encendido o apagado), si pone un OR a nivel de bits entre él y el valor 128, encenderá ese bit y no afectará a ninguno de los otros bits. Suponga que el valor actual de una variable de 16 bits es 1010 0110 0010 0110. Si se pone un OR a nivel de bits entre los 16 bits y el valor 128, el resultado sería:

```
1010 0110 0010 0110 // el bit 8 está apagado  
| 0000 0000 1000 0000 // 128
```

```
1010 0110 1010 0110 // el bit 8 está encendido
```

Hay unas cuantas cosas que observar. En primer lugar, como siempre, los bits se cuentan de derecha a izquierda. En segundo lugar, el valor 128 se compone de puros ceros, excepto el bit 8, que es el que se quiere encender. En tercer lugar, el número inicial 1010 0110 0010 0110 queda sin cambios después de la operación OR, excepto que se encendió el bit 8. En caso de que el bit 8 ya hubiera estado encendido, habría permanecido encendido, que es lo que se quería.

## Cómo apagar bits

Si quiere apagar el bit 8, puede poner un operador AND a nivel de bits con el bit y con el complemento de 128. El complemento de 128 es el número que se obtiene al tomar el patrón de bits de 128 (1000 0000), encender todos los bits que estén apagados, y apagar todos los bits que estén encendidos (0111 1111). Cuando se pone el operador AND a nivel de bits con estos números, el número original permanece sin cambio, excepto por el octavo bit, el cual vale cero.

```
1010 0110 1010 0110 // el bit 8 está encendido
& 1111 1111 0111 1111 // -128
```

```
1010 0110 0010 0110 // el bit 8 está apagado
```

Para entender perfectamente esta solución, haga usted mismo la operación matemática. Cada vez que ambos bits sean 1, escriba 1 en su respuesta. Si cualquiera de los dos bits es 0, escriba 0 en la respuesta. Compare la respuesta con el número original. Debe ser la misma, excepto que se apagó el bit 8.

## Cómo invertir los bits

Por último, si desea invertir el valor del bit 8, sin importar el estado que tenga, ponga un operador OR exclusivo a nivel de bits con el número 128. Por ejemplo:

```
1010 0110 1010 0110 // número
^ 0000 0000 1000 0000 // 128
```

```
1010 0110 0010 0110 // bit invertido
^ 0000 0000 1000 0000 // 128
```

```
1010 0110 1010 0110 // valor original
```

| DEBE                                                                           | No DEBE |
|--------------------------------------------------------------------------------|---------|
| DEBE encender bits usando máscaras y el operador OR a nivel de bits.           |         |
| DEBE apagar bits usando máscaras y el operador AND a nivel de bits.            |         |
| DEBE invertir bits usando máscaras y el operador OR exclusivo a nivel de bits. |         |

## Campos de bits

Bajo ciertas circunstancias, cada byte cuenta, y guardar 6 u 8 bytes en una clase puede ser una gran diferencia. Si su clase o estructura tiene una serie de valores booleanos o variables que puedan tener sólo un número muy pequeño de valores posibles, puede ahorrar algo de espacio si utiliza campos de bits.

Utilice los tipos de datos estándar de C++, el tipo más pequeño que puede utilizar en su clase es el tipo `char`, que es de 1 byte. Por lo general, terminará utilizando un tipo `int`, el cual es de 2 o, más comúnmente, de 4 bytes. Al usar campos de bits, puede guardar 8 valores binarios en un `char` y 32 de esos valores en un `long`.

Los campos de bits funcionan de la siguiente manera: Se nombran y se acceden de la misma manera que cualquier miembro de una clase. Su tipo siempre se declara como entero sin signo (`unsigned int`). Despues del nombre del campo de bits escriba un signo de dos puntos (:) seguido de un número. El número es una instrucción para el compilador que le indica cuántos bits debe asignar a esta variable. Si escribe 1, el bit representará ya sea

el valor 0 o 1. Si escribe 2, el bit puede representar 0, 1, 2 o 3, es decir, un total de cuatro valores. Un campo de tres bits puede representar ocho valores, y así sucesivamente. En el apéndice C se da un repaso a los números binarios. El listado 21.8 muestra el uso de los campos de bits.

**ENTRADA****LISTADO 21.8 Uso de campos de bits**

```
1: // Listado 21.8: Uso de campos de bits
2:
3: #include <iostream.h>
4: #include <string.h>
5:
6: enum ESTADO { TiempoCompleto, TiempoParcial };
7: enum NIVELGRAD { NoGrad, Grad };
8: enum ALOJAMIENTO { Dorm, FueraDelCampus };
9: enum PLANALIMENT { UnaComida, TodasLasComidas, FinesDeSemana,
   ↪SinComidas };
10:
11:
12: class estudiante
13: {
14: public:
15: estudiante():
16:     miEstado(TiempoCompleto),
17:     miNivelGrad(NoGrad),
18:     miAlojamiento(Dorm),
19:     miPlanAliment(SinComidas)
20:     {}
21: ~estudiante() {}
22: ESTADO ObtenerEstado();
23: void AsignarEstado(ESTADO);
24: unsigned ObtenerPlan()
25: { return miPlanAliment; }
26: private:
27:     unsigned miEstado : 1;
28:     unsigned miNivelGrad : 1;
29:     unsigned miAlojamiento : 1;
30:     unsigned miPlanAliment : 2;
31: };
32:
33: ESTADO estudiante::ObtenerEstado()
34: {
35:     if (miEstado)
36:         return TiempoCompleto;
37:     else
38:         return TiempoParcial;
39: }
40:
41: void estudiante::AsignarEstado(ESTADO elEstado)
42: {
43:     miEstado = elEstado;
44: }
45:
46: int main()
```

```

47:         {
48:             estudiante Jim;
49:
50:             if (Jim.ObtenerEstado() == TiempoParcial)
51:                 cout << "Jim estudia tiempo parcial" << endl;
52:             else
53:                 cout << "Jim estudia tiempo completo" << endl;
54:             Jim.AsignarEstado(TiempoParcial);
55:             if (Jim.ObtenerEstado())
56:                 cout << "Jim estudia tiempo parcial" << endl;
57:             else
58:                 cout << "Jim estudia tiempo completo" << endl;
59:             cout << "Jim tiene el plan ";
60:
61:             char Plan[ 80 ];
62:             switch (Jim.ObtenerPlan())
63:             {
64:                 case UnaComida :
65:                     strcpy(Plan, "Una comida");
66:                     break;
67:                 case TodasLasComidas :
68:                     strcpy(Plan, "Todas las comidas");
69:                     break;
70:                 case FinesDeSemana :
71:                     strcpy(Plan,"Comidas en fin de semana");
72:                     break;
73:                 case SinComidas :
74:                     strcpy(Plan,"Sin comidas");
75:                     break;
76:                 default :
77:                     cout << "¡Algo salió mal!\n";
78:                     break;
79:             }
80:             cout << Plan << " de alimentación." << endl;
81:             return 0;
82:         }

```

**SALIDA**

Jim estudia tiempo parcial  
 Jim estudia tiempo completo  
 Jim tiene el plan Sin comidas de alimentación.

**ANÁLISIS**

En las líneas 6 a 9 se definen varios tipos enumerados. Éstos sirven para definir los valores posibles para los campos de bits dentro de la clase `estudiante`.

En las líneas 12 a 31 se declara `estudiante`. Aunque es una clase trivial, es interesante ya que toda la información está empacada en cinco bits. El primer bit representa el estado del estudiante, tiempo completo o tiempo parcial. El segundo bit representa si está graduado o no. El tercer bit representa si el estudiante vive en un dormitorio. Los últimos dos bits representan los cuatro posibles planes de alimentación.

Los métodos de la clase se escriben de la misma manera que para cualquier otra clase y no se afectan de ninguna manera por el hecho de que son campos de bits y no enteros o tipos enumerados.

La función miembro `ObtenerEstado()` lee el bit booleano y regresa un tipo enumerado, pero esto no es necesario. Se hubiera podido escribir con la misma facilidad para que regresara el valor del campo de bits directamente. El compilador hubiera hecho la traducción.

Para que usted mismo pruebe esto, reemplace la implementación de `ObtenerEstado()` con este código:

```
ESTADO estudiante::ObtenerEstado()
{
    return miEstado;
}
```

No debe ocurrir ningún cambio en el funcionamiento del programa. Es cuestión de claridad al leer el código; el compilador no es específico.

Observe que el código de la línea 50 debe comprobar el estado y luego imprimir el mensaje apropiado. Es tentador escribir esto:

```
cout << "Jim estudia " << Jim.ObtenerEstado() << endl;
```

Lo que imprimirá esto:

```
Jim estudia 0
```

El compilador no tiene manera de convertir la constante enumerada `TiempoParcial` en texto significativo.

En la línea 62, el programa utiliza una instrucción `switch` para el plan de alimentación, y para cada valor posible coloca un mensaje razonable en el búfer, que se imprime después en la línea 80. Observe de nuevo que la instrucción `switch` se hubiera podido escribir de la siguiente manera:

```
case 0: strcpy(Plan,"Una comida"); break;
case 1: strcpy(Plan,"Todas las comidas"); break;
case 2: strcpy(Plan,"Comidas en fin de semana"); break;
case 3: strcpy(Plan,"Sin comidas");break;
```

Lo más importante acerca del uso de campos de bits es que el cliente de la clase no necesita preocuparse por la implementación del almacenamiento de los datos. Como los campos de bits son privados, usted puede cambiarlos después y la interfaz no necesitará cambiar.

## Estilo de codificación

Como se ha dicho en otras partes de este libro, es importante adoptar un estilo consistente de codificación, aunque no importa mucho cuál estilo adopte. Un estilo consistente facilita prever lo que quiso dar a entender en cierta parte del código, y le evita tener que esforzarse en recordar si escribió la función con mayúscula al inicio la última vez que la invocó.

Los siguientes lineamientos sobre el estilo son arbitrarios; se basan en los lineamientos utilizados en mis proyectos anteriores, y parecen funcionar bien. Usted puede hacer los suyos, pero éstos le ayudarán a empezar.

Como dijo Emerson, “La mala consistencia es el duende de las mentes pequeñas”, pero tener algo de consistencia en su código es bueno. Haga su propio estilo de codificación, pero luego trátelo como si se lo hubieran enviado los dioses de la programación.

## Uso de sangrías

El tamaño de los tabuladores debe ser de tres o cuatro espacios. Asegúrese de que su editor convierta cada tabulador en tres o cuatro espacios.

## Llaves

La forma de alinear las llaves puede ser el tema más controversial entre los programadores de C y de C++. He aquí los tips que yo sugiero:

- Las llaves relacionadas se deben alinear verticalmente.
- El conjunto principal de llaves de una definición o de una declaración debe ir en el margen izquierdo. Las instrucciones que están en su interior deben tener sangría. Todos los demás conjuntos de llaves deben estar alineados con sus primeras instrucciones.
- No debe aparecer código en la misma línea donde aparezca una llave. Por ejemplo:

```
if (condicion==true)
{
    j = k;
    UnaFuncion();
}
m++;
```

## Líneas largas

Mantenga las líneas en un ancho que se pueda mostrar en una sola pantalla. El código que desaparece a la derecha de la pantalla puede pasar desapercibido fácilmente, y es muy molesto desplazarse horizontalmente para ver todo el código. Cuando divida una línea, use sangría en las siguientes líneas. Trate de dividir la línea en un lugar razonable, y trate de dejar el operador que intervenga al final de la línea anterior (en lugar de al principio de la siguiente línea) de forma que esté claro que la línea no es individual y que hay más a continuación.

En C++, las funciones tienden a ser más cortas de lo que eran en C, pero aún se aplica el antiguo y buen consejo. Trate de mantener sus funciones lo suficientemente cortas para que se pueda imprimir toda la función en una página.

## Instrucciones switch

Utilice sangría en las instrucciones switch de la siguiente manera para conservar el espacio horizontal:

```
switch(variable)
{
    case ValorUno:
```

```
    AccionUno();
    break;
case ValorDos:
    AccionDos();
    break;
default:
    ASSERT("mala acción");
    break;
}
```

## Texto del programa

Puede usar varias sugerencias para crear código fácil de leer. El código fácil de leer es fácil de mantener.

- Utilice espacios en blanco para favorecer la legibilidad.
- Los objetos y los arreglos en realidad se refieren a una cosa. No utilice espacios dentro de referencias a objetos (.. ->, [ ]).
- Los operadores unarios se asocian con sus operandos, así que no coloque un espacio entre ellos. Pero sí coloque un espacio a un lado del operando. Los operadores unarios incluyen a !, ~, ++, --, ., \* (para apuntadores), & (conversiones) y sizeof.
- Coloque espacios en ambos lados de los operadores binarios: +, =, \*, /, %, >>, <<, <, >, ==, !=, &, |, &&, ||, ?:, =, +=, etc.
- No prescinda de los espacios para indicar la precedencia (4+ 3\*2).
- Coloque un espacio después de las comas y los signos de punto y coma, no antes.
- No ponga espacios en ninguno de los dos lados de los paréntesis.
- Separe las palabras reservadas, como if, con un espacio: if (a == b).
- El cuerpo de un comentario se debe separar de los símbolos // con un espacio.
- Coloque el apuntador o indicador de referencia en seguida del nombre del tipo, y no antes del nombre de la variable:  
`char* foo;`  
`int& elInt;`  
en vez de  
`char *foo;`  
`int &elInt;`
- No declare más de una variable en la misma línea, a menos que esté muy relacionada con la o las otras variables de esa línea.

## Nombres de identificadores

Los siguientes son lineamientos para trabajar con identificadores:

- Los nombres de identificadores deben ser lo suficientemente largos para ser descriptivos.
- Evite las abreviaciones enigmáticas.
- Tómese el tiempo y la energía necesarios para aclarar las cosas.
- Yo no utilizo la notación húngara. C++ está fuertemente tipificado y no hay razón para colocar el tipo en el nombre de la variable. Con los tipos definidos por el usuario (clases), la notación húngara se vuelve ineficiente muy pronto. Las excepciones a esto pueden ser utilizar un prefijo para los apuntadores (`ap`) y las referencias (`r`), así como para las variables miembro de una clase (`ints`). Por otro lado, algunas personas están a favor de la notación húngara.
- Los nombres cortos (`i`, `p`, `x`, y así sucesivamente) se deben utilizar sólo donde su brevedad haga que el código sea más legible y donde su uso sea tan obvio que no se necesite un nombre descriptivo.
- La longitud del nombre de una variable debe ser proporcional con su alcance.
- Asegúrese de que los identificadores se vean y suenen de manera distinta unos de otros para minimizar la confusión.
- Por lo general, los nombres de funciones (o métodos) son verbos o frases compuestas de verbo-sustantivo: `Buscar()`, `Restablecer()`, `EncontrarParrago()`, `MostrarCursor()`. Asimismo, los nombres de variables son sustantivos abstractos, posiblemente con un sustantivo adicional: `cuenta`, `estado`, `velocidadViento`, `alturaVentana`. Las variables booleanas deben ser nombradas en forma apropiada: `ventanaMinimizada`, `archivoEstaAbierto`.

## Ortografía y uso de mayúsculas en los nombres

Al crear su propio estilo, no debe descuidar la ortografía y el uso de mayúsculas. Algunas sugerencias para estas áreas son:

- Utilice sólo mayúsculas y guiones bajos para separar las palabras lógicas de los nombres, como `PLANTILLA_ARCHIVO_FUENTE`. Sin embargo, tome en cuenta que este tipo de nombres es raro en C++. Considere el uso de constantes y plantillas en la mayoría de los casos.
- Escriba los nombres de macros sólo con mayúsculas.
- Todos los demás identificadores pueden ser una mezcla de minúsculas y mayúsculas sin guion bajo, o un solo tipo de letra (mayúscula o minúscula, por lo general minúscula) con guiones bajos. Al utilizar mezcla de mayúsculas y minúsculas, empiece con una letra mayúscula los nombres de funciones, métodos, clases, tipos definidos y nombres de estructuras, y con letra minúscula los elementos, como datos miembro o locales.

- Empiece con unas cuantas letras minúsculas las constantes enumeradas, como abreviación para el enum. Por ejemplo:

```
enum EstiloTexto
{
    etSimple,
    etNegrita,
    etCursiva,
    etSubrayado,
};
```

## Comentarios

Los comentarios pueden facilitar la comprensión de un programa. Algunas veces no trabajará con un programa durante varios días, o incluso meses. Durante ese tiempo puede olvidar lo que cierto código hace o por qué se ha incluido. Los problemas de comprensión de código también pueden ocurrir cuando alguien más lee su código. Los comentarios que se aplican en un estilo consistente y bien analizado pueden ser muy valiosos. Algunas sugerencias en relación con los comentarios son:

- Siempre que sea posible, utilice comentarios estilo C++ (//) en lugar de los de estilo /\* \*/.
- Los comentarios de nivel más alto son definitivamente más importantes que los detalles del proceso. Agregue valor; no sólo vuelva a decir lo que hace el código.  
`n++; // n se incrementa en uno`
- Este comentario no vale el tiempo que toma escribirlo. Concéntrese en la semántica de las funciones y de los bloques de código. Diga lo que hace una función e indique los efectos secundarios, tipos de parámetros y valores de retorno. Describa todas las suposiciones que se hacen (o que no se hacen), como "supone que n no es negativo" o "regresará -1 si x no es válida". Dentro de la lógica compleja, utilice comentarios que indiquen las condiciones que existen en ese punto del código.
- Utilice enunciados completos en español con la puntuación y el uso de mayúsculas apropiados. La escritura adicional vale la pena. No sea demasiado enigmático y no abrevie. Lo que le parece demasiado claro al estar escribiendo el código, le será sorprendentemente confuso en unos cuantos meses.
- Utilice líneas en blanco para ayudar al lector a comprender lo que está pasando. Separe las instrucciones en grupos lógicos.

## Acceso

La manera en que acceda a porciones de su programa también debe ser consistente. Algunas sugerencias para el acceso son:

- Utilice siempre etiquetas `public::`, `private::` y `protected::`; no se base en los valores predeterminados.
- Enliste primero los miembros públicos, después los protegidos y luego los privados. Ponga los datos miembro en un grupo después de los métodos.
- Coloque primero el(los) constructor(es) en la sección apropiada, después del destructor. Enliste los métodos sobrecargados con el mismo nombre adyacentes uno con otro. Agrupe las funciones de acceso siempre que pueda.
- Considere la alfabetización de los nombres de los métodos dentro de cada grupo, así como la alfabetización de las variables miembro. Asegúrese de alfabetizar los nombres de archivos en directivas `#include`.
- Utilice la palabra reservada `virtual`, aunque su uso sea opcional al redefinir; esto le ayuda a recordar que es virtual, y además mantiene la consistencia en la declaración.

## Definiciones de clases

Trate de mantener las definiciones de los métodos en el mismo orden que las declaraciones. Esto hace que las cosas sean más fáciles de encontrar.

Al definir una función, coloque el tipo de valor de retorno y los otros modificadores en una línea anterior, para que el nombre de la clase y el nombre de la función empiecen en el margen izquierdo. Esto facilita mucho la búsqueda de funciones.

## Archivos de encabezado

Trate, lo más que pueda, de no incluir archivos en archivos de encabezado. El mínimo ideal es el archivo de encabezado para la clase de la que se deriva. Otros archivos `include` obligatorios son los de los objetos que son miembros de la clase que se está declarando. Las clases a las que simplemente se apunta o se hace una referencia sólo necesitan referencias posteriores de la forma.

No omita un archivo `include` en un encabezado sólo porque suponga que cualquier archivo `.cxx` que lo incluya también tendrá el `include` necesario.

### Tip

Todos los archivos de encabezado deben utilizar guardias de inclusión.

21

## ASSERT()

Utilice la macro `ASSERT()` libremente. Ayuda a encontrar errores, pero también ayuda mucho a que el lector entienda con claridad cuáles son las suposiciones. También ayuda a enfocar los pensamientos del escritor en lo que es y lo que no es válido.

## **const**

Utilice **const** en donde sea apropiado: para parámetros, variables y métodos. A menudo existe la necesidad tanto de un método **const** como de uno que no sea **const**; no utilice esto como excusa para omitir uno. Sea muy cuidadoso al convertir explícitamente de **const** a **no const** y viceversa (algunas veces, ésta es la única forma de hacer algo), pero asegúrese de que tenga sentido, e incluya un comentario.

## **Los siguientes pasos**

Acaba de pasar tres largas y duras semanas trabajando con C++, y ahora es un programador de C++ competente, pero esto de ninguna manera significa que haya terminado. Hay muchas más cosas que debe aprender, y puede obtener información valiosa de muchas más fuentes al avanzar de programador novato de C++ a experto.

Las siguientes secciones recomiendan varias fuentes específicas de información, y estas recomendaciones reflejan sólo mi experiencia y opinión personales. Sin embargo, hay docenas de libros disponibles acerca de cada uno de estos temas, así que asegúrese de obtener otras opiniones antes de empezar a comprar más libros.

## **Dónde obtener ayuda y orientación**

Lo primero que querrá hacer como programador de C++ será entrar a una de las conferencias sobre C++ en un servicio en línea. Estos grupos proporcionan un contacto inmediato con cientos de miles de programadores de C++ que pueden contestar sus preguntas, darle consejos y proporcionarle un portavoz para sus ideas.

Muchos programadores participan en los grupos de noticias sobre C++ en Internet (`comp.lang.c++` y `comp.lang.c++.moderated`), y se los recomiendo como excelentes fuentes de información y de soporte.

También puede buscar grupos de usuarios locales. Muchas ciudades tienen grupos interesantes de C++ en donde puede conocer otros programadores e intercambiar ideas.

## **Revistas**

También puede reforzar sus habilidades suscribiéndose a una buena revista sobre programación en C++. Algunas de las mejores revistas en relación con este tema son: *C++ Report* de SIGS Publications y *C/C++ Users Journal* de Miller Freeman. Cada número tiene artículos útiles. Guárdelos; lo que no le preocupa hoy puede ser de vital importancia mañana.

## Manténgase en contacto

Si tiene comentarios, sugerencias o ideas sobre este u otros libros, puede ponerse en contacto con los autores. Puede comunicarse con Jesse Liberty en la dirección [jliberty@libertyassociates.com](mailto:jliberty@libertyassociates.com) (página Web <http://www.libertyassociates.com>). Puede ponerse en contacto con David Horvath en la dirección [cpplinux@cobs.com](mailto:cpplinux@cobs.com) (página Web <http://www.cobs.com/~dhorvath>).

| DEBE                                                                                                                      | NO DEBE                                                                                           |
|---------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| <b>DEBE</b> consultar otros libros. Hay mucho que aprender y un solo libro no puede enseñarle todo lo que necesita saber. | <b>¡NO DEBE</b> sólo leer código! La mejor forma de aprender C++ es escribiendo programas de C++. |
| <b>DEBE</b> suscribirse a una buena revista de C++ y unirse a un buen grupo de usuarios de C++.                           |                                                                                                   |

## Su próximo paso

El próximo paso que debe realizar, después de descansar un poco, es aprender más sobre los sangrientos detalles de C++ y de Linux. ¡La semana adicional de este libro lo guiará!

## Resumen

Hoy aprendió más detalles sobre la forma de trabajar con el preprocesador. Cada vez que ejecuta el compilador, primero se ejecuta el preprocesador y traduce sus directivas de preprocesador, como `#define` e `#ifdef`.

El preprocesador realiza la sustitución del texto, aunque con el uso de las macros esto puede ser un poco complejo. Al usar `#ifdef`, `#else` e `#ifndef` puede realizar la compilación condicional, compilando instrucciones específicas bajo un conjunto de condiciones y otro conjunto de instrucciones bajo otras condiciones. Esto puede ayudarle a escribir programas para más de una plataforma y por lo regular se utiliza para incluir condicionalmente información de depuración.

Las funciones de macros proporcionan una sustitución compleja de texto basada en argumentos pasados a la macro en tiempo de compilación. Es importante colocar paréntesis alrededor de todos los argumentos de la macro para asegurar que se haga la sustitución adecuada.

En C++, las funciones de macros, y el preprocesador en general, son menos importantes de lo que eran en C. C++ proporciona una variedad de características del lenguaje, como variables `const` y plantillas, que ofrecen alternativas superiores al uso del preprocesador.

También aprendió cómo encender y probar bits individuales y cómo asignar un número limitado de bits a los miembros de una clase.

Por último, se trataron cuestiones relacionadas con el estilo de C++, y se proporcionaron recursos para su posterior estudio.

## Preguntas y respuestas

- P Si C++ ofrece mejores alternativas que el preprocesador, ¿por qué esta opción aún está disponible?**
- R** En primer lugar, C++ tiene compatibilidad con C, por lo que todas las partes importantes de C deben ser soportadas en C++. En segundo lugar, algunos usos del preprocesador se siguen utilizando con frecuencia en C++, como los guardias de inclusión, por ejemplo.
- P ¿Por qué utilizar funciones de macros si se puede usar una función normal?**
- R** Las funciones de macros se expanden en línea y se utilizan como sustituto para la escritura repetitiva de los mismos comandos con pequeñas variaciones. Sin embargo, las plantillas ofrecen una mejor alternativa.
- P ¿Cómo se sabe cuándo usar una macro y cuándo usar una función en línea?**
- R** Por lo general, esto no es muy importante; utilice la que sea más simple. Sin embargo, las macros ofrecen la sustitución de caracteres, de cadenas y la concatenación. Ninguna de estas tres está disponible en las funciones.
- P ¿Cómo se puede usar el preprocesador para imprimir valores interinos durante la depuración?**
- R** La mejor alternativa es utilizar instrucciones `watch` dentro de un depurador. Para obtener información sobre las instrucciones `watch`, consulte el manual de `g++` y `gdb` y el archivo de información (o la documentación que venga con su compilador o depurador, si utiliza algo distinto de las herramientas GNU).
- P ¿Cómo se puede decidir cuándo utilizar una macro `ASSERT()` y cuándo producir una excepción?**
- R** Si la situación que está probando puede ser verdadera sin que haya cometido un error de programación, use una excepción. Si la única razón de que esta situación sea verdadera es un bug en su programa, utilice una macro `ASSERT()`.
- P ¿Cuándo se deben utilizar estructuras de bits, en lugar de simplemente usar enteros?**
- R** Cuando el tamaño del objeto sea muy importante. Si está trabajando con memoria limitada o con software de comunicaciones, tal vez descubra que los ahorros ofrecidos por estas estructuras son esenciales para el éxito de su producto.

**P ¿Por qué generan tanta emoción las guerras de estilo?**

**R** Los programadores se apegan mucho a sus hábitos. Si usted está acostumbrado a utilizar el siguiente estilo de sangría:

```
if (UnaCondicion){  
    // instrucciones  
} // llave de cierre
```

es muy difícil dejar este hábito. Los nuevos estilos se ven mal y crean confusión. Si se harta, pruebe entrando a un servicio en línea que sea popular y pregunte qué estilo de sangría funciona mejor, qué editor es el mejor para C++, o qué producto es el mejor procesador de palabras. Luego siéntese a observar cómo se generan 10 mil mensajes, todos contradiciéndose entre sí.

**P ¿Qué es lo mejor que hay para leer después de este libro?**

**R** He aquí algunos libros que he escrito para proporcionar un curso de estudio, aunque hay muchos otros libros de gran valor. *C++ Unleashed*, *Beginning Object-Oriented Analysis and Design* y *Career Change C++* son buenos puntos para comenzar.

**P ¿Es todo?**

**R** ¡Sí! Ya ha aprendido C++, pero...¡no, todavía no termina! Hace 10 años era posible que una persona aprendiera todo lo que había que saber acerca de las computadoras y la programación, o por lo menos que confiara en estar cerca de ello. Hoy eso queda fuera de cuestionamiento. Simplemente no se puede ponerse al corriente, porque mientras usted tratar de hacerlo, la industria va cambiando. Pero asegúrese de seguir leyendo, y manténgase en contacto con los recursos (revistas y servicios en línea) que lo mantendrán al corriente de los cambios más recientes.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, "Respuestas a los cuestionarios y ejercicios", y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Qué es un guardia de inclusión?
2. ¿Cómo le indica al compilador que imprima el contenido del archivo intermedio, para que muestre los efectos del preprocesador?
3. ¿Cuál es la diferencia entre `#define depurar 0` y `#undef depurar`?

4. ¿Qué hace el operador de complemento (~) a nivel de bits?
5. ¿Cuál es la diferencia entre OR y OR exclusivo (XOR)?
6. ¿Cuál es la diferencia entre & y &&?
7. ¿Cuál es la diferencia entre | y ||?

## Ejercicios

1. Escriba las instrucciones de guardias de inclusión para el archivo de encabezado `STRING.H`.
2. Escriba una macro `ASSERT()` que imprima tanto un mensaje de error como el archivo y el número de línea si el nivel de depuración es 2, que imprima un mensaje (sin archivo ni número de línea) si el nivel es 1, y que no haga nada si el nivel es 0.
3. Escriba una macro llamada `DImprimir` que evalúe si `DEPURAR` está definida y, de ser así, que imprima el valor que se pasa como parámetro.
4. Escriba un programa que sume dos números sin utilizar el operador de suma (+).  
Pista: ¡Use los operadores a nivel de bits!

# SEMANA 3

## Repasso

El programa de repaso de la semana 3, que se muestra en el listado R3.1, reúne muchas de las técnicas avanzadas que aprendió durante las últimas tres semanas de trabajo duro. El repaso de la semana 3 proporciona una lista enlazada basada en plantillas con manejo de excepciones. Examine este programa con detalle; si lo entiende sin problemas, entonces usted es un programador de C++.

### Precaución

Necesita utilizar una versión reciente de los compiladores GNU (2.9.5 o posterior) para que este ejemplo funcione.

Si su compilador no soporta el uso de plantillas, o si no soporta las instrucciones try y catch, no podrá compilar ni ejecutar este listado.

### ENTRADA

### LISTADO R3.1 Listado de repaso de la semana 3

```
1:  //  
*****  
2:  //  
3:  // Title: Repaso de la semana 3  
4:  //  
5:  // File: Semana3  
6:  //  
7:  // Descripción: Proporcionar un programa de  
→muestra de una lista  
8:  // enlazada basada en plantillas con manejo de  
→excepciones  
9:  //  
10: // Clases: PIEZA - guarda números de piezas y  
→potencialmente otra  
11: // información sobre las piezas. Ésta será la  
→clase de
```

*continua*

8

9

10

11

12

13

14

**LISTADO R3.1 CONTINUACIÓN**

```
12: // ejemplo que la lista guardará
13: // Observe el uso de operator<< para imprimir la
14: // información acerca de una pieza con base en su
15: // tipo en tiempo de ejecución.
16: //
17: // Nodo - actúa como un nodo de la lista
18: //
19: // Lista - lista basada en plantilla que proporciona los
20: // mecanismos para una lista enlazada
21: //
22: //
23: // Autor: Jesse Liberty (jl)
24: //
25: // Desarrollado en: Pentium 200 Pro. 128MB RAM MVC 5.0
26: //
27: // Destino: Independiente de la plataforma
28: //
29: //
30: //
31: // ****
32: #include <iostream.h>
33:
34:
35: // clases de excepciones
36: class Excepcion {};
37: class NoHayMemoria : public Excepcion {};
38: class NodoNulo : public Excepcion {};
39: class ListaVacia : public Excepcion {};
40: class ErrorLimites : public Excepcion {};
41:
42: // **** Pieza ****
43: // Clase base abstracta de piezas
44: class Pieza
45: {
46: public:
47:     Pieza() : suNumeroObjeto(1) {}
48:     Pieza(int NumeroObjeto) :
49:         suNumeroObjeto(NumeroObjeto) {}
50:     virtual ~Pieza() {};
51:     int ObtenerNumeroObjeto() const
52:     { return suNumeroObjeto; }
53:     // se debe redefinir el siguiente método
54:     virtual void Mostrar()const = 0;
55: private:
56:     int suNumeroObjeto;
57: };
58:
59: // implementación de función virtual pura para que
60: // las clases derivadas se puedan encadenar
61: void Pieza::Mostrar() const
```

```
62:  {
63:      cout << "\nNúmero de pieza: "
64:          << suNumeroObjeto << endl;
65:  }
66:
67: // este operator<< será llamado para todos los objetos Pieza.
68: // No necesita ser amigo ni tener acceso a los datos privados
69: // Llama a Mostrar(), el cual utiliza el polimorfismo requerido
70: // Nos gustaría poder redefinirlo con base en el tipo real
71: // de la Pieza, pero C++ no soporta la contravarianza
72: ostream & operator<< (ostream & elFlujo, Pieza & laPieza)
73: {
74:     // icontravarianza virtual!
75:     laPieza.Mostrar();
76:     return elFlujo;
77: }
78:
79: // ***** Pieza de Auto *****
80: class PiezaAuto : public Pieza
81: {
82: public:
83:     PiezaAuto() : SuAnioModelo(94) {}
84:     PiezaAuto(int anio, int numeroPieza);
85:     int ObtenerAnioModelo() const
86:     { return SuAnioModelo; }
87:     virtual void Mostrar() const;
88: private:
89:     int SuAnioModelo;
90: };
91:
92: PiezaAuto::PiezaAuto(int anio, int numeroPieza):
93:     SuAnioModelo(anio),
94:     Pieza(numeroPieza)
95: {}
96:
97: void PiezaAuto::Mostrar() const
98: {
99:     Pieza::Mostrar();
100:    cout << "Año del modelo: "
101:        << SuAnioModelo << endl;
102: }
103:
104: // ***** Pieza de AeroPlano *****
105: class PiezaAeroPlano : public Pieza
106: {
107: public:
108:     PiezaAeroPlano() : suNumeroMotor(1) {};
109:     PiezaAeroPlano(int NumeroMotor,
110:                     int NumeroPieza);
111:     virtual void Mostrar() const;
112:     int ObtenerNumeroMotor()const
```

*continúa*

**LISTADO R3.1** CONTINUACIÓN

```
113:     { return suNumeroMotor; }
114: private:
115:     int suNumeroMotor;
116: };
117:
118: PiezaAeroPlano::PiezaAeroPlano(int NumeroMotor,
119:                                 int NumeroPieza):
120:     suNumeroMotor(NumeroMotor),
121:     Pieza(NumeroPieza)
122: {}
123:
124: void PiezaAeroPlano::Mostrar() const
125: {
126:     Pieza::Mostrar();
127:     cout << "Número de motor: "
128:         << suNumeroMotor << endl;
129: }
130:
131: // adelantar declaración de la clase Lista
132: template < class T >
133: class Lista;
134:
135: // ***** Nodo *****
136: // Nodo genérico, se puede agregar a una lista
137: // *****
138:
139: template < class T >
140: class Nodo
141: {
142: public:
143:     friend class Lista< T >;
144:     Nodo (T *);
145:     ~Nodo();
146:     void AsignarSiguiente(Nodo * node)
147:     { suSiguiente = node; }
148:     Nodo * ObtenerSiguiente() const;
149:     T * ObtenerObjeto() const;
150: private:
151:     T * suObjeto;
152:     Nodo * suSiguiente;
153: };
154:
155: // Implementaciones de Nodo...
156:
157: template < class T >
158: Nodo< T >::Nodo(T * apObjeto):
159:     suObjeto(apObjeto),
160:     suSiguiente(0)
161: {}
162:
163: template < class T >
```

```
164: Nodo<T>::~Nodo()
165: {
166:     delete suObjeto;
167:     suObjeto = 0;
168:     delete suSiguiente;
169:     suSiguiente = 0;
170: }
171:
172: // Regresa NULL si no hay Nodo siguiente
173: template < class T >
174: Nodo< T > * Nodo< T >::ObtenerSiguiente() const
175: {
176:     return suSiguiente;
177: }
178:
179: template < class T >
180: T * Nodo< T >::ObtenerObjeto() const
181: {
182:     if (suObjeto)
183:         return suObjeto;
184:     else
185:         throw NodoNulo();
186: }
187:
188: // ***** Lista *****
189: // Plantilla de lista genérica
190: // Funciona con cualquier objeto numerado
191: // *****
192: template < class T >
193: class Lista
194: {
195: public:
196:     Lista();
197:     ~Lista();
198:     T * Buscar(int & posicion,
199:                int NumeroObjeto) const;
200:     T * ObtenerPrimero() const;
201:     void Insertar(T *);
202:     T * operator[] (int) const;
203:     int ObtenerCuenta() const
204:     { return suCuenta; }
205: private:
206:     Nodo< T > * apCabeza;
207:     int suCuenta;
208: };
209:
210: // Implementaciones para las Listas...
211: template < class T >
212: Lista< T >::Lista():
213:     apCabeza(0),
214:     suCuenta(0)
215: {}
216:
```

**LISTADO R3.1 CONTINUACIÓN**

```
217: template < class T >
218: Lista< T >::~Lista()
219: {
220:     delete apCabeza;
221: }
222:
223: template < class T >
224: T * Lista< T >::ObtenerPrimero() const
225: {
226:     if (apCabeza)
227:         return apCabeza->suObjeto;
228:     else
229:         throw ListaVacia();
230: }
231:
232: template < class T >
233: T * Lista< T >::operator[] (int offSet) const
234: {
235:     Nodo< T > * apNodo = apCabeza;
236:
237:     if (!apCabeza)
238:         throw ListaVacia();
239:     if (offSet > suCuenta)
240:         throw ErrorLimites();
241:     for (int i = 0; i < offSet; i++)
242:         apNodo = apNodo->suSiguiente;
243:     return apNodo->suObjeto;
244: }
245:
246: // Buscar un objeto dado en una lista con base en su número único (id)
247: template < class T >
248: T * Lista< T >::Buscar(int & posicion,
249:                         int NumeroObjeto) const
250: {
251:     Nodo< T > * apNodo = NULL;
252:     for (apNodo = apCabeza, posicion = 0;
253:          apNodo!=NULL;
254:          apNodo = apNodo->suSiguiente, posicion++)
255:     {
256:         if (apNodo->suObjeto->ObtenerNumeroObjeto() == NumeroObjeto)
257:             break;
258:     }
259:     if (apNodo == NULL)
260:         return NULL;
261:     else
262:         return apNodo->suObjeto;
263: }
264:
265: // insertar si el número del objeto es único
266: template < class T >
267: void Lista< T >::Insertar(T * apObjeto)
```

```
268: {
269:     Nodo< T > * apNodo = new Nodo< T >(apObjeto);
270:     Nodo< T > * apActual = apCabeza;
271:     Nodo< T > * apSiguiente = NULL;
272:
273:     int Nuevo = apObjeto->ObtenerNumeroObjeto();
274:     int Siguiente = 0;
275:     suCuenta++;
276:
277:     if (!apCabeza)
278:     {
279:         apCabeza = apNodo;
280:         return;
281:     }
282:
283: // si éste es más pequeño que el nodo cabeza
284: // entonces se convierte en el nuevo nodo cabeza
285: if (apCabeza->suObjeto->ObtenerNumeroObjeto() > Nuevo)
286: {
287:     apNodo->suSiguiente = apCabeza;
288:     apCabeza = apNodo;
289:     return;
290: }
291:
292: for (;;)
293: {
294:     // si no hay siguiente, agregar éste
295:     if (!apActual->suSiguiente)
296:     {
297:         apActual->suSiguiente = apNodo;
298:         return;
299:     }
300:
301:     // si va después de éste y antes del siguiente
302:     // entonces insertarlo aquí, de no ser así
303:     // obtener el siguiente
304:     apSiguiente = apActual->suSiguiente;
305:     Siguiente = apSiguiente->suObjeto->
306:                 ObtenerNumeroObjeto();
307:     if (Siguiente > Nuevo)
308:     {
309:         apActual->suSiguiente = apNodo;
310:         apNodo->suSiguiente = apSiguiente;
311:         return;
312:     }
313:     apActual = apSiguiente;
314: }
315: }
316:
317:
318: int main()
319: {
320:     Lista< Pieza > laPieza;
```

*continúa*

**LISTADO R3.1** CONTINUACIÓN

```
321:     int opcion;
322:     int NumeroObjeto;
323:     int valor;
324:     Pieza * apPieza;
325:
326:     while (1)
327:     {
328:         cout << "(0)Salir (1)Auto (2)Avión: ";
329:         cin >> opcion;
330:
331:         if (!opcion)
332:             break;
333:         cout << "¿Nuevo NúmeroPieza?: ";
334:         cin >> NumeroObjeto;
335:
336:         if (opcion == 1)
337:         {
338:             cout << "¿Año del modelo?: ";
339:             cin >> valor;
340:             try
341:             {
342:                 apPieza = new PiezaAuto(valor,
343:   NumeroObjeto);
344:             }
345:             catch (NoHayMemoria)
346:             {
347:                 cout << "No hay suficiente memoria."
348:                     << " Saliendo..." << endl;
349:                 return 1;
350:             }
351:         }
352:         else
353:         {
354:             cout << "Número de motor?: ";
355:             cin >> valor;
356:             try
357:             {
358:                 apPieza = new PiezaAeroPlano(valor,
359:   NumeroObjeto);
360:             }
361:             catch (NoHayMemoria)
362:             {
363:                 cout << "No hay suficiente memoria."
364:                     << " Saliendo..." << endl;
365:                 return 1;
366:             }
367:         }
368:         try
369:         {
```

```
370:         laPieza.Insertar(apPieza);
371:     }
372:     catch (NodoNulo)
373:     {
374:         cout << "¡La lista está dividida,"
375:             << " y el nodo es nulo!" << endl;
376:         return 1;
377:     }
378:     catch (ListaVacia)
379:     {
380:         cout << "¡La lista está vacía!" << endl;
381:         return 1;
382:     }
383: }
384: try
385: {
386:     for (int i = 0; i < laPieza.ObtenerCuenta(); i++)
387:         cout << *(laPieza[ i ]);
388: }
389: catch (NodoNulo)
390: {
391:     cout << "¡La lista está dividida,"
392:         << " y el nodo es nulo!" << endl;
393:     return 1;
394: }
395: catch (ListaVacia)
396: {
397:     cout << "¡La lista está vacía!" << endl;
398:     return 1;
399: }
400: catch (ErrorLimites)
401: {
402:     cout << "¡Trató de leer más allá "
403:         << "del final de la lista!" << endl;
404:     return 1;
405: }
406: return 0;
407: }
```

**SALIDA**      (0)Salir (1)Auto (2)Avión: 1  
¿Nuevo NumeroPieza?: 2837  
¿Año del modelo?: 90

(0)Salir (1)Auto (2)Avión: 2  
¿Nuevo NumeroPieza?: 378  
¿Número de motor?: 4938

(0) Salir (1) Auto (2) Avión: 1  
¿Nuevo NumeroPieza?: 4499  
¿Año del modelo?: 94

(0) Salir (1) Auto (2) Avión: 1

```
¿Nuevo NumeroPieza?: 3000
¿Año del modelo? 93

(0) Salir (1) Auto (2) Avión: 0
```

```
Número de pieza: 378
Número de motor: 4938
```

```
Número de pieza: 2837
Año del modelo: 90
```

```
Número de pieza: 3000
Año del modelo: 93
```

```
Número de pieza: 4499
Año del modelo: 94
```

**ANÁLISIS** El listado R3.1 modifica el programa que se proporcionó en la semana 2, agregándole plantillas, procesamiento `ostream` y manejo de excepciones. La salida es idéntica.

En las líneas 36 a 40 se declaran varias clases de excepciones. En el manejo algo primitivo de excepciones proporcionado por este programa no se requieren datos ni métodos de estas excepciones; sirven como indicadores para las instrucciones `catch`, los cuales imprimen una simple advertencia y luego hacen que el programa termine. Un programa más robusto podría pasar estas excepciones por referencia y luego extraer el contexto o cualquier otra información de los objetos de las excepciones en un intento por recuperarse del problema.

En la línea 44 se declara la clase base abstracta `Pieza` exactamente como se declaró en la semana 2. El único cambio interesante aquí se encuentra en el miembro `operator<<()` que no es de la clase, el cual se declara en las líneas 72 a 77. Observe que éste no es miembro ni amigo de `Pieza`; simplemente toma una referencia a `Pieza` como uno de sus argumentos.

Tal vez querría hacer que `operator<<` tomara un objeto `PiezaAuto` y un `PiezaAeroPlano` esperando que se llamara al `operator<<` correcto, con base en el tipo de pieza que se pasara, ya sea de auto o de aeroplano. Sin embargo, como el programa pasa un apuntador a una pieza, y no un apuntador a una pieza de auto o de aeroplano, C++ tendría que llamar a la función correcta con base en el tipo real de uno de los argumentos para la función. Esto se conoce como contravarianza, y C++ no la soporta.

Existen sólo dos maneras de lograr el polimorfismo en C++: el polimorfismo de funciones y las funciones virtuales. El polimorfismo de funciones no funcionará aquí, ya que en todo caso se está relacionando la misma firma: la de tomar una referencia a una `Pieza`.

Las funciones virtuales no funcionarán aquí ya que `operator<<` no es una función miembro de `Pieza`. No puede hacer que `operator<<` sea una función miembro de `Pieza` debido a que debe invocar a

```
cout << laPieza
```

y eso significa que la llamada en realidad sería a `cout.operator<<(Pieza &)`, ¡y `cout` no tiene una versión de `operator<<` que tome una referencia a `Pieza`!

Para sobrepasar esta limitación, el programa de la semana 3 utiliza sólo un `operator<<`, que toma una referencia a una `Pieza`. Éste a su vez llama a `Mostrar()`, que es una función miembro virtual, y por consecuencia se llama a la versión adecuada.

En las líneas 139 a 153, la clase `Nodo` se define como una plantilla. Esta clase sirve para lo mismo que la clase `Nodo` del programa de repaso de la semana 2, pero esta versión de `Nodo` no está atada a un objeto `Pieza`. De hecho, puede ser el nodo para cualquier tipo de objeto.

Observe que si trata de obtener el objeto de `Nodo`, pero no hay objeto, esto se considera como una excepción, y la excepción se produce en la línea 185.

En las líneas 192 a 208 se define una plantilla de la clase genérica `Lista`. Esta clase `Lista` puede guardar nodos de cualquier objeto que tenga número de identificación único, y los mantiene ordenados en forma ascendente. Cada una de las funciones de la lista revisa si hay circunstancias excepcionales y produce las excepciones apropiadas, según se requiera.

En la línea 320, el programa controlador crea una lista de objetos `Pieza` donde se almacenarán los dos tipos distintos que existen: `PiezaAuto` y `PiezaAeroPlano`. Posteriormente, se imprimirán los valores de cada objeto de la lista usando el mecanismo de flujo estándar.

### Preguntas frecuentes

**FAQ:** En el comentario que está arriba de la línea 72 del listado R3.1, mencionó que C++ no soporta la contravarianza. ¿Qué es la contravarianza?

**Respuesta:** La contravarianza es la habilidad de asignar un apuntador de una clase base a un apuntador de una clase derivada.

Si C++ soportara la contravarianza, podríamos redefinir la función con base en el tipo real del objeto, en tiempo de ejecución. El listado R3.2 no compilará en C++, pero si C++ soportara la contravarianza, sí se compilaría. ¡Este archivo no compilará!



Advertencia: ¡No será posible compilar este listado!

### **LISTADO R3.2 Una demostración de la contravarianza**

---

```
#include<iostream.h>
class Animal
{
public:
    virtual void Hablar() { cout << "Animal habla\n"; }

class Perro : public Animal
{
public:
    void Hablar() { cout << "Perro habla\n"; }

class Gato : public Animal
{
public:
    void Hablar() { cout << "Gato habla\n"; }

void Hacerlo(Gato *);
void Hacerlo(Perro *);

int main()
{
    Animal * apA = new Perro;
    Hacerlo(apA);
    return 0;
}

void Hacerlo(Gato * c)
{
    cout << "¡Pasaron un gato!\n" << endl;
    c->Hablar();
}

void Hacerlo(Perro * d)
{
    cout << "¡Pasaron un Perro!\n" << endl;
    d->Hablar();
}
```

Lo que puede hacer, desde luego, es utilizar una función virtual, lo que resuelve parcialmente el problema. El listado R3.3 muestra cómo se hace esto.

### **LISTADO R3.3 Una muestra de funciones virtuales**

---

```
#include<iostream.h>

class Animal
{
public:
    virtual void Hablar() { cout << "Animal habla\n"; }

class Perro : public Animal
{
public:
    void Hablar() { cout << "Perro habla\n"; }

class Gato : public Animal
{
public:
    void Hablar() { cout << "Gato habla\n"; } .

void Hacerlo(Animal *);

int main()
{
    Animal * apA = new Perro;
    Hacerlo(apA);
    return 0;
}

void Hacerlo(Animal * c)
{
    cout << "Pasaron algún tipo de animal \n" << endl;
    c->Hablar();
}
```

---



# SEMANA 4

## De un vistazo

Ya terminó la tercera y última semana regular del aprendizaje de C++ para Linux. Para estos momentos debe estar familiarizado con los aspectos avanzados de la programación orientada a objetos.

## Objetivos

Por fin ha llegado a la semana adicional, que trata los temas más específicos y avanzados sobre el desarrollo de programas de C++ en Linux. En el día 22, “El entorno de programación de Linux”, aprenderá acerca de algunas de las herramientas de programación avanzadas en Linux, y en el día 23, “Programación shell”, aprenderá cómo utilizar los intérpretes de comandos en Linux y cómo crear secuencias sencillas de comandos de shell. El día 24, “Programación de sistemas”, se enfoca en la interfaz del sistema operativo y las funciones y objetos de las bibliotecas de C++ que puede utilizar. En el día 25, “Comunicación entre procesos”, aprenderá a utilizar las funciones del sistema que permiten la comunicación entre distintos procesos. Por último, en el día 26, “Programación de la GUI”, aprenderá cómo crear aplicaciones para aprovechar las herramientas de la GUI (interfaz gráfica de usuario) disponibles en Linux.

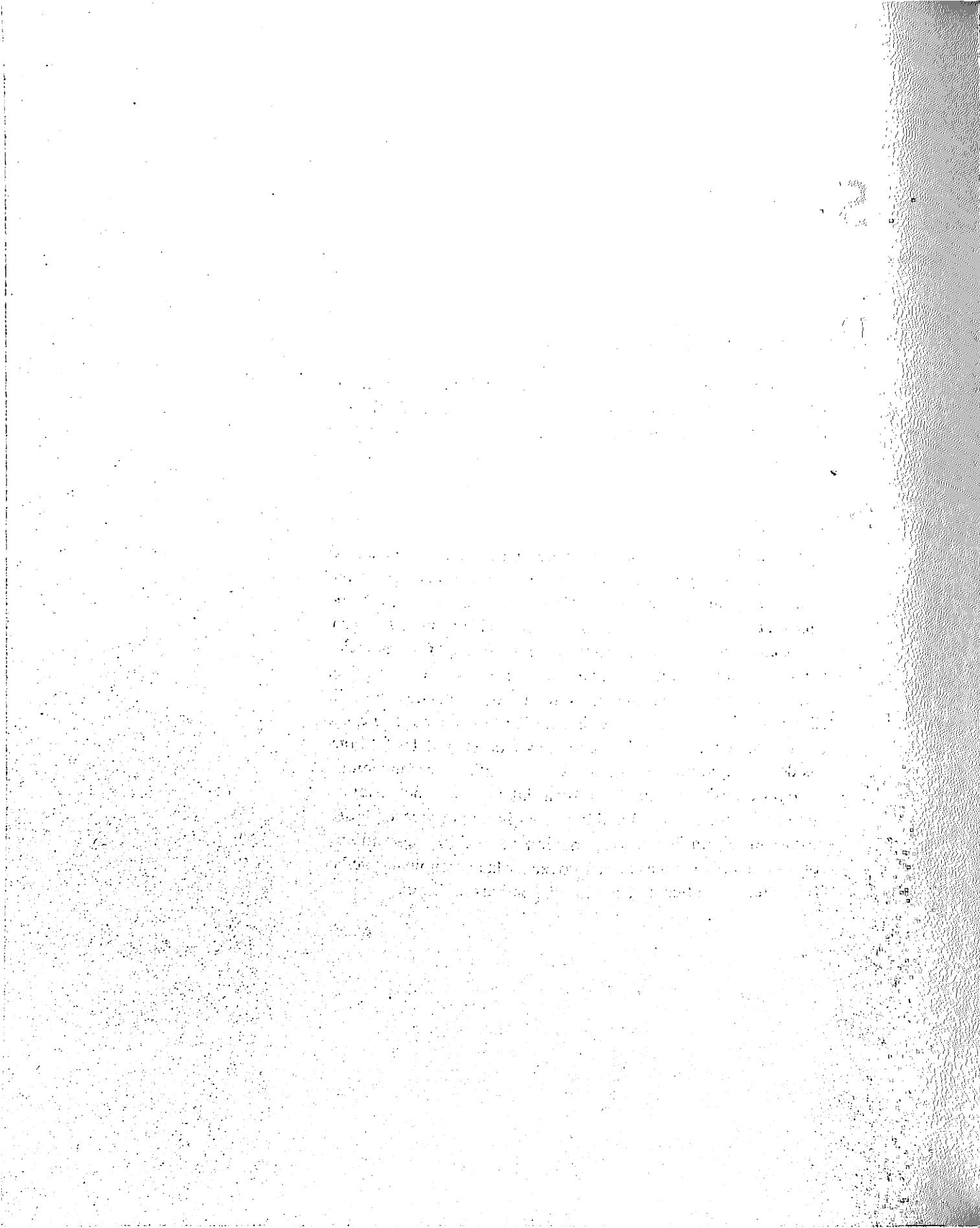
22

23

24

25

26



# SEMANA 4

DÍA 22

## El entorno de programación de Linux

Hasta estos días, ha estado aprendiendo a utilizar el lenguaje de programación C++. Para estos momentos ya conoce la sintaxis del lenguaje y la forma de diseñar y escribir programas orientados a objetos. La lección de hoy trata sobre el entorno para escribir programas que proporciona Linux. ¿Cuáles son las herramientas disponibles? ¿Cómo se crea, se compila y se depura un programa, y cómo se rastrea su historial a medida que se va modificando?

Esta lección trata acerca de los siguientes conceptos básicos:

- Editores
- Compiladores
- Creación de archivos ejecutables con make
- Bibliotecas y enlace
- Depuración con gdb
- Control de versiones con RCS
- Documentación

## Filosofía e historia

En sus primeros días, Linux era popular casi exclusivamente entre los desarrolladores (en especial, personas que querían estudiar o escribir piezas del kernel, o personas que querían crear herramientas y otras utilerías). Si va a desarrollar software, lo primero que necesita son herramientas de desarrollo. Una vez que tiene un editor, necesita un compilador. Como resultado, las primeras cosas que se llevaron a Linux fueron compiladores, ensambladores y enlazadores.

Un compilador es algo difícil de escribir, y usted realmente no querría crear uno desde cero si no tuviera que hacerlo. Por lo tanto, era normal que los desarrolladores de Linux buscaran un compilador gratuito, que tuviera el código fuente disponible para poder portarlo. La elección obvia (tal vez la única elección) en ese entonces era el compilador GNU de C. Las herramientas GNU se adherían a una filosofía similar de código abierto, y estaban ampliamente disponibles, además de contar con una alta calidad comprobada.

A diferencia de muchos sistemas operativos comerciales de la actualidad, las herramientas disponibles para Linux no están limitadas a las que proporciona el fabricante o a las que se pueden comprar. Linux es software de código abierto, y las herramientas de código abierto disponibles en Internet funcionan generalmente con Linux.

Si hay una herramienta de código abierto que no haya sido incluida en su distribución, lo más probable es que ya pueda descargar de Internet una versión creada para Linux. Por otro lado, casi todas las distribuciones de Linux vienen con un conjunto muy completo de herramientas GNU para desarrollo de programas.

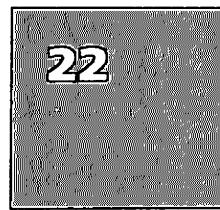
## POSIX

En los 80, el IEEE (Instituto de Ingenieros Eléctricos y Electrónicos) empezó a desarrollar un entorno de programación estándar para promover la portabilidad de aplicaciones entre distintos entornos UNIX. El nombre que se dio a este esfuerzo y al estándar resultante fue “Interfaz Portable de Sistema Operativo”, y se conoce comúnmente como POSIX. Este estándar no regula la forma en que debe comportarse un sistema operativo, pero si define la API (interfaz de programación de aplicaciones) que el sistema operativo debe proporcionar para el escritor de programas de aplicaciones. Esto puede parecer un poco confuso. Basta decir que Linux se apega al estándar POSIX. En otras palabras, proporciona un conjunto estándar de llamadas al sistema y servicios como los definidos por el estándar POSIX.

## El sistema X Windows

Si ha utilizado computadoras Macintosh o computadoras que ejecuten cualquier versión de Microsoft Windows, entonces está familiarizado con lo que se conoce como GUI (interfaz gráfica de usuario). En estos sistemas, la GUI es una parte integral del sistema operativo. Esto no es así con Linux. En Linux, la GUI se encuentra en el nivel superior

del software de sistema de ventanas, que es, en esencia, un accesorio para el sistema operativo. Con este método hay muchas ventajas en cuanto a rendimiento, confiabilidad y flexibilidad. El nombre “sistema X Windows” se aplica libremente a todos los componentes de este software de ventanas, incluyendo con frecuencia a la propia GUI. El desarrollo de “X”, como se le conoce comúnmente, empezó en el MIT a mediados de los 80. El sistema X Windows proporcionado con Linux viene del proyecto XFree86.



## Uso de los editores de Linux

Si va a escribir código, evidentemente necesita alguna forma de introducirlo a un archivo. Uno de los primeros resultados del proyecto GNU fue el editor emacs, y los editores estuvieron entre los primeros programas que se portaron a Linux.

### **ed, ex, vi y las variantes de vi**

ed y ex son de los primeros editores que se utilizan desde la línea de comandos, disponibles en los sistemas UNIX. Por lo general, ed se utiliza como un filtro para modificar el texto que pasa a través de él. ex es un editor de texto en línea útil pero primitivo. Uno de los primeros editores de UNIX basados en pantallas es vi, el cual es básicamente una interfaz basada en pantallas escrita con base en ex.

En la mayoría de los sistemas Linux está disponible la reimplementación de código abierto de vi, llamada vim. Muchas personas consideran que vi (y también vim) es enigmático y difícil de utilizar. Sin embargo, tiene muchas ventajas con respecto a otros editores:

- Es muy poderoso.
- Está disponible prácticamente en todos los sistemas UNIX y Linux.
- Necesita menos recursos del sistema que emacs, y por consecuencia se ejecuta aunque el sistema no esté funcionando completamente.
- No es tan personalizable como emacs, lo que significa que todas las implementaciones se comportan casi de la misma forma.

Puede utilizar otro editor para el uso diario, pero es conveniente que domine los fundamentos de vi. Los sistemas UNIX siempre vienen con vi, y como vim es la implementación de vi que se proporciona con Linux, por lo general está vinculado a vi para que pueda invocarlo con el nombre vi. Para el resto de esta lección, utilizaremos los términos vi y vim como si fueran uno solo. En realidad nos estamos refiriendo a vim.

### **Inicio de vi**

Antes de ejecutar cualquiera de los editores de pantalla completa (incluyendo a vi), debe tener configurada la variable de entorno TERM para que indique el tipo de su terminal. Esta variable se configura normalmente a xterm cuando se inicia una ventana en el entorno gráfico. Si no está ejecutando el entorno gráfico, la terminal predeterminada es Linux.

Puede ver la página del manual en línea para vi con el comando `man vi`. El comando `man vim` le proporciona exactamente la misma página.

También hay bastante ayuda disponible dentro del editor.

Inicie el editor con el comando `vi nombrearchivo`

`vim` cuenta con un modo gráfico, con el cual no cuenta `vi`. Incluye soporte para ratón y menús desplegables. Puede invocar este modo con `gvim nombrearchivo` o con `vi -g nombrearchivo`. Observe que para que el modo gráfico funcione, debió compilar el editor con esa opción específicamente habilitada. Su versión tal vez no haya sido compilada de esta manera. La figura 22.1 muestra un archivo editado en `vi`.

**FIGURA 22.1**

Un archivo editado en `vi`.

```
# inittab      This file describes how the INIT process should set up
#                                     the system in a certain run-level.
#
# Author:      Miquel van Smoorenburg, <miquels@drinkel.nl.mugnet.org>
#               Modified for RHS Linux by Marc Ewing and Donnie Barnes
#
# Default runlevel. The runlevels used by RHS are:
#   0 - halt (Do NOT set initdefault to this)
#   1 - Single user mode
#   2 - Multiuser, without NFS (The same as 3, if you do not have networking)
#   3 - Full multiuser mode
#   4 - unused
#   5 - X11
#   6 - reboot (Do NOT set initdefault to this)
#
id:5:initdefault:

# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit'

10:0:wait:/etc/rc.d/rc 0
11:1:wait:/etc/rc.d/rc 1
12:2:wait:/etc/rc.d/rc 2 .
13:3:wait:/etc/rc.d/rc 3
"inittab" [readonly] line 1 of 57 --1%-- col 1
```

## Conceptos de `vi`

`vi` es un “editor de modo” basado en texto, y viene tres modos: command, insert y ex. El modo predeterminado es el modo de comandos. Esto significa que lo que escribe son comandos para el editor en lugar de entrada para el archivo en el que está trabajando.

La mayoría de los comandos de `vi` es de una sola letra, algunos tienen un modificador de alcance. La mayoría puede estar precedida de un número, que es el “factor de repetición” (el cual ocasiona que el comando subsecuente se repita ese número de veces).

En modo insertar, las pulsaciones de teclas se capturan y se introducen en el archivo que se está editando. En este modo, la palabra *INSERT* se despliega en la parte inferior de la pantalla.

El `vi` original era en realidad una interfaz basada en pantalla para el editor de línea `ex`, y `vim` emula este comportamiento. El tercer modo disponible en `vim` es el modo ex. En este modo aparece un indicador “`:`” en la parte inferior de la pantalla. Puede escribir cualquier comando ex en este indicador. Los comandos ex son útiles para alternar entre archivos sin salir del

editor, y ofrecen la completación del nombre de archivo para facilitar esto (es decir, usted escribe las primeras letras del nombre de un archivo y después presiona la tecla "Entrar", y el nombre del archivo se completa automáticamente). Casi todas las otras tareas que puede realizar con un comando ex, también puede realizarlas desde el modo de comandos vi.

Para regresar al modo de comandos desde cualquier otro modo, oprima "Esc". Si ya se encuentra en modo de comandos, esta tecla no tiene efecto, así que si alguna vez no está seguro de en cuál modo se encuentra, sólo oprima "Esc" un par de veces para regresar al modo de comandos.

Para tener acceso a la ayuda en línea con vi, como se muestra en la figura 22.2, escriba :help desde el modo de comandos.

**FIGURA 22.2**

Ayuda de vi.

```

help.txt*      For Vim version 5.3. Last modification: 1998 Aug 23
              VIM - main help file

Move around: Use the cursor keys, or "h" to go left,          k
              "j" to go down, "l" to go right.           h   l
Close this window: Use ":q{Enter}".
Get out of Vim: Use ":qa!{Enter}" (careful, all changes are lost!).
Jump to a subject: Position the cursor on a tag between {bars} and hit CTRL-T.
With the mouse:  ":set mouse=a" to enable the mouse (in xterm or GUI).
Double-click the left mouse button on a tag between {bars}.
Jump back:      Type CTRL-T or CTRL-O.
Get specific help: It is possible to go directly to whatever you want help
on, by giving an argument to the ":help" command |:help|.
It is possible to further specify the context:
    WHAT          PREPEND     EXAMPLE
    Normal mode commands  (nothing)  :help x
    Visual mode commands  v_        :help v_u
    Insert mode commands  i_        :help i_{Esc}
    command-line commands  :        :help :quit

#
# inittab      This file describes how the INIT process should set up
#               the system in a certain run-level.
#
inittab [RO]
"help.txt" [readonly] 1185 lines, 55790 characters

```

## Uso de vi

Ahora pruebe una sesión de ejemplo con vi. Para esto, creará un programa sencillo en C++ llamado `hola.cxx`.

1. Escriba `vi hola.cxx` para empezar a editar el archivo nuevo.
2. Escriba `i` para entrar al modo insertar.
3. Escriba el siguiente texto:

```

#include <iostream.h>
int main(int argc, char * argv[])
{
    cout << "¡Hola, mundo!" << endl;
    return 0;
}

```

4. Oprima "Esc" para salir del modo insertar.

Ahora que se encuentra de nuevo en el modo de comandos, observe que las teclas h, j, k y l le permiten mover el cursor hacia la izquierda, hacia abajo, hacia arriba y hacia la derecha, respectivamente.

Puede volver a entrar al modo insertar en cualquier momento. La tecla "i" empieza la inserción en el lugar donde se encuentra el cursor, y la tecla "a" empieza la inserción después del cursor. Otras teclas que puede utilizar son: "A" para insertar al final de la línea actual; "o" para insertar en una nueva línea después de la línea actual; y "O" para insertar una nueva línea en la posición actual. Observe que cuando se encuentra en modo insertar, se despliega la palabra *INSERT* en la parte inferior de la pantalla. Para salir del modo insertar, oprima "Esc".

5. Escriba ZZ para escribir el archivo y salir (también puede utilizar :wq).

## emacs de GNU

Las guerras religiosas abundan. Muchos usuarios de Linux juran que vi es demasiado antiguo y que el único editor que vale la pena usar es emacs. Usted mismo tiene que decidir cuál editor es más fácil de utilizar. Ciertamente, emacs tiene más características, y hay varios libros importantes que tratan sobre este editor. emacs es demasiado complejo como para tratarlo detalladamente aquí, por lo que esta sección cubre sólo sus aspectos básicos.

### Cómo iniciar emacs de GNU

emacs se puede ejecutar dentro de una ventana de terminal o en su propia ventana en X Windows. Si está ejecutando X, el comando `emacs` & abrirá una nueva ventana con el editor. Para ejecutarlo dentro de la ventana actual, utilice el comando `emacs -nw`.

Si inicia emacs sin un nombre de archivo de destino, éste creará un búfer "scratch" por usted y le mostrará una breve pantalla de ayuda, como se muestra en la figura 22.3.

Si está ejecutando la versión X de emacs, puede utilizar el ratón para navegar por los menús desplegables que se encuentran en la parte superior de la pantalla. A medida que vaya adquiriendo más experiencia, descubrirá que hay teclas de método abreviado para todos estos comandos desplegables.

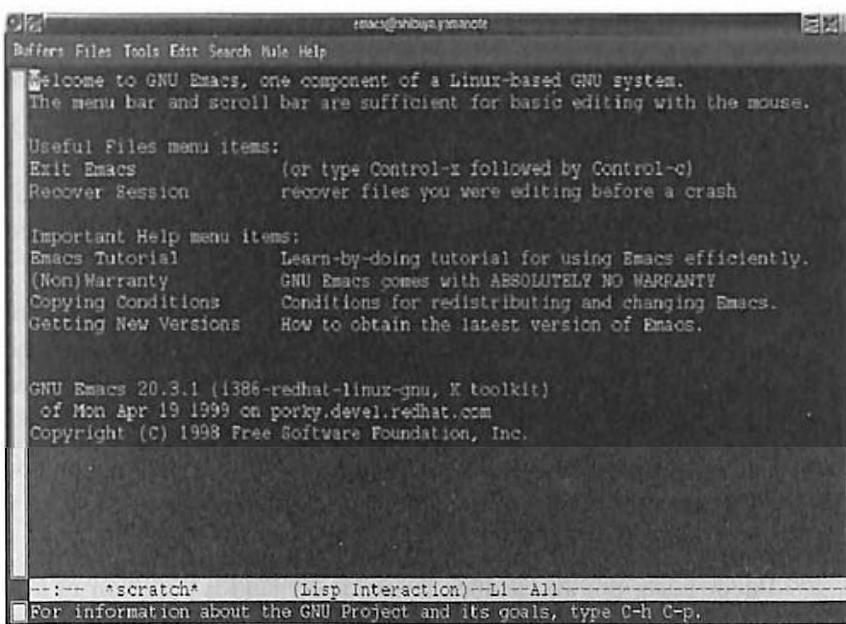
La tabla 22.1 muestra las notaciones convencionales de emacs.

### Nota

En la documentación de emacs, es común referirse a la secuencias de teclas en notación abreviada. La tabla 22.1 describe la notación convencional de emacs. La clave Meta se puede asignar de manera distinta en varias computadoras. En algunas es una tecla etiquetada como "Meta". Si su computadora no tiene esa tecla (la mayoría de las PCs típicas no la tienen; en su lugar, tienen la tecla "Alt"), entonces en vez de esta tecla oprima y libere la tecla "Esc". Por ejemplo, si la documentación de emacs dice "M-v", entonces debe oprimir y liberar "Esc", y luego oprimir "v".

**FIGURA 22.3**

Ventana de bienvenida de emacs de GNU.

**TABLA 22.1** Notación convencional de emacs

| Comando    | notación de emacs |
|------------|-------------------|
| Entrar     | RET               |
| Retroceso  | DEL               |
| Escape     | ESC               |
| Control    | C-                |
| Meta (Alt) | M-                |
| Espacio    | SPC               |
| Tabulador  | TAB               |

Para ir directamente al tutorial en línea, oprima *C-h t* (oprima *Control* y *h* al mismo tiempo, luego libere ambas teclas y oprima *t*).

El sistema *info* (descrito más adelante) proporciona ayuda en línea para todo el software de GNU. Para ver la información, oprima *C-h i*.

Hay otros tipos de ayuda disponibles en emacs. Puede oprimir *C-h ?* para averiguar qué tipo de ayuda existe. Por ejemplo, si desea ayuda acerca de una combinación específica de teclas, siga estos pasos:

1. *C-h* (muestra las opciones de ayuda)
2. *c* (selecciona la opción “describe key briefly” [breve descripción de teclas])
3. *C-x C-c* (muestra las acciones que realiza este comando)

Esto revelará que la secuencia *C-x C-c* es una abreviación del comando “*save-buffers-Kill-emacs*”.

## Conceptos sobre emacs de GNU

A diferencia de vi, emacs se conoce como un editor “sin modos”. Escribir en la ventana de emacs ocasiona que se escriban caracteres en el búfer que se está editando. Los comandos de emacs siempre se deben señalar (mediante “caracteres de escape”) con alguna secuencia de teclas de control.

emacs tiene muchos comandos, la mayoría de los cuales tienen nombres descriptivos muy largos. Muchos de estos comandos están “vinculados” con secuencias más cortas de teclas de control. Utilice la secuencia *C-h c* para descubrir los vínculos para una tecla o secuencia de control.

Una de las cosas buenas acerca de emacs es la gran cantidad de comandos disponibles. Una de las cosas difíciles sobre este editor es aprender las secuencias de control a las que están vinculadas estas teclas.

Puede pasar por alto la abreviación de comandos y utilizar el nombre largo del comando. Oprima “Esc” y escriba *x* para desplegar el indicador *M-x* en la parte inferior de la ventana. Ahora empiece a teclear un comando, por ejemplo, *save-buffer*. Escriba *sa* y oprima la barra espaciadora o el tabulador, y entonces emacs completará la palabra *save-*. Presione otra vez la barra espaciadora y emacs le mostrará los posibles comandos completos. Después escriba la letra “*b*” y oprima de nuevo la barra espaciadora, y se completará la palabra *buffers*. Ahora oprima “Entrar” y se ejecutará el comando.

En cualquier momento puede oprimir *C-g* para abortar cualquier comando parcial que haya empezado a escribir.

## Uso de emacs

Utilice los siguientes pasos para modificar el archivo que escribió anteriormente en vi.

1. Escriba *emacs* para iniciar una sesión de edición sin archivo.
2. Oprima *C-x C-f* para obtener un indicador que le permita elegir un archivo.
3. Escriba *hola.cxx* para abrir el archivo que creó anteriormente. Al estar escribiendo el nombre, en cualquier momento puede presionar la barra espaciadora o el tabulador. Si emacs puede completar sin ambigüedad el nombre de archivo, lo hará. En caso contrario, vuelva a presionar la barra espaciadora o el tabulador, y entonces emacs le mostrará una lista de los archivos existentes cuyos nombres inicien con la subcadena que escribió.
4. Oprima “Entrar” y se desplegará el archivo *hola.cxx*.
5. Utilice las teclas de dirección para desplazarse por el archivo. También puede utilizar *C-n* para moverse una línea hacia abajo, *C-p* para moverse hacia arriba, *C-f* para moverse un carácter hacia adelante, y *C-b* para moverse hacia atrás. Muévase a la “*m*” de la palabra “mundo”.

6. Escriba de nuevo y observe que el texto se inserta en donde se encuentra el cursor.
7. Oprima *C-x C-s* para escribir en el archivo.
8. Oprima *C-x C-c* para salir del editor.

Muchas personas sólo utilizan emacs. Puede utilizarlo no sólo para editar archivos, sino también para ejecutar comandos y para compilar, depurar y ejecutar programas. Si tiene errores en tiempo de compilación, emacs puede saltar automáticamente hasta la línea que contenga el error. Sin duda, vale la pena invertir tiempo para investigar más sobre emacs.

## ctags y etags

Al escribir un programa grande en C++, cabe la posibilidad de que necesite dividir el código fuente en varios archivos. Cada archivo definirá los métodos para una o varias clases.

Después, al depurar el archivo, puede ser difícil navegar por todos esos archivos fuente. Tal vez esté editando el archivo A y haya un método invocado que esté definido en alguna otra parte, tal vez en el archivo B. Los programas ctags y etags crean archivos índice o "tag", que vi y emacs pueden utilizar para ayudarlo a navegar por sus archivos fuente.

ctags es el programa más antiguo, y genera marcas para vi de manera predeterminada. Puede indicarle que genere marcas para emacs. etags genera marcas para emacs de manera predeterminada, pero también puede indicarle que genere marcas para vi.

### Ejemplo de ctags con vi

Escriba el siguiente código en un archivo llamado `holaPrincipal.cxx`:

```
#include <iostream.h>
void Saludar(int i);
int main(int argc, char * argv[])
{
    for(int i = 0; i < 5; i++)
    {
        Saludar(i);
        cout << endl;
    }
}
```

Ahora escriba el siguiente código en un archivo llamado `Saludar.cxx`:

```
#include <iostream.h>
void Saludar(int i)
{
    cout << "[" << i << "] ¡Hola, mundo!";
    return 0;
}
```

Escriba el comando `ctags *.cxx`

Ha creado el nuevo archivo `tags`.

Inicie una sesión de edición con el comando `vi holaPrincipal.cxx`. Ahora, usando las teclas `h`, `j`, `k` y `l`, ponga el cursor en la palabra `Saludar`. Oprima `C-J`, y verá que el editor abre automáticamente el archivo que contiene la definición de esa función (`Saludar.hxx`) y coloca el cursor al inicio de la función.

Hay una funcionalidad similar en emacs. Utilice la ayuda en línea para ver cómo se utiliza.

## Lenguajes

Linux proporciona todos los lenguajes disponibles en sistemas UNIX tradicionales, y algo más. Muchos lenguajes no tradicionales están disponibles en Internet. La mayoría de las distribuciones vienen con C y C++, y a menudo con una implementación de Java. Los lenguajes de secuencias de comandos como perl, sed y awk también son parte de la mayoría de las distribuciones.

### gcc y g++

El compilador C de GNU se llama `gcc` y puede compilar C, C++ y Objective-C. El compilador de C se apega al estándar ANSI, por lo que debe ser sencillo portar de ANSI a Linux casi cualquier programa de C. Si está familiarizado con el compilador de C de cualquier otro sistema UNIX, descubrirá que `gcc` es bastante similar. Debido a que `gcc` es gratuito y de alta calidad, muchos sitios comerciales lo utilizan como su compilador de C preferido.

### Cómo compilar con gcc

El compilador GNU se invoca con el comando `gcc`. De manera predeterminada, este comando preprocesará, compilará y enlazará un programa de C. Existen muchas opciones para `gcc`, y entre ellas existen controles que le permiten ejecutar cualquier fase específica de la secuencia preproceso/compilación/enlace.

El siguiente ejemplo sencillo tira un dado *n* veces, y luego imprime el número de veces que sale cada una de sus caras.

El listado 22.1 muestra el programa principal llamado para el juego dado.

**ENTRADA****LISTADO 22.1** Programa principal para el juego "dado"

```
1: // Listado 22.1 Programa principal del juego Dado
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <string.h>
6:
7: int tirarDado(void);
8:
9: int main(int argc, char * argv[])
10: {
```

```

11: int i;
12: int iIter;
13: int Dado[ 6 ];
14:
15: if (argc < 2)
16: {
17:     printf("Uso: %s\n", argv[ 0 ]);
18:     return 1;
19: }
20: iIter = atoi(argv[ 1 ]);
21: memset(Dado, 0, sizeof(Dado));
22: for(i = 0; i < iIter; i++)
23: {
24:     Dado[ tirarDado() - 1 ]++;
25: }
26: printf("%d tiradas\n", iIter);
27: printf("\tCara\tTiradas\n");
28: for(i = 0; i < 6; i++)
29: {
30:     printf("\t%d :\t%d\n", i + 1, Dado[ i ]);
31: }
32: }
```

La función `tirarDado()` se implementa en el archivo `lst22-02.cxx`:

### ENTRADA LISTADO 22.2 La función `tirarDado()`

```

1: // Listado 22.2 Implementación de la función tirarDado()
2:
3: #include <stdlib.h>
4:
5: int tirarDado(void)
6: {
7:     return((rand() % 6) + 1);
8: }
```

Podría utilizar un solo comando para crear este programa:

```
gcc -o dado lst22-01.cxx lst22-02.cxx
```

Puede ver que `-o` indica el nombre del archivo de salida. `gcc` es lo suficientemente inteligente para ver que los archivos que terminan con `.c` son archivos fuente de C, y los compila como tales. Si no especifica un nombre de archivo de salida, el programa de salida predeterminado se llamará `a.out`.

Estamos tratando de demostrar un concepto ligeramente más complejo, así que hagamos esto otra vez, sólo que esta vez compilaremos los módulos por separado:

```
gcc -c lst22-01.cxx
gcc -c lst22-02.cxx
gcc -o dado lst22-01.o lst22-02.o
```

## Cómo compilar con g++

El comando `gcc` es en realidad una “interfaz” del compilador. Al analizar los archivos que se le proporcionan, sabe si se requiere un enlace o una compilación. También sabe si el archivo fuente es de C o de C++. Si está compilando C++, puede invocar en forma alternativa el compilador `g++` directamente con el comando `g++ nombrearchivo`.

Aunque `gcc` puede compilar programas de C++, no hace automáticamente todos los enlaces requeridos con las bibliotecas de clases. Necesita usar `g++` para esto. Como resultado, por lo general es más sencillo compilar y enlazar programas de C++ con `g++`.

Por ejemplo, para crear el programa `hola` que creó anteriormente con `vi`, utilizaría los siguientes comandos de `g++`:

```
g++ -c holaPrincipal.cxx
g++ -c Saludar.cxx
g++ -o hola holaPrincipal.o Saludar.o
```

Al igual que con `gcc`, puede lograr lo mismo con la simple invocación:

```
g++ -o hola holaPrincipal.cxx Saludar.cxx
```

## Lenguajes de secuencias de comandos (perl, sed, awk)

Como entorno integral similar a UNIX, Linux también proporciona los lenguajes típicos de secuencias de comandos. `perl`, `sed` y `awk` son tres lenguajes importantes que se proporcionan de manera predeterminada con la mayoría de las distribuciones de Linux. Hay otros lenguajes de secuencias de comandos (`Tcl/Tk`, `Python`, `Expect`, entre otros) disponibles para Linux, pero están más allá del alcance de esta lección.

## ELF

Cuando se compila un programa, se genera un archivo objeto, y cuando se enlaza el programa, se crea un archivo binario ejecutable. El enlazador debe entender el formato de los archivos objeto, y como el sistema operativo debe cargar y ejecutar el programa ejecutable, también debe entender ese formato.

Ya vio que el archivo ejecutable predeterminado se llama `a.out`. Hasta hace poco, el formato de los archivos objeto y de los archivos ejecutables se conocía como formato `a.out`. Este formato es bastante antiguo y tiene varios defectos. El formato más moderno utilizado por la mayoría de los sistemas UNIX y Linux se conoce como ELF (formato ejecutable y de enlace). ELF es mucho más versátil que `a.out`, y se presta muy bien para crear bibliotecas compartidas.

Puede saber cuál es el formato de un archivo por medio del comando `file`:

```
file dado /usr/bin/archivo 1st22-02.o
```

**SALIDA**

```
dado: ELF 32-bit LSB executable, Intel 80386, ...
/usr/bin/archivo: ELF 32-bit LSB executable, Intel 80386, ...
1st22-02.o: ELF 32-bit LSB relocatable, Intel 80386, ...
```

## Bibliotecas compartidas

A menudo, varios programas necesitan hacer las mismas cosas, como E/S por ejemplo. Hace mucho tiempo se desarrolló el concepto de biblioteca para adaptar esto. Las funciones comunes se pueden colocar en un archivo, y luego, cada vez que se crea un programa, éste extrae de la biblioteca las funciones que necesita.

En su momento, esto fue un avance, pero tenía varias desventajas. Los ejecutables se hacen más grandes ya que cada uno de ellos incrusta código copiado de las bibliotecas. Si se encuentra un error en la biblioteca o se agrega una característica, el ejecutable no hace uso de esto a menos que se vuelva a crear.

La solución para este problema es la biblioteca compartida (o dinámica). El mecanismo de funcionamiento de las bibliotecas compartidas está más allá del alcance de esta lección. Sólo veremos cómo crearlas y utilizarlas.

Regresemos al programa para tirar dados que vimos antes. Este programa tiene dos archivos fuente. Compilamos ambos archivos y los enlazamos para crear un ejecutable.

Parece que hay un mercado para los programas para tirar dados, y creemos que podemos usar la función `tirarDado()` en una variedad de productos que creará nuestra nueva compañía. Tiene sentido colocar la función en una biblioteca para que todos nuestros programas puedan utilizarla.

Primero necesitamos crear la biblioteca compartida. Compile el módulo con el siguiente comando:

```
gcc -fPIC -c lst22-02.cxx
```

Ahora conviértalo en una biblioteca compartida llamada `bibtirar.so.1.0`:

```
gcc -shared -Wl,-soname,libtirar.so.1 -o libtirar.so.1.0 lst22-02.o
```

Por último, cree un enlace para `libtirar.so`, para que el programa en ejecución no necesite mantener un registro de la información de versión en el nombre de la biblioteca compartida:

```
ln -s libtirar.so.1.0 libtirar.so  
ln -s libtirar.so.1 libtirar.so
```

Ahora que tenemos la biblioteca, debemos volver a crear el programa principal para que se enlace con esa biblioteca en tiempo de ejecución, en lugar de incorporar el código dentro del ejecutable:

```
gcc -o dado lst22-01.cxx -L. -ltirar
```

La opción `-L.` le indica al compilador que busque bibliotecas en el directorio actual, y la opción `-ltirar` le indica que busque una biblioteca llamada `libtirar.so`.

Al ejecutar el programa, el sistema operativo cargará dinámicamente la biblioteca correcta, pero tiene que saber en dónde buscarla. Si la biblioteca no se encuentra en un lugar estándar (`/usr/lib`), puede asignar una variable de entorno para que le indique en dónde localizar bibliotecas adicionales:

```
setenv LD_LIBRARY_PATH /home/myname/mylibs (si utiliza csh o tcsh)
export LD_LIBRARY_PATH=/home/myname/mylibs (si utiliza sh o bash)
```

Por último, para ver qué bibliotecas usa un programa, utilice el comando ldd:

```
ldd dado
```

**SALIDA**

```
libtirar.so.1 => /home/Dado/libtirar.so.1 (0x40014000)
libc.so.6 => /lib/libc.so.6 (0x4001b000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

## Construcción o creación

Con seguridad, usted no querrá escribir todos estos comandos de gcc cuando cree un programa. Con los programas pequeños como los que se muestran en esta lección, esto es sólo un poco molesto. Con programas más grandes en los que se involucren varios archivos y tal vez varias bibliotecas, esto se vuelve casi imposible.

Linux viene con la utilería make de GNU. make lee de un archivo conocido como make toda la información que necesita para crear su programa. Esta utilería es tan importante y popular que se ha especificado como estándar de POSIX. La versión GNU de make se apega al estándar POSIX.

### make

make de GNU busca automáticamente un archivo make llamado **GNUmakefile**. Si no lo encuentra, busca **makefile**, y si tampoco lo encuentra busca **Makefile**. Éstos son los nombres predeterminados. Puede nombrar a su archivo make como usted quiera e indicar explícitamente a make que lo utilice. Un archivo make es un archivo de texto ordinario con una sintaxis muy específica que la utilería make puede entender.

make tiene una gran variedad de reglas integradas. Por ejemplo, sabe que los archivos que terminan con .c son archivos fuente de C, y sabe cómo compilarlos para convertirlos en archivos objeto (.o). Usted puede redefinir cualquiera de estas reglas si gusta. En el caso más simple, todo lo que necesita especificar en su archivo make es el nombre que va a tener su archivo ejecutable, así como los archivos .o que se necesitan para crearlo.

He aquí un archivo make sencillo que crea el programa para tirar dados:

```
dado: 1st22-01.o 1st22-02.o
$(CC) -o $@ 1st22-01.o 1st22-02.o
```

Los archivos que se incluyen en el CD-ROM tienen extensión .cxx, por lo que podría recibir el siguiente mensaje de error del comando make:

```
make: ***No hay ninguna regla para construir el objetivo `1st22-01.o' necesario para `dado'. Alto.
```

Para resolver este inconveniente, puede hacer dos cosas. La primera es cambiar la extensión de los archivos involucrados de .cxx a .c; esta solución no es recomendable, porque se buscará código de C y no de C++, lo cual puede generar más errores de compilación.

La segunda opción es agregar las líneas para crear los objetivos `lst22-01.o` y `lst22-02.o` como se muestra a continuación.

```
dado: lst22-01.o lst22-02.o
      $(CC) -o $@ lst22-01.o lst22-02.o

lst22-01.o: lst22-01.cxx
      $(CC) -c lst22-01.cxx

lst22-02.o: lst22-02.cxx
      $(CC) -c lst22-02.cxx
```

Obviamente, si ya creó los archivos `.o`, no recibirá ningún mensaje de error.

Ahora puede crear el programa con un solo comando:

```
make
cc      -c lst22-01.cxx
cc      -c lst22-02.cxx
cc -o dado lst22-01.o lst22-02.o
```

Nota: la línea del archivo `make` que empieza con “`dado`” se conoce como *destino*. Define las “dependencias” del programa. La siguiente línea es la regla de construcción. `make` requiere que el primer carácter de esa línea sea un tabulador, y no espacios. Si hay espacios en lugar de un tabulador, el error generado por `make` es el siguiente:

```
make
makefile:2: *** missing separator. Stop.
```

He aquí un archivo `make` un poco más completo:

```
# Makefile para crear el programa para tirar dados
CFLAGS = -O

OBJS    = lst22-01.o lst22-02.o

all:    dado

dado:   $(OBJS)
        $(CC) $(CFLAGS) -o $@ $(OBJS)

lst22-01.o: lst22-01.cxx
        $(CC) -c lst22-01.cxx

lst22-02.o: lst22-02.cxx
        $(CC) -c lst22-02.cxx

clean:
        - $(RM) dado *.o
```

Este archivo `make` define una regla llamada `clean`, que se utiliza para eliminar todos los archivos previamente compilados (incluso el ejecutable) y empezar con un directorio limpio. Esta regla también se utiliza para eliminar los archivos objeto temporales después de instalar el programa ejecutable y las bibliotecas (si se han definido).

El listado 22.3 muestra cómo se vería el archivo make si queremos usar bibliotecas compartidas.

**ENTRADA LISTADO 22.3 Archivo make con bibliotecas compartidas**

```

1: # Archivo make para crear el programa para tirar dados
2: # usando bibliotecas compartidas
3: CFLAGS = -O
4: OBJS = lst22-01.o
5: LIBS = libtirar.so
6:
7: all: dado
8:
9: dado: $(OBJS) $(LIBS)
10:    $(CC) $(CFLAGS) -o $@ $(OBJS) -L. -ltirar
11:
12: lst22-01.o: lst22-01.cxx
13:    $(CC) -c lst22-01.cxx
14:
15: lst22-02.o: lst22-02.cxx
16:    $(CC) -fPIC -c $<
17:
18: libtirar.so: lst22-02.o
19:    -$(RM) libtirar*
20:    $(CC) -shared -Wl,-soname,libtirar.so.1 \
21:        -o libtirar.so.1.o $<
22:        ln -s libtirar.so.1.o libtirar.so.1
23:        ln -s libtirar.so.1 libtirar.so
24:
25: clean:
26:    - $(RM) dado *.o libtirar*

```

### Opciones de línea de comandos de make

make tiene varias opciones útiles de línea de comandos.

Si quiere especificar un archivo make alternativo en lugar de uno de los predeterminados que se mencionaron antes, invoque a make de la siguiente manera:

`make -f nombrearchivo`

make es un programa muy sofisticado. Una de las cosas que hace es comprender las dependencias. Por ejemplo, sabe que los archivos .o se crean a partir de archivos .c. Su programa puede consistir en varios archivos fuente .c. Si cambia uno, no es necesario volver a compilarlos todos cada vez que vaya a crearlo. Sólo necesita volver a compilar el archivo fuente que haya cambiado. make comprende esto y compila sólo aquellos archivos que no estén actualizados. Algunas veces será necesario que vea primero qué es lo que make necesita para crear el programa. Puede hacer esto con el siguiente comando:

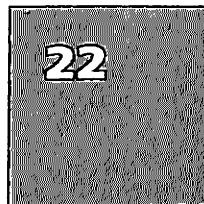
`make -n`

Esto le indica a make que analice el `makefile` y que reporte qué comandos emitirá para crear el programa. make no ejecutará ningún comando.

Nota: Cuando usted programa en C++, tiene una extensión de archivo distinta de .c (.cpp, .cxx, .cc, etcétera), así que deberá especificar las reglas para construir los archivos .o.

Otra característica útil en make es el uso de variables. Observe que definimos una variable llamada **CFLAGS**. make pasa automáticamente esta variable a gcc cuando compila su programa. Tal vez quiera cambiar el valor de esta variable una vez sin cambiar el archivo make. Puede especificar un nuevo valor en la línea de comandos que redefina el valor en el archivo:

```
make CFLAGS="-g -O"
```



## Depuración

Todo buen entorno de desarrollo tiene que proporcionar algo de capacidad de depuración, y Linux incluye el depurador GNU llamado **gdb**. Éste es un excelente depurador de código fuente con una interfaz de línea de comandos. Su distribución también debe incluir a **xxgdb**, una versión del mismo depurador con una interfaz gráfica que se ejecuta en X Windows. Si no tiene un ejecutable para **xxgdb**, realmente vale la pena conseguirlo.

### **gdb**

**gdb** le permite analizar un programa paso a paso, establecer puntos de interrupción y examinar y modificar variables por su nombre. Puede utilizarlo tanto con programas de C como de C++.

Para preparar un programa para depurarlo, sólo necesita agregar la opción **-g** a **gcc** cuando cree el programa. Puede hacer esto en la línea de comandos de **gcc** (si está creando el programa de esta manera). Si está utilizando **make**, puede colocar esta opción en el archivo **make**. De manera alternativa, si está utilizando **make**, puede hacer esto como se describió anteriormente, redefiniendo el valor de **CFLAGS** en la línea de comandos de **make**:

```
make CFLAGS=-g
```

Cuando se haya creado su programa, inicie **gdb** con el comando

```
% gdb dado
```

#### SALIDA

```
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
  are welcome to change it and/or distribute copies of it under certain
  conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for
  details.
This GDB was configured as "i386-redhat-linux"...
```

**gdb** tiene muchos comandos disponibles. Para verlos, escriba **help**.

```
(gdb) help
```

**SALIDA**

List of classes of commands:

aliases — Aliases of other commands  
 breakpoints — Making program stop at certain points  
 data — Examining data  
 files — Specifying and examining files  
 internals — Maintenance commands  
 obscure — Obscure features  
 running — Running the program  
 stack — Examining the stack  
 status — Status inquiries  
 support — Support facilities  
 tracepoints — Tracing of program execution without stopping the program  
 user-defined — User-defined commands

Type "help" followed by a class name for a list of commands in that class.

Type "help" followed by command name for full documentation. Command name abbreviations are allowed if unambiguous.

El comando `help` le muestra por sí mismo las clases de comandos que proporciona `gdb`. Para encontrar más acerca de los comandos dentro de una clase particular, utilice otra vez `help`, seguido del nombre de la clase de comandos:

```
(gdb) help breakpoints
```

**SALIDA**

Making program stop at certain points.

List of commands:

awatch — Set a watchpoint for an expression  
 break — Set breakpoint at specified line or function  
 catch — Set catchpoints to catch events  
 clear — Clear breakpoint at specified line or function  
 commands — Set commands to be executed when a breakpoint is hit  
 condition — Specify breakpoint number N to break only if COND is true  
 delete — Delete some breakpoints or auto-display expressions  
 disable — Disable some breakpoints  
 enable — Enable some breakpoints  
 hbreak — Set a hardware assisted breakpoint  
 ignore — Set ignore-count of breakpoint number N to COUNT  
 rbreak — Set a breakpoint for all functions matching REGEXP  
 rwatch — Set a read watchpoint for an expression  
 tbreak — Set a temporary breakpoint  
 tcatch — Set temporary catchpoints to catch events  
 thbreak — Set a temporary hardware assisted breakpoint  
 tbreak — Set temporary breakpoint at procedure exit  
 watch — Set a watchpoint for an expression  
 xbreak — Set breakpoint at procedure exit

Type "help" followed by command name for full documentation. Command name abbreviations are allowed if unambiguous.

Por último, para salir de `gdb`, escriba `q`.

La tabla 22.2 muestra algunos de los comandos más útiles de `gdb`.

**TABLA 22.2** Comandos útiles de *gdb*

| Comando                 | Función                                                                                                                                                                                               |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| break [archivo:]funcion | Establecer un punto de interrupción en la entrada a la función del archivo llamado archivo.                                                                                                           |
| run [listarg]           | Iniciar el programa y pasarle la [listarg], en caso de haber argumentos.                                                                                                                              |
| bt                      | Desplegar la pila del programa.                                                                                                                                                                       |
| print expr              | Evaluuar la expresión e imprimir el resultado.                                                                                                                                                        |
| c                       | Continuar la ejecución desde el punto actual.                                                                                                                                                         |
| next                    | Ejecutar la siguiente línea del programa. Si la siguiente línea es una llamada a una función, entonces esto hará que se llame a la función y se detendrá en la siguiente línea después de la llamada. |
| step                    | Ejecutar la siguiente línea del programa, y entrar a la función si esa línea es una llamada a una función.                                                                                            |
| help [nombre]           | Mostrar la ayuda general, o ayuda sobre el tema especificado por [nombre], si se proporciona uno.                                                                                                     |
| q                       | Salir de gdb.                                                                                                                                                                                         |

## xxgdb

xxgdb es una interfaz gráfica creada con base en gdb. Asegúrese de que su variable de entorno DISPLAY esté definida, y escriba:

```
xxgdb nombre_programa
```

La figura 22.4 muestra la interfaz gráfica de xxgdb.

**FIGURA 22.4**

xxgdb .

 A screenshot of the xxgdb graphical interface. The main window shows C code for a dice rolling application. Below the code, a toolbar contains buttons for various gdb commands like run, cont, next, step, etc. At the bottom, a license notice for GDB is displayed.
 

```

#include <stdio.h>
int doRoll(void);
main( int argc, char * argv[] )
{
    int i;
    int filter;
    int Die6;
    if( argc < 2 ) {
        printf("Usage: %s n\n", argv[0]);
        exit(1);
    }
    filter = atoi(argv[1]);
    if( filter > 6 || filter < 1 ) {
        printf("Header: /home/hal/work/jnk/Dice/diceMain.c,v 1.1 1999/12/20 08:37:58 hal Exp $n");
        for( i = 0; i < filter; i++ ) {
            Die6=roll();
            -11+;
        }
    }
}

Ready for execution
[run] [cont] [next] [step] [finish] [break] [tbreak] [delete] [up] [down] [print] [print w] [display] [undisplay] [show display] [args]
[locals] [stack] [edit] [search] [internet] [file] [show breakpoints] [yours] [no] [quit]

GCCDB comes with ABSOLUTELY NO WARRANTY.
GNU gdb 4.17.0.11 with Linux support
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
  
```

Hay tres secciones en la pantalla de xxgdb. La sección superior muestra el código fuente para el módulo que se está ejecutando en ese momento. La sección que está en medio contiene botones que son métodos abreviados para varios comandos de xxgdb. La sección inferior es una ventana de comandos. En esta ventana se muestra la salida del programa y de gdb. Si el programa o gdb requieren de entrada por medio del teclado, mueva el ratón hacia esta ventana y escriba.

Puede desplegar el valor de una variable o expresión resaltándola con el ratón y luego oprimiendo el botón Display (Mostrar). En este caso se abrirá otra sección que mostrará la actualización continua del valor de la variable que haya seleccionado.

## Sesión de ejemplo de depuración con gdb

He aquí una sesión de ejemplo de la ejecución del programa dado en gdb:

% gdb dado

**SALIDA**

```
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for
details.
This GDB was configured as "i386-redhat-linux"...
(gdb) list 1,25
1      // Listado 22.1 Programa principal del juego Dado
2
3      #include <stdio.h>
4      #include <stdlib.h>
5      #include <string.h>
6
7      int tirarDado(void);
8
9      int main(int argc, char * argv[])
10     {
11         int i;
12         int iIter;
13         int Dado[ 6 ];
14
15         if (argc < 2)
16         {
17             printf("Uso: %s \n\n", argv[ 0 ]);
18             return 1;
19         }
20         iIter = atoi(argv[ 1 ]);
21         memset(Dado, 0, sizeof(Dado));
22         for(i = 0; i < iIter; i++)
23         {
24             Dado[ tirarDado() - 1 ]++;
25         }
```

```
(gdb) break16 20
Breakpoint 1 at 0x8048658: file lst22-01.cxx, line 20.
(gdb) run 5
Starting program: /root/office52/user/work/dia22/dado 5

Breakpoint 1, main (argc=2, argv=0xbffffa64) at lst22-01.cxx:20
20      iIter = atoi(argv[ 1 ]);
(gdb) print iIter
$1 = -1073743352
(gdb) next
21      memset(Dado, 0, sizeof(Dado));
(gdb) print iIter
$2 = 5
(gdb) next
22      for(i = 0; i < iIter; i++)
(gdb) next
24      Dado[ tirarDado() - 1 ]++;
(gdb) print tirarDado()
$3 = 2
(gdb) next
25      }
(gdb) next
24      Dado[ tirarDado() - 1 ]++;
(gdb) cont
Continuing.
5 tiradas
     Cara   Tiradas
     1 :    0
     2 :    2
     3 :    0
     4 :    1
     5 :    1
     6 :    1

Program exited normally.
(gdb)
```

## Control de versiones

Los programas nunca son tan simples como se piensa que serán al principio. Cualquier buen programa va más allá de su propósito original. Con el tiempo hay cambios, se agregan cosas, se solucionan errores y se hacen mejoras.

Los comentarios son una excelente manera de mantener información relacionada con los cambios, pero para cualquier trabajo serio se necesita alguna forma de control de versiones. Suponga que cambia el programa `lst22-02.cxx`, agregándole varias características. Un año después, su cliente más importante le llama y le dice que no quiere todas esas características; que quiere la versión original del año pasado, en la que se había arreglado un error.

Linux viene con *RCS (Sistema de Control de Revisiones)*. RCS es una colección de comandos que le permiten rastrear cambios realizados en archivos, recuperar cualquier versión anterior y comparar las versiones actuales con las más antiguas.

## RCS

Los principales comandos de la suite RCS se muestran en la tabla 22.3.

**TABLA 22.3** Comandos de RCS

| Comando | Descripción                                                                                 |
|---------|---------------------------------------------------------------------------------------------|
| ci      | Insertar en el depósito una nueva revisión de un archivo                                    |
| co      | Obtener la última versión de un archivo                                                     |
| ident   | Buscar identificadores de RCS en archivos                                                   |
| merge   | Crear una versión de un archivo que incorpore cambios de otras dos versiones de ese archivo |
| rcsdiff | Comparar dos versiones de un archivo                                                        |
| rlog    | Ver el historial de un archivo                                                              |

RCS mantiene en un depósito el historial de las revisiones de los archivos. Por lo general, ese depósito es un directorio llamado **RCS**, que se encuentra en su directorio actual.

En el siguiente ejemplo, iniciamos el historial de RCS de un archivo en nuestro proyecto de tirar dados:

```
% mkdir RCS
% ci Makefile
```

### SALIDA

```
RCS/Makefile,v  <-- Makefile
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> Makefile del programa para tirar dados
>> .
initial revision: 1.1
done
```

Ahora hemos registrado el **Makefile** en el depósito de RCS, y RCS ha creado un archivo en el directorio **RCS** llamado **Makefile,v**. A medida que modifiquemos el **Makefile** y verifiquemos las versiones más recientes, RCS llevará el registro de esos cambios en su copia **Makefile,v**.

### Nota

Después de registrar un archivo en RCS, verá que su archivo original ha desaparecido. No se asuste, no lo perdió. RCS ha rastreado sus cambios en su copia y ha eliminado su original. Aún puede revisar su archivo con el comando **co**.

Registre todos los archivos necesarios para crear el programa de los datos, con los comandos **ci 1st22-01.cxx** y **ci 1st22-02.cxx**.

Piense en RCS como si fuera una biblioteca que guarda sus archivos. Puede sacar copias de sólo lectura con el comando **co nombrearchivo**. Cuando quiera modificar un archivo, puede sacar una copia en la que se pueda escribir (bloqueada) con **co -l**. Puede sacar

cualquier cantidad de copias de sólo lectura (desbloqueadas) a la vez. Sólo puede sacar una copia bloqueada a la vez.

Hay varias palabras reservadas de identificación que puede colocar en su archivo y que son reconocidas por RCS. Estas palabras reservadas empiezan y terminan con \$. Yo podría modificar nuestro programa `lst22-01.cxx` como se muestra en el listado 22.4:

**ENTRADA** **LISTADO 22.4** Programa `lst22-01.cxx` modificado

```
1: // Listado 22.4 Muestra el uso de control de versiones
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <string.h>
6:
7: int tirarDado(void);
8:
9: main (int argc, char * argv[])
10: {
11: int i;
12: int iIter;
13: int Dado[ 6 ];
14:
15: if(argc < 2)
16: {
17: printf("Uso: %s\n", argv[0]);
18: return 1;
19: }
20: iIter = atoi(argv[1]);
21: memset(Dado, 0, sizeof(Dado));
22: printf("$Header$\n");
23: for(i = 0; i < iIter; i++)
24: {
25: Dado[ tirarDado() - 1 ]++;
26: }
27: printf("%d tiradas\n", iIter);
28: printf("\tNúmero\tTiradas\n");
29: for(i = 0; i < 6; i++)
30: {
31: printf("\t%d :\t%d\n", i + 1, Dado[i]);
32: }
33: }
```

Cuando saco el archivo desbloqueado (sólo lectura), RCS reemplaza la palabra reservada `$Header$` con información acerca del nombre y la versión del archivo. Cuando lo saco bloqueado, RCS no reemplaza el encabezado. Ahora saco una copia desbloqueada, creo el programa y lo ejecuto:

```
% dado 10
```

**SALIDA**

```
$Header: /tmp/RCS/1st22-03.cxx,v 1.1 2000/12/20 08:37:38 usr Exp $  
10 tiradas  
Número Tiradas  
1 : 1  
2 : 4  
3 : 0  
4 : 2  
5 : 2  
6 : 1
```

## Documentación

Hay bastante documentación disponible en Linux, y la mayor parte de ella se incluye con las distribuciones. Usted tiene acceso a `man`, `info` y a los HOWTOs y FAQs de Linux. Como la documentación puede ser extensa, muchas de las distribuciones dejan la instalación del conjunto completo como algo opcional.

### Páginas del manual

Al igual que cualquier sistema operativo similar a UNIX, Linux incluye el comando `man` y las páginas del manual para todos los comandos.

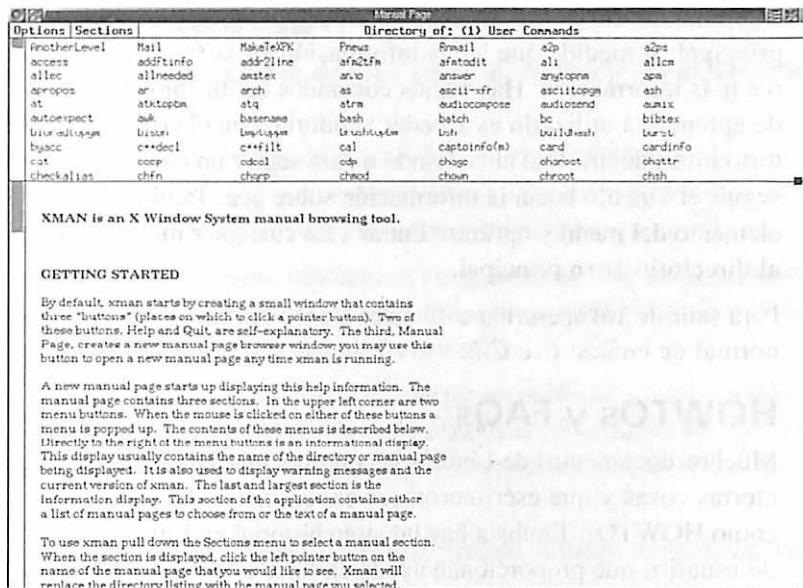
Algunas veces encontrará un comando con una página en más de una sección del manual. Por lo general, al utilizar `man`, éste despliega la primera entrada que encuentra. Si utiliza `man` con la opción `-a`, se mostrará la página de manual para el comando seleccionado en todas las secciones del manual que tengan una.

El comando `xman` proporciona una interfaz gráfica para el comando `man` básico. Escriba `xman -notopbox -bothshown`

Puede hacer clic con el botón izquierdo del ratón en el panel **Sections** (Secciones) para seleccionar otra sección del manual. Haga clic con el botón izquierdo del ratón en cualquier comando del panel superior para ver la página de manual correspondiente en el panel inferior. Haga clic con el botón izquierdo en el botón **Options** (Opciones) y seleccione la última entrada (`quit`) para salir de `xman`. La figura 22.5 muestra la interfaz gráfica para la página de manual.

FIGURA 22.5

xman



22

info

La documentación en línea en forma de páginas de manual ha estado en UNIX casi desde su comienzo. Una contribución más reciente es el sistema *info*. Navegar con *info* le parecerá natural si está familiarizado con emacs. De no ser así, puede ser un poco difícil de aprender.

Puede invocar a `info` desde la línea de comandos con el comando `info`. Si ya se encuentra en emacs, oprima “Esc” y luego escriba `-x info`. La figura 22.6 muestra a `info`.

**FIGURA 22.6**

info



El comando `info` está organizado en nodos, cada uno de los cuales representa un tema principal. A medida que lee la información en `info`, puede seguir vínculos a otros temas o a más información. Hay varios comandos de un solo carácter en `info`. La mejor manera de aprender a utilizarlo es acceder al tutorial con el comando `h` (desde la ventana de directorio `info` inicial). Use el comando `m` para seguir un elemento del menú. Escriba `m gcc` para seguir el vínculo hacia la información sobre `gcc`. También puede mover el cursor hacia un elemento del menú y oprimir “Entrar”. En cualquier momento puede escribir `d` para regresar al directorio `info` principal.

Para salir de `info`, escriba `q`. Si entró a `info` por medio de `emacs`, use la secuencia de salida normal de `emacs`: `C-c C-h`.

## HOWTOs y FAQs

Muchos documentos de Linux son contribuciones de usuarios que averiguaron cómo hacer ciertas cosas y que escribieron sus experiencias. Estas experiencias escritas se conocen como HOWTOs. También hay un largo historial en Linux de varios grupos y organizaciones de usuarios que proporcionan ayuda en línea a los usuarios. A menudo, las preguntas y sus respuestas correspondientes se recolectan en documentos conocidos como FAQs (Preguntas frecuentes). La mayoría de las distribuciones de Linux incluyen tanto los HOWTOs como las Preguntas frecuentes. Por lo general, se pueden encontrar en el directorio `/usr/doc/-HOWTO` y `/usr/doc/FAQ`. Su instalación puede colocarlos en cualquier otro lado.

Las Preguntas frecuentes y los HOWTOs más recientes están disponibles en línea en el sitio Web del Proyecto de documentación de Linux: <http://www.linuxdoc.org>. También hay un sitio llamado <http://lucas.linux.org.mx>, en el que se puede encontrar mucha información relacionada con Linux en español.

## Resumen

La lección de hoy trató sobre el entorno de programación y depuración proporcionado por Linux. Se incluyeron introducciones rápidas para:

- Editores (`vi` y `emacs`)
- Compiladores (`gcc` y `g++`)
- Creación de ejecutables con `make`
- Bibliotecas y enlaces
- Depuración con `gdb`
- Control de versiones con `RCS`
- Documentación en línea, incluyendo `man` e `info`

Éste es un entorno rico en características, equivalente al disponible en la mayoría de las estaciones de trabajo profesionales. Ofrece todas las herramientas básicas necesarias para el desarrollo de programas.

## Preguntas y respuestas

- P Compilé y construí mi programa de C++ con gcc. Compila bien, pero tengo muchos errores de enlace.**
- R** gcc enlaza de manera predeterminada con las bibliotecas estándar de C, no con las bibliotecas requeridas por C++. Puede especificarlas en forma explícita; sin embargo, es más sencillo utilizar g++ para la fase de enlace del programa que esté creando.
- P He creado un programa usando bibliotecas compartidas, y he creado las bibliotecas. Ahora, cuando ejecuto el programa, obtengo un error que dice: "cannot open shared object file: No such file or directory".**
- R** No le ha indicado al cargador en dónde encontrar las bibliotecas. La variable de entorno LD\_LIBRARY\_PATH es una lista de rutas separadas por el signo de dos puntos (:) que indican los directorios en los que se pueden encontrar las bibliotecas compartidas.
- P Cuando escribo make, se produce el mensaje "make: nothing to be done for ...".**
- R** make intenta buscar todos los objetos y sus archivos fuente correspondientes. Examina las etiquetas de tiempo en estos archivos. Si el objeto es más reciente que el archivo fuente, make concluye que no es necesario volver a compilar ese objeto. Si su destino existe y todos los objetos componentes son más recientes que sus archivos fuente, make cree que no hay nada por hacer.
- P Cuando ejecuto gdb, no puedo ver mis archivos fuente. Cuando ejecuto xxgdb, la sección superior no muestra ningún archivo fuente, y la sección inferior muestra el mensaje "No default source file yet".**
- R** Para que el compilador preserve los símbolos y los archivos fuente para que el depurador pueda leerlos, debe compilar y enlazar todos los módulos de su programa con la opción -g.
- P He creado un directorio RCS y registrado todos mis archivos fuente y archivos de encabezado. Ahora han desaparecido.**
- R** Cuando registra con el comando ci, RCS actualiza un archivo de historial en el directorio RCS. Es probable que sus archivos no hayan desaparecido; simplemente aún no ha sacado las versiones en las que va a trabajar. Use el comando co para sacar copias bloqueadas o desbloqueadas.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Qué es POSIX?
2. ¿Qué es X Windows?
3. ¿Cuáles son los dos principales editores de texto disponibles en Linux?
4. Cite una de las principales distinciones entre vi y emacs de GNU
5. Cite una de las ventajas de las bibliotecas compartidas en comparación con las bibliotecas estáticas, y cite una de las ventajas de las bibliotecas estáticas en comparación con las bibliotecas compartidas.
6. ¿Qué utilería se usa para compilar y crear programas? ¿Cuál es su archivo de entrada predeterminado?

### Ejercicios

1. Cree una función adicional para el programa de los dados que se mostró en la lección de hoy. La función debe tomar como entrada un apuntador al arreglo Dado. Para cada cara del dado, esta función debe imprimir el porcentaje de veces que salió esa cara. La función debe estar en un archivo separado de main() y de tirarDado().
2. Modifique el archivo make para enlazar la nueva función.
3. Analice el programa paso a paso con gdb.

# SEMANA 4

DÍA 23

## Programación shell

Aunque ha estado estudiando principalmente el lenguaje C++, también ha aprendido acerca de los aspectos específicos y las implicaciones de este lenguaje en el entorno Linux. Continuando con este estilo, la lección de hoy trata sobre los intérpretes de comandos de Linux y su programación, conocida como programación shell.

Aunque el intérprete de comandos es independiente del lenguaje de programación que se utilice, Linux permite crear programas utilizando las características propias del intérprete. Como se habrá imaginado, estos programas son interpretados y reducen el desempeño del sistema; pero la facilidad con que se pueden crear y mantener supera por mucho las expectativas de desarrollo. Por ello, es importante que entienda como trabaja el intérprete de comandos y cómo utilizarlo para poder lograr algo útil con el menor esfuerzo. En el nivel más básico, debe interactuar con el intérprete para lograr cualquier cosa; si comprende el mecanismo básico, descubrirá que es una herramienta poderosa que puede ayudarle en su trabajo.

### Nota

Los términos shell e intérprete de comandos son equivalentes. Se utilizarán indistintamente, a menos que surja alguna ambigüedad entre un programa para el shell y el shell mismo.

Esta lección trata los siguientes conceptos básicos:

- Qué es el shell
- Qué shells están disponibles
- Principios de los shells (redirección y procesamiento en segundo plano)
- Construcción de comandos del shell (sustitución y creación de alias)
- Variables de entorno
- Secuencias de comandos de los shells

## Qué es un shell

Muchas personas piensan que el indicador que se ve al estar sentado frente a una computadora es el sistema operativo, o que lo identifica. Éste no es el caso.

El sistema operativo es el software que dirige a la computadora, habla con el hardware, carga y ejecuta programas, etc. El sistema operativo es algo que, básicamente, nunca se ve.

Cuando se ve el indicador de la computadora y se escriben comandos para ejecutar, con lo que estamos tratando es con el shell. En el pasado, el shell se conocía como intérprete de línea de comandos. De hecho, en el mundo de DOS, este software se conocía comúnmente como CLI, y se llamaba COMMAND.COM.

Una característica interesante de UNIX y sus derivados, incluyendo a Linux, es que los shells son completamente independientes. El usuario tiene la libertad de elegir uno de varios shells disponibles. Como verá más adelante en esta lección, el usuario puede elegir interactuar con un shell pero escribir secuencias de comandos en otro.

## Shells disponibles en Linux

Piense que el shell es un programa que está entre el usuario y el sistema operativo. Este programa implementa el lenguaje que utiliza el usuario para controlar el sistema operativo. El shell se inicia de manera automática al iniciar una sesión en el sistema.

El shell original disponible en los sistemas UNIX era el shell Bourne ("sh"). Después, dos shells que se volvieron populares fueron el shell C ("csh") y el shell Korn ("ksh"). Cada uno tiene características únicas y muy útiles.

El shell Bourne fue reescrito, y la nueva versión se conoce como "Bourne again shell (Shell Bourne nuevamente)", o "bash". El shell C fue reescrito y nombrado shell T ("tcsh"). Los tres shells están disponibles en Linux: bash, tcsh y ksh. Probablemente, bash es el shell que se utiliza con más frecuencia, y en muchas instalaciones es el shell predeterminado.

## Operación de los shells y conceptos básicos de sintaxis

El shell realiza varias funciones importantes.

En primer lugar, es el programa que le permite interactuar con el sistema operativo. La “línea de comandos” es la entrada del usuario para el shell. El shell examina la línea de comandos, determina si lo que se ha escrito es el nombre de un programa (un programa binario compilado), y de ser así, envía ese programa al kernel o núcleo para su ejecución. Si la línea de comandos contiene el nombre de un archivo de secuencia de comandos para algún shell, entonces se llama al shell apropiado que se encarga de interpretar la secuencia de comandos que tiene como entrada.

Todos los comandos de shell utilizan el siguiente formato general:

*comando opcion1 opcion2 opcion3 ... opcionN argumento1 ... argumentoM*

Esta línea se conoce como línea de comandos. Una *línea de comandos* consiste en un nombre de comando y una o más opciones (o *argumentos*). Por lo general, el espacio en blanco de la línea de comandos se ignora. Los nombres de comandos y los argumentos son generalmente sensibles al uso de mayúsculas y minúsculas. El comando se termina oprimiendo la tecla “Entrar”. Puede continuar un comando en una nueva línea usando la barra diagonal inversa (\):

*nombre-de-comando-muy-largo opcion\_larga\_1 opcion\_larga\_2 \ opcion\_larga\_3 ... opcionN*

Puede concatenar más de un comando en una línea con un signo de punto y coma (;), como se muestra a continuación:

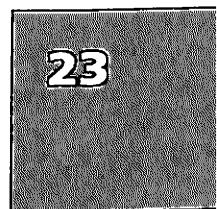
*comando\_1 opcion argumento; comando\_2 opcion argumento; comando\_3 opcion argumento*

### Tip

Tal vez le parezca útil la siguiente secuencia:

*clear; pwd; ls*

Esto limpiará la ventana de terminal, y luego, en la ventana limpia, imprimirá el nombre del directorio actual y una lista de todos los archivos que estén en ese directorio.



## Características del shell

Los shells tienen demasiadas características como para hablar detalladamente sobre ellas aquí. Esta sección habla sobre los fundamentos de los shells en general. Estas características están presentes en casi todos los shells. La sintaxis exacta puede diferir de un shell a otro, y por ende los ejemplos específicos se limitan a bash. Así pues, usted notará que cada uno de los shells populares agrega alguna característica que tal vez no se encuentre disponible en los demás.

Todos los shells tienen entradas completas y concisas en las páginas del manual. Además, el archivo info para bash contiene aproximadamente 3,000 líneas. Para obtener información más específica o avanzada, debe consultar esta documentación en línea.

## Redirección de E/S

Cuando se ejecutan programas en Linux, se abren automáticamente tres archivos de E/S para ellos. Estos archivos son la *entrada estándar*, la *salida estándar* y el *error estándar*. Tal vez le parezca confuso, pero todos los sistemas UNIX están basados en el manejo de archivos. Si usted quiere enviar información a través de un módem, debe enviar los mensajes a su archivo asociado que comúnmente es el archivo /dev/modem.

De manera predeterminada, la entrada estándar está conectada al teclado, y la salida y el error estándar se conectan a la pantalla. Como el shell inicia el programa que usted especifica, puede reasignar estas asignaciones predeterminadas en un proceso conocido como redirección de la entrada, redirección de la salida, o de manera más genérica, redirección de E/S, antes de que el programa inicie.

Suponga que quiere crear una lista de archivos en el directorio /usr/include que incluya otros archivos. Una forma de hacer esto es la siguiente:

```
grep -l "#include" /usr/include/*.h > ListaInclude
```

El signo “\*” es un carácter “comodín”. Hablaremos con mayor detalle sobre los comodines más adelante en esta lección. Por ahora, sólo asuma que grep comprobará todos los archivos del directorio /usr/include cuyos nombres terminen con “.h”. Imprimirá el nombre de cada archivo en el que encuentre la cadena #include.

El “>” es el carácter de redirección de salida. Ocasiona que el shell redireccione la salida del comando grep a un archivo llamado ListaInclude. El mismo comando grep no está consciente de la redirección; éste es trabajo del intérprete de comandos. Los nombres que están en el archivo ListaInclude se verán así:

```
/usr/include/FlexLexer.h  
/usr/include/Fnlib.h  
/usr/include/Fnlib_types.h  
/usr/include/Imlib.h  
/usr/include/Imlib_private.h  
/usr/include/Imlib_types.h  
/usr/include/QwCluster.h  
/usr/include/QwPublicList.h  
/usr/include/QwSpriteField.h
```

Para reemplazar el /usr/include/ al principio de cada nombre de archivo, puede utilizar el comando sed:

```
sed 's#^/usr/include/#+ #' < ListaInclude > ListaIncludes
```

El comando sed opera sobre los datos de la entrada estándar. La entrada estándar se redirige por medio del carácter “<” de redirección de entrada. En lugar de leer del teclado, sed leerá del archivo ListaInclude. Como con la redirección de la salida, el comando sed no sabe que su entrada estándar ha sido redirigida (a menos que verifique esto explícitamente). La salida del comando sed se redirige al archivo ListaIncludes.

Observe que el carácter “>” no redirige la salida de error estándar, sólo la salida estándar. Utilice >& para redirigir el error estándar de la siguiente manera:

```
sed 's#^/usr/include/#+ #' < ListaInclude >& ErrorSed > SalidaSed
```

El siguiente comando redirige tanto el error estándar como la salida estándar al archivo llamado Salida:

```
sed 's#^/usr/include/#+ #' < ListaInclude >& Salida
```

Esto es algo confuso y se utiliza muy poco. Es más común que se redirija sólo la salida estándar, y que se permita que el error estándar vaya a la pantalla.



## Tuberías

Una forma relacionada de redirección se conoce como *tubería*.

Tal vez quiera una lista de archivos que contengan #include, y quiera que la lista esté ordenada.

Puede usar la redirección de la siguiente manera:

```
grep -l "#include" /usr/include/*.h > ListaInclude  
sort ListaInclude > ListaOrdenada
```

### Nota

El comando sort ListaInclude > ListaOrdenada no está redireccionando la entrada. Muchos comandos tienen como entrada los archivos que se escriben en su lista de argumentos (no de opciones), y en caso de no haber argumentos leen de la entrada estándar. Éste es el caso de sort y de varios comandos más que se conocen como “filtros”. Para el usuario será equivalente escribir sort ListaInclude y sort < ListaInclude. Sin embargo, para el intérprete, sort ListaInclude es un comando completo y lo envía al núcleo directamente; por el contrario, sort < ListaInclude contiene un redireccionamiento “<” que el shell debe interpretar antes de enviar el comando al núcleo.

Evidentemente, debe haber una mejor manera que utilizar dos comandos y un archivo temporal. A continuación se muestra algo mejor:

```
grep -l "#include" /usr/include/*.h | sort > ListaOrdenada
```

El carácter de tubería (!) encadena dos comandos y conecta (redirecciona) la salida estándar del primero a la entrada estándar del segundo. Una sola línea de comando puede contener cualquier número de tuberías:

```
grep -l "#include" /usr/include/*.h | sort | sed 's#^/usr/include/#+ #' >  
SortedModifiedList
```

## Variables

Con frecuencia, un programa necesita cierta información específica para poder funcionar correctamente. Por ejemplo, los editores de pantalla vi y emacs necesitan saber qué tipo de terminal se está utilizando.

Esta información se podría proporcionar como opción de línea de comandos; sin embargo, sería innecesariamente tedioso requerir que usted agregara esta información cada vez que iniciara el editor. Además, hay más información que no cambia de una invocación a otra, además del tipo de terminal que el editor requiere.

Los shells se encargan de este problema con las variables de entorno. Una *variable de entorno* es simplemente un par nombre/valor. El shell mantiene una lista de estas variables y las hace disponibles para cualquier programa.

En realidad, existen dos tipos de variables: las variables normales de shell y las variables de entorno. La distinción entre estos dos tipos es delicada. Las variables de entorno también se conocen como variables “globales”, mientras que las variables de shell también se conocen como variables “locales”. Las variables globales se pasan al inicio de un nuevo comando o shell. Para establecer una variable de shell en bash, utilice esta sintaxis:

`NOMBRE=valor`

Para establecer una variable de entorno, la sintaxis es

`export NOMBRE=valor`

Si la variable incluye espacios en blanco o cualquier otro carácter especial, el valor se puede encerrar entre comillas sencillas o dobles. A menudo es útil agregar valores a una variable ya existente. Puede hacerlo de la siguiente manera:

`export NOMBRE="$NOMBRE nueva_cadena"`

Esto agrega “nueva\_cadena” a la variable de entorno NOMBRE.

## Variables utilizadas por el shell

Ciertas variables de entorno tienen un significado especial y son utilizadas por el shell. A continuación se muestra una lista parcial de algunas de las variables más notables.

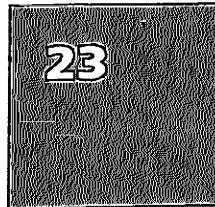
- **DISPLAY**—Esta variable la leen los programas X para saber en dónde desplegar su salida. Por lo general se establece en “`:0.0`”, lo que significa que la salida se desplegará en la primera terminal gráfica (virtual) del equipo host.
- **PATH**—Una lista de nombres de directorios separados por el signo de dos puntos (`:`), en la que el shell debe buscar los programas. Cuando usted escribe cualquier nombre de comando, el shell buscará en estos directorios un programa con ese nombre.
- **TERM**—El tipo de terminal o de emulación de terminal. Los programas, como los editores, deben saber el tipo de terminal para poder enviar los comandos apropiados para manipular la pantalla y el cursor.

- PS1—El indicador desplegado por el shell para indicar al usuario que está listo para recibir entrada.
- HOME—El directorio personal del usuario.

## Variables establecidas por el shell

Ciertas variables son establecidas por el shell y pueden ser referenciadas por programas iniciados desde el shell. A continuación se muestra una lista parcial de variables importantes establecidas por el shell:

- SHELL—Contiene el nombre completo de la ruta del shell actual (por ejemplo, /bin/bash).
- PWD—El directorio actual que haya sido establecido por el comando cd más reciente.



## Procesamiento en segundo plano, suspensión y control de procesos

Por lo general, usted escribe el nombre de un comando en la línea de comandos y espera a que ese comando termine. Suponga que el comando que ejecuta va a tardar mucho. En lugar de esperar, podría abrir otra ventana xterm y continuar trabajando ahí. Sin embargo, en vez de eso puede utilizar la naturaleza multitareas de Linux. Puede ejecutar el comando en segundo plano con el carácter especial “&” de la siguiente manera:

```
bash# find . -name '*.c' -print | sort > ListaOrdenada &
[1] 142
bash#
```

Este comando crea una lista de todos los archivos cuyos nombres terminen con “.c”. Ordena esta lista y coloca la salida en un archivo llamado ListaOrdenada. El shell imprime “[1] 142” y regresa inmediatamente, listo para recibir otro comando. La salida indica que se está ejecutando una tarea en segundo plano, y que el PID (identificador del proceso) es 142. Puede ejecutar más de un proceso en segundo plano. En este caso, el proceso es el número 1 en la cola de procesamiento en segundo plano.

Si ejecuta un programa desde la línea de comandos y no utiliza el carácter &, el shell espera que el proceso termine antes de pedirle un nuevo comando. Se dice que este proceso se está ejecutando en primer plano.

Si ejecuta un proceso en segundo plano, y luego decide que mejor quiere esperar a que termine, lo puede “traer al primer plano” con el comando fg.

Si está ejecutando un proceso en primer plano y quiere suspenderlo sin eliminarlo permanentemente, oprima “Ctrl+Z” (la tecla Control y “Z” al mismo tiempo). Ahora el proceso está suspendido. Puede hacer que el proceso siga ejecutándose en primer plano con fg, o en segundo plano con bg.

Por último, si quiere eliminar permanentemente el proceso, puede utilizar el comando `kill`. Puede eliminarlo usando su número de PID (`kill -142`), o por su lugar en la cola de procesamiento en segundo plano (`kill %1`).

Para averiguar qué procesos se están ejecutando en segundo plano, utilice el comando `jobs`.

La figura 23.1 muestra el control de procesos. El primer comando “find” se ejecuta en segundo plano, y el comando “jobs” indica que se está ejecutando ahí. El siguiente comando “find” se ejecuta en primer plano y luego se suspende con “Ctrl+Z”. De nuevo, el comando “jobs” muestra esto. A continuación, el comando “find” suspendido se reanuda en segundo plano, y esto se verifica con el comando “jobs”. El comando “fg” se utiliza para que el primer proceso que se ejecuta en segundo plano pase al primer plano, y luego se suspende con “Ctrl+Z”.

**FIGURA 23.1**

## *Procesamiento en segundo plano, suspensión y control de procesos.*

```
[1] 1245 1246
[1]@shibuya ~% find . -name '*.c' -print | sort > /dev/null s
[1]@shibuya ~% jobs
[1] + Running           find . -name '*.c' -print | sort >
/dev/null
[1]@shibuya ~% find . -name '*.h' -print | sort > /dev/null
[1]@shibuya ~% suspended
[1]@shibuya ~% jobs
[1] + Running           find . -name '*.c' -print | sort >
/dev/null
[2] + Suspended         find . -name '*.h' -print | sort >
/dev/null
[1]@shibuya ~% bg %2
[2]  find . -name '*.h' -print | sort > /dev/null s
[1]@shibuya ~% jobs
[1] + Running           find . -name '*.c' -print | sort >
/dev/null
[2] + Running           find . -name '*.h' -print | sort >
/dev/null
[1]@shibuya ~% fg %1
find . -name '*.c' -print | sort > /dev/null
z
[1]@shibuya ~% suspended
[1]@shibuya ~% jobs
[1] + Suspended         find . -name '*.c' -print | sort >
/dev/null
[2] + Running           find . -name '*.h' -print | sort >
/dev/null
[1]@shibuya ~% bg %1
[1]  find . -name '*.c' -print | sort > /dev/null s
[1]@shibuya ~% jobs
[1] + Running           find . -name '*.c' -print | sort >
/dev/null
[2] + Running           find . -name '*.h' -print | sort > .
[1]@shibuya ~% kill %1
[1] Terminated
[1]@shibuya ~% jobs
[2] + Running           find . -name '*.c' -print | sort > /dev/null
[1]@shibuya ~% done
[1]@shibuya ~%
[root@shibuya ~]%
```

## Completabilitat de comandos

bash incluye muchos métodos abreviados para reducir la cantidad de escritura que usted necesite realizar. Esto se logra mediante la completación y la sustitución de comandos. La siguiente sección trata sobre la sustitución y el mecanismo de comodines.

La *completación de comandos* se refiere a la habilidad de bash para “adivinar” el comando o nombre de archivo que usted está escribiendo. Escriba algunos de los primeros caracteres de un comando y, en lugar de escribir el nombre completo del comando, oprima “Tab”. Si bash puede identificar ese comando en forma única, lo completará por usted. En caso contrario, sonará un bip. Oprima “Tab” una segunda vez y se mostrarán los comandos más aproximados a lo que usted está escribiendo.

El mismo mecanismo funciona si está especificando un nombre de archivo como argumento para un comando.

Por ejemplo, escriba `mo`, oprima "Tab", y el shell deberá emitir un bip. Oprima "Tab" otra vez y deberá aparecer una lista de comandos que empiecen con "mo". Escriba `r` y oprima "Tab" otra vez. El shell deberá completar el comando `more`. Ahora oprima la barra espaciadora, luego escriba `/etc/in` y oprima "Tab". El shell emitirá un bip. Oprima "Tab" de nuevo y se mostrará una lista de archivos en `/etc` que empiecen con "in". Escriba `e` y luego oprima "Tab". Se completará el nombre de archivo `inetd.conf`.



## Sustitución de comandos

El shell incluye otros mecanismos para ahorrarse la escritura. Estos mecanismos incluyen cadenas de sustitución. Se permiten varios tipos de sustituciones.

### Sustitución mediante comodines

En muchos juegos de cartas se puede utilizar un comodín para reemplazar cualquier otra carta. De la misma manera, en el shell se puede utilizar un comodín para guardar un lugar que el shell puede reemplazar con cualquier otro carácter o caracteres.

Existen dos caracteres comodines importantes: el asterisco (\*), para representar cualquier secuencia de cero o más caracteres de un nombre de archivo, y el signo de interrogación (?), que representa cualquier carácter individual.

¿Qué significa esto? Tal vez usted quiera ejecutar el comando `ls` en el directorio actual para ver una lista de todos los nombres de archivos que terminen con ".h". Puede simplemente escribir `ls *.h`. El shell expandirá el carácter comodín a una lista de todos los archivos que terminen con ".h" antes de invocar el comando `ls`. La lista completa de nombres de archivo se pasará al comando.

Tal vez quiera ver todos los nombres de archivos que contengan exactamente tres caracteres antes de ".h". Para esto, puede escribir `ls ??? .h`.

En bash, la sustitución de comodines es mucho más sofisticada que en DOS. bash no tiene problemas para expandir "a??def\*.xyz" a una lista de nombres de archivos que empiecen con una "a" seguida por 2 caracteres cualesquiera, seguidos por "def", por cero o más caracteres y que terminen con ".xyz".

#### Nota

Es importante recordar que los comodines (y cualquier otra sustitución descrita en el resto de esta lección) se expanden antes de que se ejecute el comando. Es decir, el shell los interpreta antes de enviarlos al núcleo.

## Sustitución mediante cadenas

bash permite la sustitución de secuencias específicas de caracteres. Existen dos estilos de sustitución.

Puede especificar una lista separada por comas de cadenas entre llaves, y cada una se utilizará en orden. Por ejemplo:

```
bash# ls a{b,c,de,fgh}z  
abz acz adez afghz  
bash#
```

La “a” y la “z” se combinan en el siguiente orden con las cadenas que aparecen entre llaves: primero con “b”, luego con “c”, luego con “de” y al último con “fgh”.

Puede especificar rangos de caracteres con corchetes:

```
bash# ls a[b-h]z  
abz acz adz aeaz afz agz ahz  
bash#
```

Los corchetes también pueden especificar caracteres específicos:

```
bash# ls a{bcde}z  
abz acz adz aeaz  
bash#
```

Por último, puede mezclar comodines, llaves y corchetes.

## Sustitución mediante la salida de un comando

Otra forma de sustitución es mediante la salida de un comando. La salida de un comando se puede especificar como argumento para otro comando:

```
bash# ls -l `find /usr/src -name Makefile -print`
```

Este ejemplo ejecutará el comando `find` para localizar todos los archivos `make` que estén en el árbol de directorio `/usr/src`. La lista de archivos se presentará en la línea de comandos a `ls`, el cual mostrará las entradas en el directorio de estos archivos.

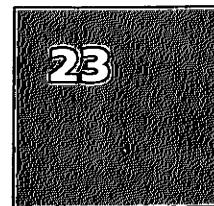
## Sustitución mediante variables

Una última forma útil de sustitución es mediante variables. El shell puede reemplazar una variable con su valor en la línea de comandos.

Por ejemplo, suponga que su directorio personal es `/home/nombreusuario`:

```
bash# echo ${HOME}  
/home/nombreusuario  
bash#
```

El valor de la variable de entorno llamada “`HOME`” reemplaza la cadena  `${HOME}` en el argumento de la línea de comandos.



## Historial y edición de comandos

bash mantiene una lista de los comandos que usted ha escrito en lo que se conoce como una lista del historial. La longitud de la lista depende del entorno, pero por lo general no es mayor de 500 comandos. bash preservará esta lista de sesión en sesión. Si escribe el comando `history`, se mostrará la lista de comandos que ha escrito. Por ejemplo, considere la siguiente secuencia de comandos:

```
bash# cd /
bash# cd /tmp
bash# cd
bash# cat .bashrc > /dev/null
bash# history
1 cd /
2 cd /tmp
3 cd
4 cat .bashrc > /dev/null
5 history
```

Para invocar cualquier comando anterior, escriba un signo de admiración y el número del comando. Para repetir el comando `cd /tmp` de la secuencia anterior, escriba `!2`.

Puede repetir el último comando con `!!`, los dos últimos comandos con `!-2`, y así sucesivamente.

También puede editar una línea de comandos anterior antes de repetirla. Suponga que escribió el comando `ls -l /TMp`. Para corregir este comando y repetirlo, podría escribir `^TMp^tmp^`. bash da por hecho que usted quiere editar el comando anterior y procede con la sustitución, de ser posible. También hay soporte para la edición estilo emacs. Oprima “`Ctrl+P`” y se mostrará el comando anterior. Utilice las teclas de dirección o “`Ctrl+B`” (para moverse un carácter hacia atrás) y “`Ctrl+F`” (para moverse un carácter hacia adelante) y corrija la línea de comandos. Oprima “Entrar” para ejecutar el comando editado.

## Creación de alias de comandos

Probablemente utilice con frecuencia ciertos comandos o secuencias de comandos. Puede crear sus propios métodos abreviados para estos comandos, lo que se conoce como alias. El shell reemplaza un alias con su definición.

Por ejemplo, el comando `ls` le muestra los nombres de archivos. Con la opción de línea de comandos `-F`, se indica también el tipo del archivo (“\*” para ejecutables, “/” para directorios, etcétera). Puede crear un alias para sustituir comandos de la siguiente manera:

```
bash# ls -F
a b* c* d e f/ g/
bash# ls
a b c d e f g
bash# alias ls="ls -F"
bash# ls
a b* c* d e f/ g/
bash#
```

El comando `ls` con y sin el argumento `-F` muestra sus resultados de manera diferente. Al crear un alias para `ls`, y después cuando se escribe el comando `ls`, el shell sustituye el alias automáticamente: `ls -F`.

El comando `rm` elimina un archivo. Con la opción `-i`, primero pedirá la confirmación de la operación. Muchas personas encuentran útil sustituir el comando `rm` para evitar eliminar accidentalmente un archivo: `alias rm="rm -i"`. Las comillas en este alias de comando son importantes, ya que agrupan todo el lado que está a la derecha del signo de igualdad como un solo argumento para el alias del comando. Sin las comillas, el alias del comando trataría de interpretar los caracteres `-i` como una opción.

## Secuencias de comandos de los shells

En un archivo se pueden colocar secuencias complejas de comandos de shell, para que se puedan repetir en cualquier momento. Esto es muy parecido a la escritura de un programa, sólo que no se necesita la compilación. El shell tiene muchas de las características de los lenguajes estándar, incluyendo variables e instrucciones de control (de las que aún no hemos hablado).

Casi todas las secuencias de comandos de shell empiezan con `#!/bin/sh` (o con el intérprete de comandos deseado). Los primeros dos caracteres indican al sistema que éste es un archivo de secuencia de comandos, y `/bin/sh` inicia el shell `bash`. Esto se debe a que `/bin/sh` es un vínculo al programa `/bin/bash`; si se desea utilizar otro intérprete, se debe especificar en esta línea. Por ejemplo, los archivos de secuencias de comandos para `tcsh` deberán tener como primera línea `#!/bin/csh` o `#!/bin/tcsh` y las secuencias de comandos para `perl` incluirán la línea `#!/usr/bin/perl`. Después de iniciar el shell, éste recibe, una por una, las líneas restantes del archivo.

Las secuencias de comandos de shell pueden ser programas simples de una línea, o complejos con cientos de líneas. No son tan rápidos y eficientes como los programas de lenguajes compilados.

Las secuencias de comandos de shell deben tener encendido su bit de permiso de "ejecución". Puede encender este bit con el comando: `chmod a+x nombrearchivo`.

## Variables

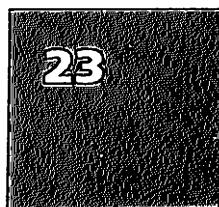
Ya hemos hablado sobre las variables de los shells. Cuando se está ejecutando una secuencia de comandos, ya están definidas algunas variables útiles:

- `$$`—El identificador del proceso que está ejecutando el programa de shell
- `$0`—El nombre de la secuencia de comandos
- `$1` hasta `$9`—Los primeros nueve argumentos de línea de comandos que se pasan a la secuencia de comandos
- `#$`—El número de parámetros de línea de comandos que se pasan a la secuencia de comandos

## Estructuras de control

bash soporta instrucciones de control que son parecidas a muchas de las que hay en los lenguajes C y C++, aunque la sintaxis es diferente. Entre estas instrucciones se incluyen `if -then -else`, `for`, `case` y `while`.

La secuencia de comandos del listado 23.1 proporciona una simple muestra de las principales instrucciones de control disponibles en una secuencia de comandos de bash.



### ENTRADA LISTADO 23.1 Principales instrucciones de control de bash

```
1:#!/bin/bash
2:
3:MAX=9
4:
5:# 
6:# Muestra del control 'if-endif'
7:if [ $# -gt $MAX ]
8:then
9:    echo "$0 : $MAX o menos argumentos requeridos"
10:   exit 1
11:fi
12:
13:# 
14:# imprimir los primeros 2 argumentos
15:echo "$0 : Parámetro 2 : $2"
16:echo "$0 : Parámetro 1 : $1"
17:echo ""
18:
19:# 
20:# Muestra del control 'for-done'
21:for i in $1 $2
22:do
23:    ls -l $i;
24:done
25:echo ""
26:
27:# 
28:# muestra del control 'case'
29:echo "ejemplo de case"
30:for i
31:do
32:    case "$i" in
33:        a) echo "case a";;
34:        b) echo "case b";;
35:        *) echo "default case $i";;
36:    esac
37:done
38:echo ""
39:
40:# 
41:# muestra del control 'while-done'
```

continúa

**LISTADO 23.1** CONTINUACIÓN

---

```
42: echo "ejemplo de while"
43: i=1;
44: while [ $i -le $# ]
45: do
46:   echo $i;
47:   i=$[$i+1];
48: done
49: echo ""
50:
51: #
52: # muestra del control 'until-done'
53: echo "ejemplo de until"
54: i=1;
55: until [ $i -gt $# ]
56: do
57:   echo "argumento $i";
58:   i=$[$i+1];
59: done
60: echo ""
61:
62: exit 0
```

---

**SALIDA**

```
bash# touch a b
bash# ls -F
a b lst23-01.sh*
bash# ./lst23-01.sh a b el zorro café rápido salta
./lst23-01.sh : Parámetro 2 : b
./lst23-01.sh : Parámetro 1 : a

-rw-r--r--    1 hal      hal          0 Jan 19 22:33 a
-rw-r--r--    1 hal      hal          0 Jan 19 22:33 b

ejemplo de case
case a
case b
default case el
default case zorro
default case café
default case rápido
default case salta

ejemplo de while
1
2
3
4
5
6
7

ejemplo de until
argumento 1
```

```
argumento 2
argumento 3
argumento 4
argumento 5
argumento 6
argumento 7
bash#
```

**ANÁLISIS**

La línea “MAX=9” asigna el valor 9 a la variable de shell llamada MAX. El grupo de líneas “if-fí” que le sigue comprueba el número de argumentos de línea de comandos. Si hay más de 9, entonces se imprime un mensaje y la secuencia de comandos termina.

**23**

Las instrucciones “echo” muestran cómo se pueden referenciar los argumentos de línea de comandos. El shell sustituye todos los argumentos de la línea de comandos (incluyendo el nombre del comando) por las variables \$0, \$1, \$2 y así sucesivamente.

Las siguientes agrupaciones de comandos muestran las principales instrucciones de control disponibles en el shell. Entre éstas se incluyen ‘for-do-done’ (similar a un ciclo ‘for’ de C), ‘case-esac’ (similar a la instrucción ‘switch’ de C), ‘while-do-done’ (similar al ciclo ‘while’ de C) y ‘until-do-done’ (similar al ‘do-while’ de C). Consulte las páginas del manual de bash para más detalles acerca de la sintaxis y el uso de éstas y de otras instrucciones de control.

El comando `touch a b` creará los archivos a y b vacíos, es decir, sólo creará la entrada en el directorio pero no colocará nada dentro de ellos. El comando ‘ls -F’ muestra que hay 2 archivos (llamados a y b) en el directorio actual, además de la secuencia de comandos de shell llamada ‘lst23-01.sh’. Luego se invoca la secuencia de comandos con 7 argumentos. Los primeros 2 argumentos son los nombres de los 2 archivos.

## Archivo(s) de inicio de shell

Un shell interactivo (a diferencia de un shell no interactivo, como el que se inicia para una secuencia de comandos) es uno en el que se conecta la entrada estándar con el teclado y la salida estándar con la pantalla. Si bash es el shell predeterminado para el usuario, al momento del inicio de sesión se inicia un shell bash interactivo. Si se ejecuta una secuencia de comandos para el shell bash, se inicia un shell no interactivo y recibe la secuencia de comandos una línea a la vez.

Al momento del inicio de sesión, el shell busca una secuencia de comandos llamada `/etc/profile`. Si encuentra esa secuencia de comandos, la ejecuta. Si el archivo `.bash_profile` existe en el directorio personal del usuario, se ejecuta. Si no existe, se ejecuta `.profile`.

Al momento de cerrar la sesión, se ejecuta el archivo `.bash_logout` que se encuentra en el directorio personal, si es que existe.

Estos archivos de inicio se utilizan para establecer la ruta, el indicador y otras variables de entorno que el usuario necesite establecer al momento de iniciar cada sesión. Es común ejecutar comandos y aplicaciones desde estos archivos. Por ejemplo, si el usuario quiere que se abra el cliente de correo electrónico cada vez que inicia una sesión, sólo debe agregar el nombre del programa al final del archivo `.bash_profile` o `.profile`. Otro ejemplo común es eliminar todos los archivos core de su directorio de trabajo cada que termina una sesión. Para ello sólo escriba el comando

```
find $HOME -name core -type f -exec rm -rf {} \;
```

al final del archivo `.bash_logout` o `.logout`. Una desventaja es que los comandos que agregue a `.bash_profile` se ejecutarán cada vez que abra una terminal (aunque ya esté en sesión), así que no abuse de estas características.

## Resumen

Esta lección ha sido una descripción de los shells en general, y de bash en particular. La lección cubrió los siguientes aspectos:

- Redirección
- Procesamiento en segundo plano
- Historial
- Edición de línea de comandos, comodines y sustitución

También hablamos brevemente sobre las secuencias de comandos de los shells.

El shell es un programa que permite que el usuario interactúe con el sistema operativo. Proporciona características que facilitan esta interacción (como el historial, la completación de comandos y el control de procesos), así como muchas de las características de un lenguaje de programación (incluyendo variables, sustitución de variables e instrucciones de control). Existen varios shells disponibles para Linux, aunque bash es tal vez el que más se utiliza. El manual en línea proporciona una excelente referencia para la sintaxis y las características del shell.

## Preguntas y respuestas

**P** Escribo un comando común de Linux pero recibo el mensaje de error “command not found”.

**R** Éste es un problema común. Tal vez haya escrito mal el comando, o tal vez no exista. Lo más probable es que haya un error en su ruta de acceso. bash (además de otros shells) sólo ejecutará un archivo si puede encontrarlo en la ruta de ejecución. Esta ruta se establece por medio de la variable de entorno `PATH`. Puede determinar cuál

es esta ruta con el comando `echo ${PATH}`. Con los comandos `man` o `find` puede encontrar el directorio en el que se encuentra el comando faltante. Si ese directorio no se encuentra en la lista de la ruta, agréguelo con `export PATH="${PATH}:nuevo_directorio"`.

- P** He creado una secuencia de comandos de shell, establecí su permiso de ejecución y escribí su nombre. Recibo el error “command not found”.
- R** Compruebe que el “.” se encuentre en su variable de entorno `PATH`. El “.” significa “el directorio actual”.

23

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

1. ¿Qué es un shell?
2. ¿Cuál es la sintaxis general de una línea de comandos de shell?
3. ¿Cuáles son los tres archivos de E/S disponibles para los programas?
4. ¿Cuáles son las 3 formas de redirección de E/S, y qué caracteres se utilizan para representarlos en la línea de comandos?
5. ¿Qué son las variables de entorno, “locales” o “globales”? ¿Y las variables de shell?
6. ¿Cuál variable de entorno de bash establece la ruta de búsqueda de comandos?  
¿Cuál establece el indicador de comandos?
7. Nombre 2 caracteres de sustitución (comodines) de la línea de comandos.
8. ¿Qué necesita haber en un archivo de secuencia de comandos de shell para que el shell sepa a cuál intérprete debe enviar la secuencia de comandos?

### Ejercicios

1. Escriba una secuencia de comandos de bash para imprimir todos los argumentos de línea de comandos, además del número total de argumentos que reciba.



# SEMANA 4

DÍA 24



## Programación de sistemas

La lección de hoy trata sobre la programación del sistema Linux usando el lenguaje de programación C++. La programación del sistema Linux es el proceso de escribir programas usando las llamadas al sistema proporcionadas por el sistema operativo Linux. Las llamadas al sistema son la interfaz de programación primaria que interactúa con los diversos componentes del sistema operativo. La programación de sistemas es un tema muy amplio. La lección de hoy se enfoca en el desarrollo de programas usando procesos y subprocesos.

### Procesos

Un *proceso* es un programa en ejecución. Un programa es una imagen binaria en disco o en cualquier otro medio de almacenamiento masivo, por ejemplo, ps. ps es una utilería de Linux que despliega el estado del proceso. Cuando se ejecuta un programa, se lee del área de almacenamiento en el dispositivo periférico, que es en donde reside, se envía a la memoria y luego se ejecuta.

Usando ps como ejemplo, cuando un usuario escribe ps en el indicador de comandos del sistema, el programa ps se lee del dispositivo de almacenamiento masivo, se envía a la memoria y se ejecuta.

Cada proceso de Linux consta de varios segmentos de programa, recursos del sistema e información del estado de ejecución. Los segmentos son la pila, el heap, los datos y el texto. La organización de los segmentos que se encuentran en la

memoria se muestra en la figura 24.1. La información sobre el estado de ejecución del proceso se conoce como *contexto del proceso*. El contexto del proceso consiste en el espacio de direcciones, espacio de la pila, contenido de los registros, estructuras de datos del núcleo y el estado del proceso (mismo que no debe confundir con el estado de ejecución del proceso).

Cada proceso de Linux se ejecuta en su propio entorno, y cada proceso tiene su propia memoria virtual y espacio de direcciones. En Linux no es posible afectar indirectamente la ejecución de otro proceso.

**FIGURA 24.1**  
*Diagrama de un proceso de Linux.*



## Creación y terminación de procesos

Un proceso nuevo se crea cuando un programa en ejecución utiliza la llamada de sistema conocida como `fork()`. `fork()` crea un proceso nuevo y luego copia el entorno del proceso padre en el proceso hijo recién creado.

**Tip**

El programa que llama a `fork()` se conoce como proceso padre. El proceso recién creado se conoce como hijo.

En Linux, cuando se crea un proceso nuevo, los segmentos de datos del proceso padre no se copian por medio de `fork()`. Inicialmente, los permisos de acceso a los segmentos del proceso padre se establecen como de sólo lectura, y los comparten los procesos padre e hijo. Cuando un proceso hijo accede al segmento del proceso padre, ese segmento se copia en el entorno del proceso hijo. El proceso de retrasar la copia de los segmentos del proceso padre hasta que el proceso hijo acceda a ellos se conoce como *Copiar en la escritura* (*Copy on Write*). Copiar en la escritura optimiza la llamada a `fork()` minimizando la cantidad de información copiada del entorno del proceso padre durante dicha llamada.

Cuando se hace una llamada a `fork()`, ésta regresa dos veces. Cuando termina la llamada a `fork()` del proceso padre, ésta regresa el ID del proceso hijo, y cuando termina la llamada a `fork()` del proceso hijo, regresa un 0. Tanto en el padre como en el hijo, la ejecución del programa continúa justo después de que `fork()` termina. El proceso es un identificador único en el sistema; a cada proceso se le asigna un ID de proceso.

El listado 24.1 muestra un ejemplo de una interfaz que crea un proceso y puede determinar si es el proceso padre o el hijo.

### Nota

En los listados de este capítulo se utiliza el nombre del archivo como aparece en el CD-ROM que viene en el libro. Por convención, en la programación de sistemas se utilizan nombres específicos para bibliotecas, archivos de encabezado y funciones, y aquí se respeta esa convención. Muchos listados tienen un nombre entre paréntesis al final de la primera línea; usted puede guardar ese listado con dicho nombre y después hacer las llamadas a estos archivos con la instrucción #include "archivo.h" que aparece en los listados subsecuentes.

Como ejemplo, vea la línea 1 del listado 24.2; aparece el nombre process1.h entre paréntesis. Guarde ese listado con este nombre y después cambie la línea 3 del listado 24.3 para que aparezca la instrucción #include "process1.h", en lugar de #include "lst24-02.h".

**24**

### ENTRADA LISTADO 24.1 Listado para la clase Process

```

1: // Listado 24.1 Clase Process (process.h)
2:
3: #ifndef C_PROCESS_H
4: #define C_PROCESS_H
5:
6: class Process
7: {
8: public:
9: Process();
10: ~Process();
11: void Create();
12: bool isParent()
13: { return (pid != 0); }
14: bool isChild()
15: { return (pid == 0); }
16: private:
17: Process & operator= (const Process &); // no permitir la copia
18: int pid;
19: };
20:
21: #endif

```

### ANÁLISIS

La función miembro Process::Create() de la línea 11 llama a fork() para crear un nuevo proceso. Para determinar cuál proceso se está ejecutando, si el padre o el hijo, se proporcionan dos funciones receptoras, Process::isParent(), en la línea 12, y Process::isChild(), en la línea 14. Debido a que un proceso tiene información del sistema y datos del núcleo, esta clase necesita evitar explícitamente que un usuario copie la clase. Esto se logra haciendo que el constructor de copia sea privado.

**Tip**

En los programas de sistema es común utilizar recursos que pertenecen al kernel. Como la copia de recursos del kernel puede producir efectos indeseables, todas las clases definen al constructor de copia como función miembro privada sin implementación. Al hacer esto, garantiza que los usuarios de la clase no puedan tener acceso al constructor de copia, y hace que las funciones miembro puedan compilar el código pero no enlazarlo.

Cuando se crea un proceso hijo, nunca existe garantía sobre cuál proceso, padre o hijo, se ejecutará primero. En el ejemplo anterior, el proceso padre espera, por medio de la llamada a `sleep()`, y deja que el proceso hijo se ejecute primero. Este método es insuficiente para aplicaciones reales. Hay una condición en la que no es posible determinar cuál proceso se ejecutará a continuación, la cual se conoce como *condición de carrera*.

Un proceso utiliza la llamada de sistema conocida como `exit()` para terminar. `exit()` toma un parámetro, el código de estado, y regresa ese valor al proceso padre.

## Control de procesos

En Linux se utilizan varias llamadas de sistema para el control de procesos. Esta sección se enfoca en las rutinas básicas para el control de procesos.

### `exec()`

La familia `exec()` de llamadas de sistema reemplaza la imagen actual del proceso, o programa, con la imagen especificada en el argumento para `exec()`. Recuerde desde la definición de un proceso que éste es un programa en ejecución. `fork()` crea un proceso nuevo que es una copia del proceso actual, proporcionando efectivamente dos instancias en ejecución del mismo programa. `exec()` le proporciona la capacidad para ejecutar un programa diferente en un proceso que ya se encuentre en ejecución.

Un proceso hijo utiliza la familia `exec()` de funciones para ejecutar otro programa. Una vez que se crea el proceso hijo, éste se cubre a sí mismo con otro programa y luego continúa su ejecución.

### `wait()`

La llamada de sistema conocida como `wait()` ocasiona que el proceso padre se suspenda a sí mismo hasta que termine el proceso hijo o hasta que reciba una señal. Un uso típico de `wait()` sería que el padre llamaría a `wait()` inmediatamente después de una llamada a `fork()` para esperar a que se complete el proceso hijo.

En el listado 24.2 se extiende la clase `Process` definida en el listado 24.1 con dos nuevas funciones miembro que implementan a `exec()` y a `wait()`.

**ENTRADA****LISTADO 24.2** Clase Process con funciones miembro adicionales para control de procesos

```

1: // Listado 24.2 Clase Process (process1.h)
2:
3: #include <wait.h>
4: #include <unistd.h>
5:
6: #ifndef C_PROCESS1_H
7: #define C_PROCESS1_H
8:
9: class Process
10: {
11: public:
12: Process();
13: ~Process();
14: void Create();
15: int WaitForChild();
16: int RunProgram(char * program);
17: bool isParent()
18: { return (pid != 0); }
19: bool isChild()
20: { return (pid == 0); }
21: private:
22: Process & operator= (const Process &); // no permitir la copia
23: int pid;
24: int wait_status;
25: int exec_status;
26: };
27:
28: #endif

```

**ANÁLISIS**

Las funciones miembro agregadas permiten que un proceso padre espere a un proceso hijo mediante la llamada a la función miembro `Process::WaitForChild()`, en la línea 15. El proceso hijo puede cubrirse a sí mismo con la imagen de otro programa usando la función miembro `Process::RunProgram()` que se muestra en el listado 24.3.

**ENTRADA****LISTADO 24.3** Código de ejemplo usando funciones miembro para control de procesos

```

1: // Listado 24.3 Ejemplo de control de procesos
2:
3: #include "lst24-02.h" //#include "process1.h"
4:
5: int main(int argc, char** argv)
6: {
7: Process p;
8: char * program = "ls";
9:
10: p.Create();
11: if (p.isParent())
12: p.WaitForChild();
13: else if (p.isChild())

```

**LISTADO 24.3** CONTINUACIÓN

```
14: p.RunProgram(program);  
15: return 0;  
16: }
```

**ANÁLISIS**

En el listado 24.3, la línea 10 crea un proceso hijo. La línea 11 determina si el proceso que se encuentra actualmente en ejecución es el padre o el hijo. Si el proceso es el padre, espera; de no ser así, se ejecuta el comando ls y el programa termina.

**ptrace()**

ptrace() es una llamada de sistema que permite que un proceso padre controle a un proceso hijo que esté en ejecución, leyendo o escribiendo en la memoria del proceso hijo. ptrace() se utiliza principalmente para depurar un proceso hijo. La interfaz principal primitiva de ptrace() son las señales.

Una *señal* es una interrupción de software. Por lo general, las señales se usan para terminar un proceso, detener la ejecución del proceso o invocar alguna otra acción.

Existen dos tipos de señales, síncronas y asíncronas. Las señales síncronas se producen por la ejecución de una instrucción que ocasiona un problema, por ejemplo, desreferenciar un apuntador no inicializado. Una señal asíncrona se entrega a un proceso sin importar la instrucción que esté en ejecución en ese momento.

gdb es un ejemplo de un proceso que usa ptrace() para controlar un proceso hijo para su depuración. En este caso, gdb es el proceso padre, y el programa que se está depurando es el proceso hijo. El proceso hijo se ejecuta hasta que ocurre una señal. Cuando esto pasa, el control se regresa a gdb. Éste bloquea al hijo por medio de la llamada de sistema wait(). Cuando gdb retoma el control, realiza la operación en el hijo por medio de ptrace().

El uso de ptrace() ha sido sustituido casi totalmente por el sistema de archivos /proc.

## El sistema de archivos /proc

Linux proporciona un sistema de archivos /proc, el cual permite a los usuarios ver información sobre el sistema y los procesos individuales en ejecución. /proc no es un sistema de archivos convencional, sino una estructura en la memoria mantenida por el kernel que proporciona estadísticas del sistema. Un usuario o un proceso pueden obtener información sobre el sistema y los procesos mediante la emisión de llamadas de lectura y escritura al sistema de archivos /proc.

/proc se representa como un directorio de archivos jerárquico. El directorio /proc contiene una entrada de archivo para cada proceso en ejecución, y cada proceso tiene su propio subdirectorío nombrado por su ID de proceso. Además de los procesos individuales, /proc contiene parámetros del sistema e información de los recursos. Un programa o comando del sistema puede leer los directorios y archivos correspondientes a un proceso para ver información del kernel.

Puede consultar a /proc en cualquier momento para obtener información acerca del sistema,



El formato de los datos presentados en el sistema de archivos /proc puede cambiar para distintas versiones del kernel. Es muy recomendable que se familiarice con las estructuras de datos del kernel antes de leer acerca del sistema de archivos /proc.

## Estado y prioridad del proceso

Antes de hablar sobre la programación de procesos, necesita tener una comprensión básica del estado y de la prioridad de un proceso.

Durante el tiempo de vida de un proceso, éste pasa por muchas transiciones de estado. El *estado del proceso* es el estado, o modo, del proceso en un tiempo determinado. Los estados del proceso se pueden encontrar en el archivo de encabezado del kernel de Linux llamado *sched.h* y se definen a continuación.

- **TASK\_RUNNING**—El proceso está esperando ser ejecutado.
- **TASK\_INTERRUPTIBLE** —La tarea está en ejecución y puede ser interrumpida.
- **TASK\_UNINTERRUPTIBLE**—La tarea está en ejecución y no puede ser interrumpida.
- **TASK\_ZOMBIE**—La tarea se ha detenido, pero el sistema aún considera que está en ejecución.
- **TASK\_STOPPED**—El proceso se ha detenido, por lo general, después de haber recibido una señal.
- **TASK\_SWAPPING**—El núcleo está intercambiando información de la tarea entre la memoria y la partición de intercambio.

La prioridad del proceso es un entero asignado a un proceso o subproceso en el que su valor se encuentra en un rango definido por la directiva de programación de tareas. Cuando se crea un proceso, se asigna un valor de prioridad estático. El valor de la prioridad del proceso es la cantidad de tiempo (en “jiffies”) que el programador de tareas da a este proceso para que se ejecute cuando se le permita hacerlo. Un “jiffy” es la duración de un pulso de reloj en el sistema. La cantidad de tiempo en la que se permite que se ejecute un proceso se conoce como *cuanto*.

Existen dos tipos de procesos en Linux, normales y de tiempo real. Los *procesos normales* tienen una prioridad de 0. Un proceso de tiempo real tiene prioridades en el rango de 1-99. Los procesos de tiempo real están programados para tener una prioridad mayor que cualquiera de los demás procesos del sistema. La prioridad de un proceso sólo puede alterarse mediante el uso de llamadas de sistema, como *nice()*. Si existe un proceso en tiempo real listo para ejecutarse, siempre se ejecutará primero. La llamada a *nice()* cambia la prioridad de un proceso. *nice()* toma como argumento un entero no negativo que se suma a la prioridad actual del proceso.

## Algoritmos de administración de procesos

El recurso máspreciado de un sistema operativo es el tiempo de la CPU. Para utilizar eficientemente el tiempo de la CPU, los sistemas operativos utilizan un concepto conocido como *multiprocesamiento*. Este concepto le da al usuario la impresión de que se están ejecutando muchos procesos en forma simultánea. El objetivo del multiprocesamiento es tener un proceso en ejecución en todo momento para que la CPU siempre esté ocupada. Esto garantiza que la CPU se esté utilizando de la manera más eficiente.

La implementación del multiprocesamiento de un sistema operativo consiste en dividir el tiempo de la CPU entre muchos procesos. El *programador de tareas* (scheduler) es el proceso del kernel que determina cuál proceso se ejecutará a continuación. El programador de tareas decide cuál proceso ejecutar, de todos los que se encuentran en el estado `TASK_RUNNING`. El criterio, o las reglas, que determinan cuál proceso se va a ejecutar a continuación se conoce como *algoritmo de administración*.

El programador de tareas de Linux utiliza la prioridad del proceso y la directiva del proceso para determinar cuál proceso se va a ejecutar a continuación.

Existen tres tipos de directivas de proceso: otros, FIFO (Primero en Entrar, Primero en Salir), y RR (por petición). Un proceso normal tiene un tipo de directiva: otros. Los procesos de tiempo real tienen dos tipos de directivas: RR y FIFO.

La directiva otros se implementa como un algoritmo estándar de programación de tareas de tiempo compartido. Ésa es una directiva en la que a cada proceso se le da un *cuanto* igual.

La directiva *FIFO* programa cada proceso ejecutable en el orden en que se encuentra en la cola de ejecución, y ese orden nunca se cambia. Un proceso FIFO se ejecutará hasta que se bloquee en la E/S o se intercambie por un proceso con mayor prioridad.

La programación *RR* ejecuta los procesos de tiempo real uno por uno. La diferencia entre un proceso FIFO y un proceso RR es que éste se ejecutará durante un tiempo especificado (*cuanto*) y luego se intercambiará y se colocará al final de la lista de prioridad.



### Tip

Un usuario debe tener privilegios de superusuario o administrador (root) para ejecutar procesos en tiempo real.

#### DEBE

DEBE ejecutar los procesos en tiempo real como superusuario.

#### NO DEBE

NO DEBE olvidar que necesita comprender muy bien las estructuras de datos del kernel al leer acerca del sistema de archivos /proc.

## Subprocesos

Un *subproceso* se define como una secuencia de instrucciones ejecutadas dentro del contexto de un proceso. Los subprocesos permiten que un proceso realice múltiples operaciones en paralelo. Esto se logra cuando una aplicación se diseña para utilizar subprocesos múltiples.

Los subprocesos reducen la sobrecarga al compartir segmentos fundamentales de un proceso. Recuerde que el entorno de un proceso consiste en la pila, los datos y el código.

La biblioteca de subprocesos POSIX, conocida como LinuxThreads, es una implementación del paquete de subprocesos POSIX 1003.1c para Linux. POSIX 1003.1c es una API para programación de subprocesos múltiples estandarizada por el IEEE como parte de los estándares POSIX. La mayoría de los fabricantes de UNIX ha patrocinado el estándar POSIX 1003.1c. Los subprocesos POSIX se conocen comúnmente como pthreads. En la lección de hoy utilizaremos los términos pthreads y LinuxThreads como uno solo. LinuxThreads se ejecuta en cualquier sistema Linux que cuente con el núcleo 2.0.0 o más reciente, y con la biblioteca glibc 2.

24

### Nota

Para compilar con pthreads, deber incluir el archivo de encabezado pthread, #include <pthread.h>, y debe enlazar la biblioteca pthread. Por ejemplo:  
g++ hola.cxx -o hola -lpthreads.

La API de subprocesos es la misma que para pthreads, pero la implementación es distinta de la de otros sistemas operativos. Otros sistemas operativos de subprocesamiento múltiple, como Windows 2000, definen un subproceso como algo separado de un proceso. LinuxThreads define un subproceso como un contexto de ejecución.

En otros sistemas operativos, los subprocesos se definen como el contexto de la CPU, mientras que el proceso posee la pila, los datos y el heap. Cuando se crea un nuevo proceso, el subproceso se crea como parte del contexto del proceso. Esta implementación de los subprocesos ofrece la ventaja de tener un cambio de contexto muy eficiente. Sin embargo, esta definición de subproceso no encaja con el módulo de procesos de Linux. Por ejemplo, si un proceso posee los recursos de la pila, los datos y el heap, ¿cómo se implementaría la llamada de sistema `fork()` cuando se llame desde un subproceso? LinuxThreads tiene la ventaja de un cambio rápido de contexto de un subproceso tradicional además de que puede definir cuáles recursos del proceso se pueden compartir.

Los subprocesos se implementan mediante la llamada de sistema `_clone`. `_clone()` toma como argumento indicadores que especifican cuál sección se va a compartir entre el proceso y el subproceso. Los recursos que se pueden compartir se definen en el archivo de encabezado del kernel de Linux llamado `schedbits.h`, y se muestran a continuación:

- `CLONE_VM`—Compartir los datos y la pila
- `CLONE_FS`—Compartir el sistema de archivos

- **CLONE\_FILES**—Compartir archivos abiertos
- **CLONE\_SIGHAND**—Compartir señales
- **CLONE\_PID**—Compartir PID del padre (no está completamente implementado)

## Subprocesamiento simple

El *subprocesamiento simple* se define como un solo contexto de ejecución. Cuando un proceso se crea con `fork()`, tiene un solo contexto de ejecución.

## Subprocesamiento múltiple

El *subprocesamiento múltiple* se define como contextos múltiples de ejecución. El subprocesamiento múltiple separa un proceso en muchos contextos de ejecución de subprocesos, por consecuencia permite que se distribuya la ejecución del proceso entre muchos subprocesos.

La *reentrancia* es una característica de la programación de subprocesos múltiples en donde varios contextos de ejecución pueden tener acceso a los datos o recursos compartidos, y se garantiza que se mantiene la integridad de los recursos compartidos. Cuando varios subprocesos comparten recursos, estos recursos deben ser protegidos mediante primitivas de sincronización de subprocesos. La sincronización se describe en la siguiente sección.

## Creación y terminación de subprocessos

Un subprocesso se crea por medio de la llamada de sistema `pthread_create()` y se destruye usando la llamada de sistema `pthread_exit()`. Estas llamadas de sistema están encapsuladas en la clase `Thread`, que se define en el listado 24.4.

---

**ENTRADA LISTADO 24.4** Objeto Thread

```
1: // Listado 24.4 Clase Thread (tcreate.h)
2:
3: #ifndef C_TC CREATE_H
4: #define C_TC CREATE_H
5:
6: #include <pthread.h>
7:
8: class Thread
9: {
10: public:
11:     Thread(void * r, void * a); // ctor/dtor
12:     virtual ~Thread();
13:     int Create(); // crea el subprocesso
14:     int Destroy(); // sale del subprocesso
15: private:
16:     Thread & operator= (const Thread &); // no permitir la copia
17:     pthread_t thread;
```

```

18: pthread_attr_t attr;
19: void * fn;
20: void * fn_args;
21: bool init;
22: };
23:
24: #endif

```

**ANÁLISIS**

El código del listado 24.4 presenta un objeto `Thread` que encapsula llamadas de LinuxThreads básicas para crear y destruir un subprocesso. El constructor de la línea 11 toma como argumentos un apuntador a una función y un argumento que se va a pasar a la función. Un `Thread` se crea mediante la llamada a la función miembro `Thread::Create`. `Create` llama a `pthread_create` usando los parámetros que se pasan al constructor de `Thread`. La función miembro `Thread::Destroy` llama a `pthread_exit` para terminar el subprocesso. El listado 24.5 muestra un programa de ejemplo que utiliza la clase `Thread`.

24

**ENTRADA****LISTADO 24.5** Ejemplo de productor/consumidor

```

1: // Listado 24.5 Ejemplo de productor/consumidor
2:
3: #include <iostream.h>
4: #include "lst24-04.h" // #include "tcreate.h"
5:
6: int data = 0;
7:
8: void read_thread(void * param)
9: {
10: while (1)
11: cout << "leer: " << data << endl;
12: }
13:
14: void write_thread(void * param)
15: {
16: while (1)
17: cout << "escribir: " << data++ << endl;
18: }
19:
20: int main(int argc, char** argv)
21: {
22: Thread thread1((void*)&write_thread, NULL);
23: Thread thread2((void*)&read_thread, NULL);
24:
25: thread1.Create();
26: thread2.Create();
27: for (int i = 0; i < 10000; i++)
28: ;
29: thread1.Destroy();
30: thread2.Destroy();
31: }

```

**ANÁLISIS** El listado 24.5 demuestra el objeto `Thread` usando el clásico ejemplo productor/consumidor en acción. Se crean los objetos `thread1` y `thread2` de la clase `Thread`. `thread1` utiliza la función `write_thread` definida en las líneas 14 a 18; éste es el productor. `thread2` utiliza la función `read_thread` definida en las líneas 8 a 12; éste es el consumidor. Los subprocessos empiezan su ejecución cuando se hace la llamada a las funciones miembro `Thread::Create()`. `read_thread`, el consumidor, lee de los datos de las variables globales, y `write_thread`, el productor, escribe en los datos de las variables globales.



### Tip

Al igual que con los procesos, nunca hay garantía de cuál subprocesso se ejecutará primero, `thread1` o `thread2`.

¿Puede ver el desastre potencial en este ejemplo? Tanto `read_thread` como `write_thread` tienen acceso a los datos globales en cualquier momento. Éste es un problema potencial, ya que la función `read_thread` podría empezar a ejecutarse y quedar suspendida, la función `write_thread` podría empezar a ejecutarse, modificar los datos globales y luego quedar suspendida, y entonces `read_thread` continuaría su ejecución con un valor distinto para los datos globales. Éste es un ejemplo de código que NO es reentrant.

## Administración

Los conceptos de administración de subprocessos son exactamente los mismos que los conceptos de administración de procesos. De hecho, en Linux se utiliza el mismo programador de tareas para los subprocessos y los procesos. Para obtener más detalles, vea la sección “Algoritmos de administración de procesos” que se trató anteriormente en esta lección.

## Sincronización

La sincronización de subprocessos es una técnica para garantizar la integridad de los recursos compartidos entre subprocessos. LinuxThreads proporciona las siguientes primitivas de sincronización: los mutex, los semáforos, las variables de condición y la unión.

### Los mutex

Un *mutex* es una primitiva de sincronización que se utiliza para garantizar el acceso exclusivo mutuo a un recurso, por lo general a datos.

Un mutex se encuentra en uno de dos estados: bloqueado o desbloqueado. Sólo un subprocesso puede bloquear un mutex en un momento dado. Si otro subprocesso intenta adquirir un mutex que ya esté ocupado por otro subprocesso, el bloqueo falla.

El paradigma bloquear/desbloquear se puede implementar mediante primitivas de sincronización múltiples; por lo tanto, definiremos una clase base virtual para esta interfaz. El listado 24.6 define una clase que representa los bloqueos síncronos.

**ENTRADA LISTADO 24.6 Clase para el bloqueo síncrono**

```

1: // Listado 24.6 Clase virtual pura SyncLock (synclock.h)
2:
3: #ifndef C_SYNCLOCK_H
4: #define C_SYNCLOCK_H
5:
6: class SyncLock
7: {
8: public:
9: SyncLock() {} ;
10: ~SyncLock() {};
11: virtual int Acquire() = 0;
12: virtual int Release() = 0;
13: private:
14: SyncLock & operator= (const SyncLock &); // no permitir la copia
15: };
16:
17: #endif

```

**ANÁLISIS**

La clase base virtual SyncLock define la interfaz para una primitiva de sincronización que bloquea un recurso para acceso exclusivo mutuo y luego desbloquea ese recurso, permitiendo que otro subproceso obtenga el bloqueo.

Dada la definición de SyncLock, el listado 24.7 muestra la implementación de un mutex usando la interfaz SyncLock.

**ENTRADA LISTADO 24.7 Clase Mutex**

```

1: // Listado 24.7 Clase Mutex (mutex.h)
2:
3: #ifndef C_MUTEX_H
4: #define C_MUTEX_H
5:
6: #include <pthread.h>
7: #include "lst24-06.h" // #include "synclock.h"
8:
9: class Mutex: public SyncLock
10: {
11: public:
12: Mutex(); // ctor/dtor
13: ~Mutex();
14: int Create();
15: int Destroy();
16: int Acquire(); // obtener el bloqueo

```

*continúa*

**LISTADO 24.7** CONTINUACIÓN

```
17: int Release(); // liberar el bloqueo
18: private:
19: Mutex & operator= (const Mutex &); // no permitir la copia
20: bool init;
21: pthread_mutex_t mutex;
22: pthread_mutexattr_t attr;
23: };
24:
25: #endif
```

**ANÁLISIS** Esta clase mutex tiene cuatro funciones miembro principales, `Mutex::Create()`, `Mutex::Destroy()`, `Mutex::Acquire()` y `Mutex::Release()`. Después de que se crea y se inicializa Mutex, un subproceso la adquiere para obtener acceso exclusivo a un recurso y libera la Mutex al terminar.

Usando el listado 24.8, volvamos a analizar el ejemplo productor/consumidor del listado 24.5 de la sección anterior, usando los mutex.

**ENTRADA** **LISTADO 24.8** Ejemplo de productor/consumidor usando a Mutex

```
1: // Listado 24.8 Ejemplo de productor/consumidor con Mutex
2:
3: #include <iostream.h>
4: #include "lst24-04.h" // #include "tcreate.h"
5: #include "lst24-07.h" // #include "mutex.h"
6:
7: int data;
8:
9: void read_thread(void * param)
10: {
11: Mutex * apMutex = static_cast< Mutex * >(param);
12:
13: while (1)
14: {
15: apMutex->Acquire();
16: cout << "leer: " << data << endl;
17: apMutex->Release();
18: }
19: }
20:
21: void write_thread(void * param)
```

```
22: {
23:     Mutex * apMutex = static_cast< Mutex *>(param);
24:
25:     while(1)
26:     {
27:         apMutex->Acquire();
28:         cout << "escribir: " << data++ << endl;
29:         apMutex->Release();
30:     }
31: }
32:
33: int main(int argc, char** argv)
34: {
35:     Mutex lock;
36:     Thread thread1((void*)&write_thread, &lock);
37:     Thread thread2((void*)&read_thread, &lock);
38:
39:     lock.Create();
40:     thread1.Create();
41:     thread2.Create();
42:     for (int i = 0; i < 100000; i++)
43:     ;
44:     lock.Destroy();
45:     thread1.Destroy();
46:     thread2.Destroy();
47:     return 0;
48: }
```

**24****ANÁLISIS**

El listado 24.8 muestra una vez más el uso de los dos subprocesos `read_thread` y `write_thread`. Sin embargo, los subprocesos reciben un parámetro, un mutex que permite que cada subproceso tenga acceso en forma segura a los datos globales. Ahora, tanto `read_thread` como `write_thread` intentan adquirir el mutex compartido por medio de la función miembro `Mutex::Acquire()`. Cuando se completa el acceso de los subprocesos, el mutex se libera usando la función miembro `Mutex::Release()`.

Éste es un ejemplo de código reentrant.

## Semáforos

Un *semáforo* es un contador para un recurso compartido entre subprocesos. Los semáforos son contadores a los que se debe tener acceso en forma atómica. La palabra *atómica* significa que cuando un subproceso está modificando el valor del semáforo, otro subproceso no puede modificar simultáneamente ese valor.

Conceptualmente, existen dos tipos básicos de semáforos: los *semáforos binarios* y los *semáforos de conteo*. Un semáforo binario nunca toma valores distintos de cero o de uno, y es lógicamente igual que un mutex. Un semáforo de conteo puede tomar valores no negativos arbitrarios.

Se podría utilizar un semáforo binario para implementar un objeto SyncLock definido en la sección anterior. La interfaz del semáforo se define en el archivo de encabezado `semaphore.h`. A continuación se muestra un breve listado de la interfaz.

```
extern int sem_init(sem_t * __sem, int __pshared, unsigned int __value)
    __THROW;

extern int sem_wait(sem_t * __sem) __THROW;

extern int sem_trywait(sem_t * __sem) __THROW;

extern int sem_post(sem_t * __sem) __THROW;

extern int sem_getvalue(sem_t * __sem, int * __sval) __THROW;

extern int sem_destroy(sem_t * __sem) __THROW;
```

## Variables de condición

Una *variable de condición* es una primitiva de sincronización que permite que un subproceso espere a que ocurra algún evento mientras permite que otro subproceso notifique a este subproceso cuando ocurra una condición.

El paradigma esperar/notificar se puede implementar por medio de primitivas de sincronización de LinuxThreads; por ejemplo, en el listado 24.9 declaramos una clase base virtual para esta interfaz.

---

### ENTRADA LISTADO 24.9 Definición del objeto Event

---

```
1: // Listado 24.9 Clase virtual pura Event (event.h)
2:
3: #ifndef C_EVENT_H
4: #define C_EVENT_H
5:
6: class Event
7: {
8: public:
9: Event() {};
10: virtual ~Event() {};
11: virtual int Wait() = 0;
12: virtual int Signal() = 0;
13: };
14:
15: #endif
```

---

**ANÁLISIS**

La clase virtual pura Event definida en el listado 24.10 contiene una interfaz para un objeto que, o espera un evento, o lo señala.

**ENTRADA****LISTADO 24.10** Objeto Variable de condición

```
1: // Listado 24.10 Clase variable de condición (cond.h)
2:
3: #ifndef C_COND_H
4: #define C_COND_H
5:
6: #include <pthread.h>
7: #include "lst24-09.h" //1#include "event.h"
8:
9: class CondVar : public Event
10: {
11: public:
12: CondVar();
13: virtual ~CondVar();
14: void Create();
15: void Destroy();
16: int Wait();
17: int Signal();
18: private:
19: CondVar & operator= (const CondVar &); // no permitir la copia
20: pthread_cond_t cond;
21: pthread_condattr_t attr;
22: pthread_mutex_t mutex;
23: pthread_mutexattr_t mattr;
24: };
25:
26: #endif
```

**24****ANÁLISIS**

El listado 24.10 define el objeto CondVar usando la interfaz Event definida anteriormente. Un subproceso utilizaría la variable CondVar como primitiva de sincronización en un esquema similar al de Mutex. Cuando el subproceso tuviera acceso a los datos, llamaría a la función miembro CondVar::Wait(). Al completar su actualización, liberaría al objeto CondVar mediante una llamada a CondVar::Signal().

**Tip**

Es importante que la variable de condición se inicialice como señalada, o se tendría una condición de punto muerto. Esta implementación de CondVar siempre inicializa la variable de condición como señalada.

## La unión

La *unión* es una primitiva de sincronización que ocasiona que un subprocesso espere que otro subprocesso se complete. La unión no sería considerada como primitiva de sincronización, ya que su propósito es esperar que otro subprocesso termine y no que se ejecute simultáneamente.

## Punto muerto

Cuando está programando en un entorno de subprocessamiento múltiple, varios subprocessos compiten por los recursos limitados. Si un subprocesso pide un recurso que no está disponible, ese subprocesso entra en un estado de espera.

Un subprocesso podría estar esperando en forma indefinida, ya que otro subprocesso en espera está ocupando dicho recurso; esto se conoce como punto muerto. El *punto muerto* se define como un subprocesso bloqueado indefinidamente que compite con otro subprocesso por el mismo conjunto limitado de recursos.

Deben existir cuatro condiciones para caracterizar una condición de punto muerto:

1. *Exclusión mutua*—Por lo menos un bloqueo no es compatible.
2. *Ocupar y esperar*—Un subprocesso está ocupando un recurso y esperando por un recurso que otro subprocesso está ocupando.
3. *No preferencia*—Un recurso ocupado sólo puede ser liberado por el subprocesso que lo posee.
4. *Espera circular*—Debe existir un conjunto de subprocessos en espera,  $\{t_0, t_1, t_2, \dots, t(n)\}$  en donde  $t_0$  está esperando un bloqueo ocupado por  $t_1$ ,  $t_1$  está esperando un bloqueo ocupado por  $t_2$ ,  $\dots$ ,  $t(n-1)$  y  $t(n)$  está esperando un bloqueo ocupado por  $t_0$ .

El *punto muerto recursivo* es una condición en la cual un subprocesso intenta adquirir un recurso que ya posee.

El *punto muerto mutuo* ocurre cuando dos subprocessos adquieren bloqueos separados y están bloqueados, cada uno esperando que el otro libere el bloqueo. Esto también se define como *abrazo mortal*.

| DEBE                                                                                            | NO DEBE                                                            |
|-------------------------------------------------------------------------------------------------|--------------------------------------------------------------------|
| DEBE proteger los datos compartidos entre varios subprocessos para que su código sea reentrant. | NO DEBE olvidar liberar un mutex después de que ha sido adquirido. |
| DEBE inicializar adecuadamente las primitivas de sincronización de LinuxThreads.                |                                                                    |

## Resumen

La biblioteca LinuxThreads no implementa completamente el estándar POSIX 1003.1c, pero está casi completo. La principal característica que no se apega a este estándar es el uso de señales.

24

En la primera mitad de la lección de hoy cubrimos los procesos y las técnicas de control de procesos, así como algunas clases básicas de C++ que utilizan las llamadas de sistema para procesos descritas.

La segunda mitad de la lección de hoy trató sobre la creación y la destrucción de los subprocessos y sobre las primitivas de sincronización de subprocessos.

## Preguntas y respuestas

- P** ¿Cómo se podría cambiar la adquisición de un mutex para evitar un punto muerto?
- R** Al adquirir un mutex, utilice un valor de interrupción. Si no se puede adquirir el mutex, la llamada de interrupción permitirá que el proceso o subprocesso continúe.
- P** ¿Cómo se podría evitar un abrazo mortal?
- R** Al adquirir varios mutex, siempre adquiéralos en el mismo orden y libérelos en el orden inverso al de la adquisición.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, "Respuestas a los cuestionarios y ejercicios", y asegúrese de comprender las respuestas antes de pasar al siguiente día.

## Cuestionario

1. Enliste y defina los estados de un proceso.
2. Describa la diferencia entre la directiva de programación FIFO y la directiva de programación RR.
3. ¿Cuál es la diferencia entre un semáforo binario y un semáforo de conteo?
4. Enliste y defina los cuatro requisitos para que se produzca un punto muerto.

## Ejercicios

1. Usando un semáforo de conteo, ¿cómo podría resolver la “condición de carrera” al iniciar subprocessos?
2. Implemente el ejemplo reentrantre del listado 24.8 usando el objeto variable de condición CondVar.

# SEMANA 4

DÍA **25**

## Comunicación entre procesos

La comunicación entre procesos (IPC) es el medio por el cual dos procesos se comunican entre sí. La lección de hoy trata sobre algunos de los métodos disponibles para que los procesos se comuniquen entre sí.

Linux ofrece numerosos métodos para que los procesos de un mismo sistema se comuniquen entre sí. Los métodos incluyen, pero no están limitados a, tuberías, tuberías con nombre y la comunicación entre procesos de System V. Linux también proporciona otros métodos para que los procesos que se encuentran en sistemas separados se comuniquen entre sí, como los sockets, por ejemplo. Esta lección se enfoca en la comunicación entre procesos de un mismo sistema.

### Antecedentes

Antes de hablar sobre el tema de hoy, empezaremos con la definición de una interfaz que se utilizará para todas las clases de hoy. La clase Object definida en el listado 25.1 implementa una interfaz común para la creación y destrucción de objetos.

Para poder implementar una interfaz común para la creación y destrucción de objetos, se define una interfaz virtual pura llamada `Object`. Todos los objetos IPC definidos en la lección de hoy se derivan de la clase `Object`.

Cada uno de nuestros objetos hereda esta interfaz para su creación y su destrucción. Esta interfaz se define en `object.h` en el código de muestra del listado 25.1.

---

**ENTRADA LISTADO 25.1 Interfaz Object**

---

```
1: // Listado 25.1 Clase Object (object.h)
2:
3: #ifndef C_OBJECT_H
4: #define C_OBJECT_H
5:
6: class Object
7: {
8: public:
9:     virtual int Create() = 0;
10:    virtual int Destroy() = 0;
11: };
12:
13: #endif
```

---

Además, todos nuestros métodos de IPC utilizan estructuras de datos, recursos y manejadores del kernel. Como tales, no es aplicable copiar estos objetos. Por ejemplo, si un usuario copia una tubería, se cierra una instancia del objeto tubería, y por consecuencia el manejador de la otra instancia de la tubería ya no es válido. Por esta razón, todos los objetos de la lección de hoy definen el constructor de copia como una función miembro privada. Además, dicho constructor no se implementa. Al no implementarlo, otras funciones miembro no pueden llamarlo sin querer; cualquier llamada al constructor de copia producirá un error de enlace.

## Tuberías

Una *tubería* es un método simple que permite que un proceso agregue su entrada estándar a la salida estándar de otro proceso. Una tubería es un canal de comunicación semidúplex entre un proceso padre y un proceso hijo.

Linux proporciona la llamada de sistema conocida como `pipe()`. Esta llamada regresa dos descriptores de archivos como un arreglo de descriptores de archivos; el descriptor de archivo 0 es el extremo de lectura de la tubería, y el descriptor de archivo 1 es el extremo de escritura. El prototipo de la función `pipe()` se muestra a continuación:

```
int pipe(int fd[2]);
```

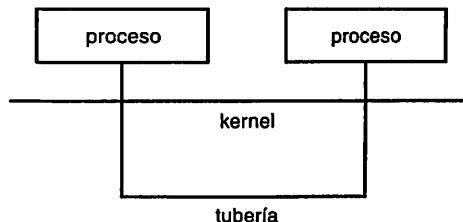
Un *descriptor de archivos* es un entero utilizado para referirse a un archivo abierto por el kernel. Cuando un proceso necesita leer o escribir en un archivo, el archivo se identifica por el descriptor de archivo.

### Nota

Como vio en el día 24, "Programación de sistemas", un proceso hijo hereda los archivos abiertos de los procesos padres (es decir, se copian los manejadores de archivo al proceso hijo). Esto incluye los manejadores utilizados para las tuberías.

El uso típico de una tubería es cuando un proceso padre crea una tubería y llama a `fork()` para crear un proceso hijo. Como un proceso hijo hereda todos los descriptores de archivo de los procesos padres, tanto el padre como el hijo ahora contienen descriptores de archivos que se refieren uno a otro. Para completar la configuración de una tubería semidúplex para que el hijo pueda escribir al padre, el proceso hijo cierra `fd[0]`, el descriptor de archivo de lectura, y el padre cierra `fd[1]`, el descriptor de archivo de escritura. Lo que queda es el descriptor de archivo de escritura para el hijo y el descriptor de archivo de lectura para el padre. Esta secuencia de eventos crea una tubería semidúplex para que el hijo escriba al padre. La figura 25.1 muestra un diagrama de una tubería entre dos procesos.

**FIGURA 25.1**  
Tubería entre dos procesos.



25

Después de que se crea la tubería y se ha completado la configuración necesaria, la comunicación se logra mediante el uso de llamadas estándar de lectura y de escritura.

De manera predeterminada, las tuberías utilizan bloqueo de E/S. Es decir, si una tubería realiza una lectura y no hay datos que leer, la tubería *se bloquea* hasta que hayan datos disponibles. Si un usuario no quiere bloquear, la tubería se puede abrir con el indicador 0 `NONBLOCK` para deshabilitar el bloqueo de E/S.

El listado 25.2 define la clase `Pipe`. Esta clase encapsula las llamadas de sistema para tuberías.

### ENTRADA LISTADO 25.2 Definición de la clase Pipe

```

1: // Listado 25.2 Clase Pipe (pipe.h)
2:
3: #ifndef C_PIPE_H

```

continúa

**LISTADO 25.2** CONTINUACIÓN

---

```
4: #define C_PIPE_H
5:
6: #include <unistd.h>
7: #include "lst25-01.h" // #include "object.h"
8:
9: class Pipe : public Object
10: {
11: public:
12: Pipe();
13: ~Pipe();
14: int Create();
15: int Destroy();
16: void SetToRead();
17: void SetToWrite();
18: int ReadFromPipe(char *);
19: int WriteToPipe(char *);
20: private:
21: // no permitir la copia
22: Pipe & operator=(const Pipe &);
23: bool init_;
24: bool read_;
25: int pipe_[2];
26: };
27:
28: #endif
```

---

**ANÁLISIS** La clase `Pipe` definida en el listado 25.2 implementa las rutinas descritas en esta lección relacionadas con las tuberías. El padre y el hijo deciden cuál proceso lee y cuál proceso escribe, y cada proceso llama a la función miembro `SetToRead()` o a `SetToWrite()`. Después de que cada proceso inicializa su tubería para leer o escribir, llama a la función miembro `ReadFromPipe()` o a `WriteToPipe()` para transferir datos.

Las tuberías están limitadas a los procesos padre/hijo.

Debe crear y configurar la tubería semidúplex utilizando ambos procesos. Si uno de los dos procesos no está listo, es decir, si un proceso escribe a una tubería que el otro proceso no ha abierto para lectura, o lee de una tubería que el otro proceso no ha abierto para escritura, se regresa la señal `SIGPIPE`.

Las tuberías semidúplex se implementan por medio de los descriptores de archivo del proceso. Como los descriptores de archivo son recursos del proceso, se regresan al sistema cuando el proceso termina.

| DEBE                                                                          | NO DEBE                                                                                                              |
|-------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| DEBE utilizar una tubería para procesos que comparten la relación padre hijo. | NO DEBE escribir en una tubería a menos que ambos procesos hayan configurado sus respectivos extremos de la tubería. |

### popen y pclose

De manera alternativa, Linux proporciona otra manera de crear una tubería semidúplex, mediante `popen()`.

```
FILE * popen (char * comando, char * acceso);
```

La llamada de sistema conocida como `popen()` crea una tubería, realiza una llamada a `fork()`, y luego llama a `exec()` para ejecutar el comando solicitado. El acceso de lectura o de escritura a la tubería creada por `popen()` se determina mediante el argumento `acceso`. El valor de retorno de `popen()` es un flujo `iostream` y se debe cerrar con `pclose()`.

Para cerrar el flujo creado por `popen()`, el usuario debe llamar a `pclose()`. Esta llamada de sistema cierra la tubería y regresa los recursos del proceso al sistema.

```
FILE * popen(ls, "r");
pclose(ls);
```

25

## Tuberías con nombre (FIFOs)

Una *tubería con nombre* es un método que permite que procesos independientes se comuniquen. Las tuberías con nombre se conocen tradicionalmente en el mundo UNIX como FIFOs (Primero en Entrar, Primero en Salir). Una tubería con nombre es similar a una tubería; es una tubería semidúplex entre dos procesos. La diferencia entre una tubería y una tubería con nombre es que cuando los procesos terminan de usar la tubería con nombre, ésta permanece en el sistema.

Otra diferencia entre tuberías y tuberías con nombre es que como una tubería con nombre es parte del sistema de archivos, se debe abrir y cerrar como un archivo.

El procedimiento para utilizar una tubería con nombre es similar al uso de la llamada de sistema `pipe()`. Un proceso abre la tubería con nombre para escribir y otro proceso la abre para leer. Una vez que la tubería con nombre se abre utilizando ambos procesos, éstos se comunican entre sí mediante la tubería con nombre. Puede crear una tubería con nombre de varias maneras. Puede utilizar los comandos de shell `mknod` y `mkfifo`. Además, un proceso puede utilizar la llamada de sistema `mknod()`.

```
int mknod(char *trayectoria, mode_t modo, dev_t dispositivo);
```

**Nota**

Por lo general, el comando de shell `mknod` se reserva para el administrador (root); sin embargo, cualquier usuario puede crear una tubería con nombre por medio de `mknod`.

Una vez que se crea la tubería, los datos se pasan entre los dos procesos usando las llamadas de sistema estándar de lectura y escritura: `fopen()`, `fclose()`, `fread()` y `fwrite()`.

El listado 25.3 muestra nuestra implementación de un objeto que representa a una tubería con nombre.

**ENTRADA LISTADO 25.3 Definición de la clase NamedPipe**

```
1: // Listado 25.3 Clase NamedPipe (npipe.h)
2:
3: #ifndef C_NP_H
4: #define C_NP_H
5:
6: #include <sys/types.h>
7: #include <sys/stat.h>
8: #include <stdio.h>
9: #include "lst25-01.h" // #include "object.h"
10:
11: class NamedPipe : public Object
12: {
13: public:
14: NamedPipe();
15: ~NamedPipe();
16: int Create();
17: int Create(char * name);
18: int Destroy();
19: int Open();
20: int Close();
21: int Read(char * buf, int len);
22: int Write(char * buf, int len);
23: private:
24: // no permitir la copia
25: NamedPipe & operator=(const NamedPipe &);
26: bool init_;
27: FILE * fp_;
28: char * name_;
29: };
30:
31: #endif
```

**ANÁLISIS**

La función `Create()` se sobrecarga para tomar el nombre de una tubería con nombre o para utilizar un valor predeterminado. Después de crear la tubería con nombre, se debe abrir. Después de abrir el objeto `NamedPipe`, se puede leer de él o escribir en él usando las funciones miembro `Read()` y `Write()`.

Al terminar de utilizar el objeto `NamedPipe`, debe llamar a `Close()` para que cierre el manejador de la tubería, y a la función `Destroy()` para limpiar las estructuras de datos del kernel relacionadas con esta conexión.

| <b>DEBE</b>                                                                                                      | <b>NO DEBE</b>                                                                                              |
|------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <code>DEBE</code> utilizar una tubería con nombre para comunicarse entre dos procesos que no estén relacionados. | <code>NO DEBE</code> escribir en una tubería a menos que ambos procesos hayan completado la inicialización. |

## Comunicación entre procesos de System V

El sistema UNIX System V de AT&T presentó tres formas adicionales de comunicación entre procesos (IPC) de un mismo sistema: mensajes, semáforos y memoria compartida.

25

Cada uno de estos métodos de comunicación entre procesos (mensajes, semáforos y memoria compartida) se describe en esta sección, junto con una clase que demuestra su uso.

### Creación de claves

Cada uno de los métodos de IPC mencionados utiliza una clave para identificarse a sí mismo. Antes de profundizar en los detalles de IPC de System V, hablaremos brevemente sobre las claves y desarrollaremos una clase para manejar las claves para los métodos de IPC.

El primer paso para utilizar la comunicación entre procesos de System V es la creación de una clave. Ésta es un entero no negativo. Para que los procesos puedan tener acceso a un método de IPC, el método debe utilizar una clave. La clave es del tipo `key_t`.

Los procesos que utilizan la comunicación entre procesos de System V deben acordar la clave que van a usar. Existen varias formas de que los procesos acuerden el valor de la clave. En esta sección llamaremos cliente y servidor a nuestros procesos.

- Un servidor puede crear una nueva clave especificando `IPC_PRIVATE` como una clave en la llamada de sistema apropiada, ya sea `msgget()`, `semget()` o `smget()`. Después de crear la clave, el servidor debe escribirla en una ubicación en la que el cliente la recuperará, por ejemplo un archivo. `msgget()`, `semget()` y `smget()` son rutinas de IPC de System V que crean instancias de los métodos de IPC.

- El cliente y el servidor pueden acordar una clave y compartir la en un archivo de encabezado común.
- El cliente y el servidor pueden acordar un nombre de ruta y un identificador y realizar la llamada de sistema `f tok()` para generar una clave. `f tok()` es una llamada de sistema que crea una clave con base en el nombre de archivo de entrada y el identificador del proceso.

En el listado 25.4 desarrollamos una clase de C++ que maneja las claves de IPC de System V. Cada uno de los objetos IPC debe tener un objeto clave para funcionar adecuadamente.

---

**ENTRADA LISTADO 25.4 Definición de la clase Key**

---

```
1: // Listado 25.4 Clase Key  (key.h)
2:
3: #ifndef C_KEY_H
4: #define C_KEY_H
5:
6: #include <sys/types.h>
7: #include <sys/ipc.h>
8: #include "lst25-01.h"    // #include "object.h"
9:
10: class Key
11: {
12: public:
13: Key();
14: ~Key();
15: int Create(char * name, char id);
16: int Create(key_t key);
17: void Destroy();
18: key_t Get(void);
19: private:
20: // no permitir la copia
21: Key & operator=(const Key &);
22: bool init_;
23: key_t key_;
24: };
25:
26: #endif
```

---

**ANÁLISIS** El primer paso para obtener una clave es crear un objeto clave y llamar a la función miembro `Create()`. Esta función está sobrecargada, así que puede pasarle una clave predefinida o crear una clave pasándole un nombre de archivo y un identificador. Después de crear la clave, puede obtener su valor llamando a la función miembro `Get()`.

## Estructura de permisos de IPC

Cada método para IPC contiene una estructura `ipc_perm`. Esta estructura contiene los permisos y el propietario del objeto IPC. El listado 25.5 muestra la estructura `ipc_perm` de Linux.

### ENTRADA LISTADO 25.5 Estructura ipc\_perm de Linux

```

1: // Listado 24.5 Estructura de permisos de IPC
2:
3: struct ipc_perm
4: {
5:     __kernel_key_t key;
6:     __kernel_uid_t uid;
7:     __kernel_gid_t gid;
8:     __kernel_uid_t cuid;
9:     __kernel_gid_t cgid;
10:    __kernel_mode_t mode;
11:    unsigned short seq;
12: };

```

**ANÁLISIS** Cuando se crea un método de IPC, la estructura `ipc_perm` se llena automáticamente. Durante la vida de un objeto IPC, el usuario puede modificar los campos `uid`, `gid` y `mode` mediante las funciones `msgctl()`, `semctl()` o `shmctl()`. Para cambiar estos campos, el usuario debió ser el creador del objeto IPC, o el superusuario. Cada una de las llamadas al sistema de IPC que crean un método de IPC toma un argumento de indicadores que define los permisos para ese objeto. Esos indicadores se utilizan para llenar la estructura `ipc_perm`.

25

## Comandos ipcs e ipcrm

Los métodos de IPC de System V se implementan en el kernel. Si un proceso termina sin limpiar sus métodos de IPC, los métodos permanecerán en el kernel. Los comandos `ipcs` e `ipcrm` se utilizan para mostrar el estado de los métodos de IPC y para limpiar cualquier recurso de IPC restante.

El comando `ipcs` se utiliza para obtener el estado de todos los objetos IPC. A continuación se muestra el resultado de usar el comando `ipcs` (dos segmentos de memoria compartida en mi sistema, identificadores 78592 y 78593).

```

----- Segmentos memoria compartida -----
key      shmid      propietario     perms      bytes      nattch     estado
0x00bc614e 78592      paul        666      1024          0
0x0000162e 78593      paul        666          64          0

----- Matrices semáforo -----

```

| key                              | semid | proprietario | perms | nsems            | estado   |
|----------------------------------|-------|--------------|-------|------------------|----------|
| <b>--- Colas de mensajes ---</b> |       |              |       |                  |          |
| key                              | msqid | proprietario | perms | bytes utilizados | mensajes |

El comando **ipcrm** se utiliza para quitar del sistema un objeto IPC. Usando el ejemplo de la sección anterior, utilizaremos el comando **iprm** para quitar uno de los segmentos de memoria compartida. A continuación se muestra la remoción de la memoria compartida con el identificador 78593. Para ello se utiliza el comando **ipcrm**, seguido del tipo del recurso, en este caso **shm** (shared memory), y por último, el o los identificadores a eliminar. Para eliminar elementos de las secciones **Matrices semáforo** y **Colas de mensajes** utilice las opciones **sem** y **msg**, respectivamente

```
bash# ipcrm shm 78593
resource deleted
bash#
```

Ahora que hemos quitado la memoria compartida con el identificador 78593, volveremos a ejecutar el comando **ipcs** para verificar que se haya quitado el método de memoria compartida. A continuación se presentan los resultados de volver a ejecutar el comando **ipcs**.

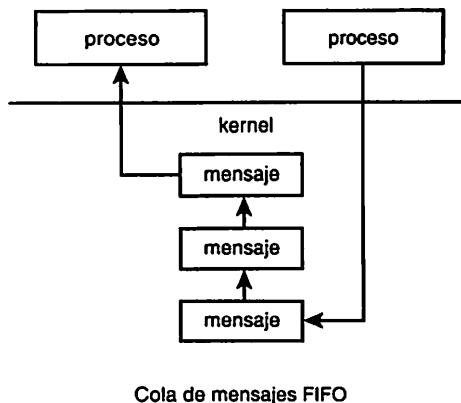
| <b>--- Segmentos memoria compartida ---</b> |       |              |       |                  |          |        |
|---------------------------------------------|-------|--------------|-------|------------------|----------|--------|
| key                                         | shmid | proprietario | perms | bytes            | nattch   | estado |
| 0x0000162e                                  | 78593 | paul         | 666   | 64               | 0        |        |
| <b>--- Matrices semáforo ---</b>            |       |              |       |                  |          |        |
| key                                         | semid | proprietario | perms | nsems            | estado   |        |
| <b>--- Colas de mensajes ---</b>            |       |              |       |                  |          |        |
| key                                         | msqid | proprietario | perms | bytes utilizados | mensajes |        |

Como puede ver, se quitó el segmento de memoria compartida. También se puede observar que no hay colas de mensajes ni semáforos en el sistema.

## Colas de mensajes

Las colas de mensajes son un método para que los procesos se envíen mensajes (datos) entre sí. Una cola de mensajes es una lista enlazada de mensajes mantenida por el kernel. Los mensajes se agregan y se quitan de una cola FIFO por medio de los procesos. Cada mensaje consta de un identificador de mensaje, los datos de mensaje y el número de bytes de los datos del mensaje. La figura 25.2 muestra una cola de mensajes.

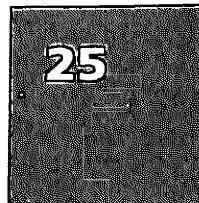
Para cada cola de mensajes, el kernel mantiene la estructura **msqid\_ds** definida en el archivo de encabezado estándar **<linux/msg.h>** del kernel. El listado 25.6 muestra la estructura **msqid\_ds** de Linux. Esta estructura mantiene apunadores a los mensajes, el número de mensajes en la cola y demás información relacionada con la cola de mensajes, como el identificador de proceso del último proceso que envió un mensaje y del último proceso que recibió un mensaje.

**FIGURA 25.2***La cola de mensajes.***ENTRADA****LISTADO 25.6** Estructura msqid\_ds de Linux

```

1: // Listado 25.6 Una estructura msqid
2: // para cada cola en el sistema
3:
4: struct msqid_ds
5: {
6:     struct ipc_perm msg_perm;
7:     struct msg * msg_first;           /* primer mensaje en la cola */
8:     struct msg * msg_last;           /* último mensaje en la cola */
9:     __kernel_time_t msg_stime;       /* última vez msgsnd */
10:    __kernel_time_t msg_rtime;        /* última vez msgrcv */
11:    __kernel_time_t msg_ctime;        /* último cambio */
12:    struct wait_queue * wwait;
13:    struct wait_queue * rwait;
14:    unsigned short msg_cbytes;        /* número actual de bytes en la cola */
15:    unsigned short msg_qnum;          /* número de mensajes en la cola */
16:    unsigned short msg_qbytes;        /* número máximo de bytes en la cola */
17:    __kernel_ipc_pid_t msg_lspid;     /* pid del último msgsnd */
18:    __kernel_ipc_pid_t msg_lrpid;     /* último pid de recepción */
19: };

```

**ANÁLISIS**

Los apunadores a la cola actual de mensajes son los miembros `*first` (primero) y `*last` (último) de la estructura `msqid_ds` que se muestra en el listado 25.6.

Antes de que se pueda utilizar una cola de mensajes, primero se debe crear. La llamada al sistema `msgget()` crea una nueva cola de mensajes. `msgget()` regresa el identificador de la cola de mensajes. Este identificador es un `int`, y se utiliza para las funciones de mensajes restantes. `msgget()` regresa un valor `msgqid` entero, y este valor `msgqid` se pasa a las otras funciones `msg*`.

```

/* Obtener cola de mensajes. */
extern int msgget (__key_t __key, int __msgflg));

```

El parámetro `key` es la clave que se describió en la sección anterior. El parámetro `msgflg` consiste en los permisos de acceso para la cola de mensajes, a los que se les aplica una operación OR a nivel de bits con los siguientes indicadores:

- `IPC_CREAT` Crear la cola si no existe.
- `IPC_EXCL` Si se utiliza con `IPC_CREAT`, fallar si la cola ya existe.

Los mensajes se colocan en la cola de mensajes por medio de la llamada de sistema `msgsnd()`.

```
/* Enviar mensaje a la cola de mensajes. */
extern int msgsnd __P ((int __msqid, void *__msgp, size_t __msgsz,
                        int __msgflg));
```

`msgsnd()` toma como argumento un tipo de estructura `msgbuf`. `msgbuf` se define en `<linux/msg.h>`. El tipo `msgbuf` es una definición de modelo para el mensaje actual que se pasa a `msgsnd()`. El listado 25.7 muestra la estructura `msgbuf` de Linux. El parámetro `size` es el tamaño del mensaje en bytes.

#### ENTRADA

#### LISTADO 25.7 Estructura `msgbuf` de Linux

```
1: // Listado 25.7 Búfer de mensajes para
2: // las llamadas msgsnd y msgrcv
3:
4: struct msgbuf
5: {
6:     long mtype;           /* tipo de mensaje */
7:     char mtext[ 1 ];      /* texto del mensaje */
8: };
```

Por lo general, un programador de IPC redefine el mensaje para que se apegue a la definición estándar de `msgbuf` en `msh.h`. Por ejemplo, un mensaje se podría definir como se muestra en el listado 25.8.

#### ENTRADA

#### LISTADO 25.8 Estructura `msgbuf ipc_message` de usuario de muestra

```
1: // Listado 25.8 Estructura de muestra para mensajes IPC
2:
3: typedef struct ipc_message
4: {
5:     long mtype;
6:     long header;
7:     char payload[ 8 ];
8: } ipc_message;
```

Este mensaje se define como un encabezado de mensaje, `mtype`. El tipo de mensaje debe ser un entero no negativo, un encabezado y una carga útil que sean específicos para la aplicación. El núcleo no modifica los datos del mensaje de ninguna manera. El encabezado

se puede usar como indicador para denotar la forma en que se deben interpretar los datos de la carga útil.

Los mensajes se leen de una cola de mensajes por medio de la llamada de sistema `msgrecv()`. El parámetro `msgtype` le permite aplicar algo de granularidad al leer mensajes. Los valores y las definiciones de `msgtype` se muestran a continuación.

```
/* Recibir mensaje de la cola de mensajes. */
extern int msgrecv __P ((int __msqid, void * __msgp, size_t __msgsz,
                         long int __msgtype, int __msgflg));
```

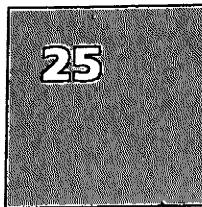
Los valores para `msgflg` se definen de la siguiente manera:

- Si `msgtype` es cero, se regresa el primer mensaje de la cola.
- Si `msgtype` es mayor que cero, se regresa el mensaje que tenga un tipo de mensaje igual.
- Si `msgtype` es menor que cero, se regresa el primer mensaje que tenga el tipo más bajo que sea menor o igual que el valor absoluto del mensaje.

Cuando una aplicación termina de utilizar la cola de mensajes, ésta se debe quitar del sistema. La llamada de sistema `msgctl()` se utiliza para quitar del sistema una cola de mensajes.

```
/* Operación de control de cola de mensajes. */
extern int msgctl __P ((int __msqid, int __cmd, struct msqid_ds *__buf));
```

El listado 25.9 define un objeto `Message`. Este objeto encapsula todas las llamadas de sistema para mensajes que se definen en esta sección.



#### ENTRADA Listado 25.9 Objeto Message

```
1: // Listado 25.9 Clase Message (message.h)
2:
3: #ifndef C_MESSAGE_H
4: #define C_MESSAGE_H
5:
6: #include <sys/types.h>
7: #include <sys/ipc.h>
8: #include <sys/msg.h>
9: #include "lst25-01.h"    // #include "object.h"
10: #include "lst25-04.h"   // #include "key.h"
11:
12: class Message : public Object
13: {
14: public:
15: Message(Key k);
16: ~Message(void);
17: int Create(void);
18: int Create(int flags);
19: int Destroy(void);
20: virtual int Read(char * buf, int len, long type);
21: virtual int Write(char * buf, int len, long type);
```

*continúa*

**LISTADO 25.9 CONTINUACIÓN**

```
22: private:  
23: // no permitir la copia  
24: Message & operator=(const Message &);  
25: typedef struct  
26: {  
27: long type;  
28: char * payload;  
29: } TheMessage;  
30: bool init_;  
31: Key key_;  
32: int msgqid_;  
33: };  
34:  
35: #endif
```

---

**ANÁLISIS** El objeto **Message** que se define en el listado 25.9 contiene las funciones miembro estándar **Create()** y **Destroy()** definidas para todos nuestros objetos. Observe que el constructor toma un objeto **Key**. Los procesos cliente y servidor crean un objeto **Key** ya sea pasando el mismo valor de clave o utilizando el mismo nombre de archivo e identificador; de esta manera crearán un objeto **Message** que haga una referencia a la misma cola de mensajes. Una vez que se ha creado el objeto **Message**, los procesos cliente y servidor envían y reciben mensajes por medio de las funciones miembro **Read()** y **Write()**.

El objeto **Message** encapsula las llamadas de sistema para mensajes que se describen en esta sección. El código de ejemplo para esta sección contiene una muestra del objeto **Message**; el programa **msgtest** utiliza el objeto **Message** para enviar y recibir mensajes.

## Semáforos

Los semáforos son un método utilizado para sincronizar operaciones entre dos procesos. Los semáforos no son un método para pasar datos entre procesos, como los mensajes, sino un medio para que dos procesos sincronicen el acceso a algún recurso compartido.

Para que un proceso obtenga acceso a un recurso, evaluará el valor del semáforo. Si el valor es mayor que cero, el proceso decrementa el valor en uno. Si el valor es cero, el proceso se bloquea hasta que el valor sea mayor que uno. Cuando el proceso termine de acceder al recurso, lo liberará incrementando el valor del semáforo en uno.

**Nota**

Un semáforo cuyos valores se limitan a cero y uno se conoce como *semáforo binario*. Los semáforos de System V no están limitados a cero y uno; pueden tener el valor de cualquier entero no negativo.

Como se describió en la sección anterior, el proceso de adquirir un semáforo requiere de dos pasos: primero se prueba el valor, y luego se decrementa. Para que este método funcione adecuadamente entre varios procesos, las operaciones de prueba y decremento deben ser atómicas. Para que las operaciones de prueba y decremento sean atómicas, la implementación de estas operaciones se hace en el kernel.

El kernel mantiene una estructura `semid_ds` para cada uno de sus semáforos. `semid_ds` se define en `<linux/sem.h>`. El listado 25.10 muestra la estructura `semid_ds` de Linux.

**ENTRADA****LISTADO 25.10** Definición de la estructura `semid_ds` de Linux

```

1: // Listado 25.10 Una estructura de datos semid para
2: //           cada conjunto de semáforos del sistema.
3:
4: struct semid_ds
5: {
6:     struct ipc_perm sem_perm; /* permisos .. ver ipc.h */
7:     __kernel_time_t sem_otime; /* última vez semop */
8:     __kernel_time_t sem_ctime; /* último cambio */
9:     struct sem * sem_base; /* apuntador al primer semáforo del arreglo */
10:    struct sem_queue * sem_pending; /* operaciones pendientes de procesar */
11:    struct sem_queue ** sem_pending_last; /* última operación pendiente */
12:    struct sem_undo * undo; /* solicitudes para deshacer en este arreglo */
13:    unsigned short sem_nsems; /* núm. de semáforos del arreglo */
14: };

```

25

**ANÁLISIS**

Antes de utilizar un semáforo, primero se debe crear. La llamada de sistema `semget()` crea un nuevo semáforo. `semget()` regresa un identificador de semáforo o -1 si no se puede crear un semáforo. El identificador de semáforo es un `int`. Este identificador se utiliza para las funciones de semáforo restantes. El prototipo de la función `semget()` se muestra a continuación:

```
/* Obtener semáforo. */
extern int semget __P ((key_t __key, int __nsems, int __semflg));
```

Observe el valor de `nsems`. `Semget()` puede crear varios semáforos en una sola llamada. Si se utiliza `semget()` para crear varios semáforos, los semáforos se enumeran empezando desde 0.

Después de crear un semáforo con `semget()`, se accede mediante el uso de la llamada de sistema `semop()`.

```
/* Operar sobre el semáforo. */
extern int semop __P ((int __semid, struct sembuf *__sops,
                      unsigned int __nsops));
```

`semop()` toma como argumento una estructura `sembuf`, la cual se define en el listado 25.11.

**ENTRADA**
**LISTADO 25.11** Definición de la estructura `sembuf` de Linux

```
1: // Listado 25.11 Las llamadas de sistema semop
2: //          toman un arreglo de éstos.
3:
4: struct sembuf
5: {
6:     unsigned short sem_num; /* índice de semáforo del arreglo */
7:     short sem_op;           /* operación del semáforo */
8:     short sem_flg;          /* indicadores de operación */
9: };
```

**ANÁLISIS**

`sembuf` utiliza el elemento `sem_num` para denotar el semáforo al que se está teniendo acceso, `sem_op` para la operación, y `sem_flag` para el valor de la operación. Las operaciones válidas se definen en `sem.h`. `semid` se utiliza en caso de que esté tratando con varios semáforos. Por medio de la estructura `sembuf`, puede realizar varias operaciones en una sola llamada a `semop()`.

Por ejemplo, una función miembro que adquiere el semáforo se implementa de esta manera usando las llamadas de sistema `sembuf()` y `semop()`. El listado 25.12 muestra la implementación de la función miembro `Acquire()`, que se lleva a cabo mediante la llamada de sistema `semop()`.

**ENTRADA**
**LISTADO 25.12** Función `Semaphore::Acquire()` que muestra la llamada de sistema `semop()`

```
1:// Listado 25.12 Muestra el uso de la llamada
2://          de sistema semop()
3:
4:int Semaphore::Acquire()
5:{           // probar el semáforo y decrementar
6:    static struct sembuf lock[ 1 ] = { { 0, 1, IPC_NOWAIT } };
7:
8:    int len = sizeof(lock) / sizeof(struct sembuf);
9:    int stat = semop(semid_, lock, len);
10:
11:   return stat;
12:}
```

Los semáforos tienen otra llamada de sistema, `semctl()`. `semctl()` se utiliza para quitar un semáforo del sistema.

```
/* Operación de control de semáforo. */
extern int semctl __P ((int __semid, int __semnum, int __cmd, ...));
```

El listado 25.13 define una clase semáforo que muestra los usos de las llamadas de sistema para semáforos definidas en esta sección.

**ENTRADA LISTADO 25.13 Definición de la clase Semaphore**

```
1: // Listado 25-13 Clase Semaphore (semap.h)
2:
3: #ifndef C_SEMAP_H
4: #define C_SEMAP_H
5:
6: #include <sys/sem.h>
7: #include "lst25-04.h" // #include "key.h"
8: #include "lst25-01.h" // #include "object.h"
9:
10: class Semaphore : public Object
11: {
12: public:
13:     Semaphore(Key);
14:     ~Semaphore();
15:     int Create();
16:     int Create(int flags);
17:     int Destroy();
18:     int QueryValue();
19:     int Acquire();
20:     int Release();
21: private:
22:     // no permitir la copia
23:     Semaphore & operator=(const Semaphore &);
24:     bool init_;
25:     Key key_;
26:     int semid_;
27: };
28:
29:#endif
```

**25****ANÁLISIS**

El objeto `Semaphore` definido en el listado 25.13 contiene las funciones miembro estándar `Create()` y `Destroy()` definidas para todos nuestros objetos. Una vez más, observe que el constructor toma un objeto `Key`; los procesos cliente y servidor crean un objeto `Key`, ya sea pasando el mismo valor de clave, o utilizando el mismo nombre de archivo e identificador. De esta manera, crearán un objeto `Semaphore` que haga referencia

al mismo semáforo. Una vez que se ha creado el semáforo, los procesos cliente y servidor tienen acceso al semáforo mediante las llamadas a las funciones miembro `Acquire()` y `Release()`.

El objeto `Semaphore` encapsula las llamadas de sistema para semáforos que se describen en esta sección. El programa `semtest` que viene en el CD-ROM muestra el uso del objeto `Semaphore`.

## Memoria compartida

La memoria compartida es un método que permite que dos procesos comparten datos de manera directa. Regresemos a lo discutido anteriormente sobre las colas de mensajes, tuberías y tuberías con nombre. Cada uno de estos métodos de comunicación entre procesos permite que varios procesos se pasen datos entre sí. Una cuestión a cerca del rendimiento con estos métodos de comunicación entre procesos es que los datos que se copian entre los procesos se copian del proceso de origen al kernel, y luego del kernel al proceso de destino. Estas copias son costosas.

La memoria compartida permite que varios procesos obtengan un apuntador a un área de memoria que sea compartida entre los procesos. Tener un apuntador a un segmento de memoria elimina las copias costosas hacia y desde el kernel.

La memoria compartida también tiene sus desventajas. Aunque cada uno de los procesos tiene acceso a un segmento de memoria, los accesos a esa memoria deben estar sincronizados.

Por lo regular, la sincronización de procesos para la memoria compartida se implementa por medio de semáforos, como se describió en la sección anterior.

Antes de utilizar la memoria compartida, primero se debe crear; `shmget()` crea un segmento de memoria compartida.

```
/* Obtener segmento de memoria compartida. */
extern int shmget __P ((key_t __key, int __size, int __shmflg));
```

Cada segmento de memoria compartida se mantiene en el kernel mediante una estructura `shmid_ds`. `shmid_ds` se define en `<kernel/shm.h>`. El listado 25.14 muestra la estructura `shmid_ds` de Linux.

---

### ENTRADA LISTADO 25.14 Definición de la estructura `shmid_ds` de Linux

---

```
1: // Listado 25.14 Definición de una estructura shmid_ds
2:
3: struct shmid_ds
4: {
5:     struct ipc_perm shm_perm;    /* permisos de operación */
6:     int shm_segsz;             /* tamaño del segmento (en bytes) */
7:     __kernel_time_t shm_atime; /* última unión */
8:     __kernel_time_t shm_dtime; /* última separación */
```

```

9:     __kernel_time_t shm_ctime; /* último cambio */
10:    __kernel_ipc_pid_t shm_cpid; /* pid del creador */
11:    __kernel_ipc_pid_t shm_lpid; /* pid del último operador */
12:    unsigned short shm_nattch; /* núm. de uniones actuales */
13:    unsigned short shm_unused; /* compatibilidad */
14:    void * shm_unused2; /* idem - usado por DIPC */
15:    void * shm_unused3; /* no utilizado */
16:};0

```

**ANÁLISIS** La llamada de sistema `shmget()` abre o crea un segmento de memoria compartida, pero no proporciona acceso a esa memoria. Para que un proceso pueda tener acceso a la memoria compartida, se debe unir a ésta. La llamada de sistema `shmat()` regresa un apuntador al segmento de memoria compartida:

```

/* Unir segmento de memoria compartida. */
extern void *shmat __P ((int __shmid, __const void * __shmaddr,
                          int __shmflg));

```

Cuando un proceso termina con la memoria compartida, debe separarse del segmento de memoria compartida. Esto se hace mediante una llamada a `shmdt()`.

```

/* Separar segmento de memoria compartida. */
extern int shmdt __P ((__const void * __shmaddr));

```

Después de que todos los procesos se separan de la memoria compartida, el segmento de memoria compartida debe ser eliminado. Esto se logra mediante una llamada de sistema `shmctl()`.

```

/* Operación de control de memoria compartida. */
extern int shmctl __P ((int __shmid, int __cmd, struct shmid_ds *__buf));

```

El listado 25.15 define un objeto `SharedMemory` que encapsula todas las llamadas de sistema para memoria compartida definidas en esta sección.

25

### ENTRADA LISTADO 25.15 Definición de la clase SharedMemory

```

1: // Listado 25.15 Clase para memoria compartida (smem.h)
2:
3:
4: #ifndef C_SMEM_H
5: #define C_SMEM_H
6:
7: #include <sys/shm.h>
8: #include <string.h>
9: #include "lst25-04.h" // #include "key.h"
10: #include "lst25-01.h" // #include "object.h"
11:

```

continúa

**LISTADO 25.15** CONTINUACIÓN

```
12: class SharedMemory : public Object
13: {
14:     public:
15:         SharedMemory(Key k);
16:         ~SharedMemory();
17:         int Create();
18:         int Create(int flags, int size);
19:         int Destroy();
20:         char * Attach();
21:         void Detach();
22:         int Read(char * buf, int len, int offset);
23:         int Write(char * buf, int len, int offset);
24:     private:
25:         // no permitir la copia
26:         SharedMemory & operator=(const SharedMemory &);
27:         bool init_;
28:         Key key_;
29:         int shmid_;
30:         char * shmaddr_;
31:     };
32:
33: #endif
```

**ANÁLISIS** El objeto `SharedMemory` definido en el listado 25.15 contiene las funciones miembro estándar `Create()` y `Destroy()` definidas para todos nuestros objetos. Observe que el constructor toma un objeto `Key`. Los procesos cliente y servidor crean un objeto `Key`, ya sea pasando el mismo valor de clave, o utilizando el mismo nombre de archivo e identificador. De esta manera, crearán un objeto `SharedMemory` que haga referencia al mismo segmento de memoria compartida. Una vez que se ha creado el objeto de memoria compartida, los procesos cliente y servidor pueden leer y escribir directamente en la memoria compartida, usando el apuntador regresado por la llamada de sistema `shmat()`. Es importante observar que el acceso a la memoria compartida debe estar sincronizado mediante el uso de un semáforo.

## IPC

Linux también ofrece soporte para la llamada de sistema conocida como `ipc()`. Ésta es una llamada centralizada de sistema que se utiliza para implementar las llamadas de IPC de System V en Linux. `ipc()` sólo se implementa en Linux y no es portable hacia otros sistemas UNIX. Por medio de `ipc()`, puede hacer cualquiera de las llamadas de sistema

de IPC de System V mencionadas hoy. A continuación se muestra el prototipo de la función `ipc()`:

```
int ipc(unsigned int call, int first, int second, int  
       third, void *ptr, long fifth);
```

## Resumen

Hoy hemos definido varios métodos disponibles para el programador de Linux en relación con la comunicación entre procesos (IPC). Cada una de las secciones de esta lección habla sobre el uso de los métodos, y sobre los detalles relacionados con dicho uso.

En la primera mitad de esta lección hablamos sobre algunos de los primeros métodos disponibles en los programas UNIX para la comunicación entre procesos: tuberías y tuberías con nombre.

En la segunda mitad de la lección hablamos sobre la comunicación entre procesos de System V, los mensajes, semáforos y la memoria compartida, y desarrollamos algunos objetos de C++ para utilizar estos métodos de IPC.

25

## Preguntas y respuestas

- P** ¿Cuáles son las ventajas de usar colas de mensajes en lugar de una tubería? ¿Y en lugar de una tubería con nombre?
- R** Las colas de mensajes son un canal dúplex total. Las tuberías y las tuberías con nombre son semidúplex.
- P** ¿Por qué es importante sincronizar el acceso a la memoria compartida?
- R** Como los procesos se están ejecutando independientemente uno de otro, cada uno podría escribir en la memoria compartida y corromper los datos leídos, o los escritos, por los otros procesos.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, "Respuestas a los cuestionarios y ejercicios", y asegúrese de comprender las respuestas antes de pasar al siguiente día.

## Cuestionario

1. Enliste las tres rutinas utilizadas para crear métodos de comunicación entre procesos de System V.

2. ¿Qué señal se produce si los extremos de lectura y de escritura de una tubería no están preparados?
3. ¿Por qué la memoria compartida es más rápida que los mensajes?
4. ¿Qué es un semáforo binario?

## Ejercicios

1. Implemente un programa cliente/servidor en el que el cliente y el servidor comparten datos usando la clase `SharedMemory`, y sincronice el acceso a la memoria compartida usando la clase `Semaphore`.
2. Usando tuberías, prepare una comunicación dúplex total entre un proceso padre y un proceso hijo.
3. Extienda la clase `NamedPipe` para que pueda abrir tuberías que no se bloqueen y una función miembro `Read()` que no bloquee si no hay datos disponibles.

# SEMANA 4

## DÍA 26

### Programación de la GUI

En las lecciones anteriores ha visto cómo se construye el lenguaje C++, su sintaxis y tipos de datos, y cómo se pueden crear objetos con él que puedan interactuar para producir programas útiles que funcionen en el sistema operativo Linux.

En esta lección llevará estos conceptos más allá, y verá cómo puede utilizar clases y objetos para representar objetos *reales* en forma gráfica, de manera que pueda interactuar con ellos y con su programa con un ratón u otro dispositivo gráfico de entrada.

Hoy aprenderá lo siguiente:

- Qué es una GUI (interfaz gráfica de usuario), y por qué es tan importante para Linux
- Cómo las GUIs representan objetos conocidos como *widets* con los que se puede interactuar
- La historia de las GUIs en UNIX
- Dos nuevas GUIs o *escritorios gratuitos* para Linux
- Cómo escribir aplicaciones gráficas para Linux usando los paquetes de herramientas de desarrollo que vienen con las distribuciones de la GUI
- Cómo escribir aplicaciones multiplataforma simples para GUI en C++ usando el nuevo kit de herramientas wxWindows

## El escritorio de Linux

Hasta hace poco, una de las quejas más frecuentes con respecto a Linux como sistema operativo (así como con respecto a otros sistemas UNIX) era que sólo tenía una interfaz de línea de comandos en forma de un “shell” o de texto simple en la pantalla. Esto significa que se interactúa con la computadora escribiendo comandos como texto y se obtiene la salida de la misma manera: como texto. Ni siquiera es texto bonito: es texto monoespaciado estilo máquina de escribir, grueso, atroz y sin formato.

Esto está bien para los fanáticos de computadoras y los desarrolladores, y también está bien para sistemas de servidores que se conectan a otras aplicaciones cliente más amigables para el usuario, pero es un obstáculo masivo para llevar a Linux a donde pertenece: al escritorio, literalmente. El usuario de computadora promedio quiere resultados, no quejas ni argumentos que parecen ser interminables en relación con entradas inadecuadas.

Las cosas empezaron a mejorar desde principios de los 80, cuando muchas aplicaciones que aparecieron en el mercado estaban en el punto intermedio, entre la interfaz de línea de comandos y la GUI (interfaz gráfica de usuario). Ésta era la interfaz no gráfica basada en menús, que le permitía interactuar por medio de un ratón en lugar de tener que escribir comandos con el teclado. Aún así, la interfaz seguía siendo sea e incómoda, pues las imágenes y los cuadros se dibujaban por medio de los caracteres de línea del conjunto extendido de caracteres ASCII. No era tan agradable como las GUIs de la actualidad.

Desde luego que, en la actualidad, la mayoría de los sistemas operativos importantes, como varias versiones de Windows, el sistema operativo de Macintosh y OS/2 proporcionan una verdadera GUI en donde los objetos con los que el usuario interactúa se dibujan en forma limpia y precisa, píxel por píxel.

Por lo general, las aplicaciones utilizan los elementos de la GUI que vienen con el sistema operativo y agregan sus propios elementos e ideas de GUI. A veces una GUI utiliza una o más metáforas para los objetos familiares de la vida real: escritorios, ventanas, o la descripción física de un edificio.

Los elementos de una GUI incluyen ventanas, menús, botones, casillas de verificación, controles deslizantes, medidores, barras de desplazamiento, iconos, iconos emotivos que cambian su naturaleza en tiempo real a medida que nos desplazamos por el sistema de archivos, asistentes, ratones y muchas otras cosas. Los dispositivos multimedia ahora forman parte de la mayoría de las GUIs, y las interfaces de sonido, voz, vídeo en movimiento y realidad virtual parecen convertirse en una parte estándar de la GUI para muchas aplicaciones.

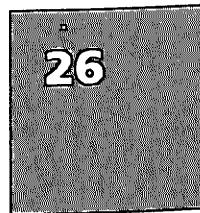
La interfaz gráfica de usuario de un sistema, junto con sus dispositivos de entrada, algunas veces se conoce como "look and feel". Otra cosa que tiende a caracterizar la aplicación GUI es que por lo general se *controla por eventos*, y no por *procedimientos*. Esto significa que, por lo general, una aplicación GUI espera que usted le pida que haga algo, en lugar de empezar desde el principio y seguir un camino lógico hasta algún punto de terminación, y luego terminar. Hay que tener en cuenta que un programa controlado por eventos también puede tener tareas que se controlen por procedimientos en su interior. Un ejemplo de esto es un IDE (Entorno de Desarrollo Integrado) de GUI, el cual espera que usted le pida crear el programa (un evento) y luego ejecutará a make por usted para realizar la creación (por procedimiento).

Las GUIs más conocidas por la mayoría de las personas en estos sistemas operativos modernos y sus aplicaciones se originaron en el Laboratorio de Investigación de Xerox en Palo Alto, a finales de los 70. La compañía Apple las utilizó en sus primeras computadoras Macintosh; después, Microsoft utilizó muchas de las mismas ideas en su primera versión del sistema operativo Windows para las PCs compatibles con IBM. Pero esto no se aplica para Linux: no tiene una GUI nativa, a diferencia de sistemas operativos como Windows NT y Apple Macintosh, en donde la GUI es el sistema operativo, al menos en parte.

Lo que realmente distingue a Linux de otros sistemas operativos modernos es que, aunque se puede crear una GUI para él usando una tarjeta de vídeo, las capacidades del monitor y software para controlar todo, la GUI realmente no forma parte del sistema operativo en sí. En vez de esto, la GUI está por encima del sistema operativo y disfraza la mayor parte de la complejidad de la línea de comandos del sistema operativo para el usuario. La mayoría de los usuarios considera que esto es algo muy bueno.

Aunque ésta puede parecer una diferencia trivial e incluso pedante entre éste y los sistemas operativos que no tienen funcionalidad GUI nativa, es una diferencia que provee a Linux (y a usted) de un enorme alcance en cuanto a flexibilidad.

Linux es nuevo, pero sus antecedentes, los variados tipos de UNIX, que son demasiados como para mencionarlos todos, han estado presentes por décadas, y en el pasado las personas han intentado utilizar las capacidades para gráficos estándar de UNIX representadas por el sistema X Windows para facilitar la funcionalidad mediante la GUI. Pero la programación de X es compleja y difícil (vea el recuadro "El protocolo X y los desarrollos más recientes"), e incluso kits de herramientas como Motif y su clon gratuito LessTif, que están diseñados para ocultar la mayoría de los horrendos detalles de la programación de X, tienen muy poco éxito en el mejoramiento de un mal trabajo.



### El protocolo X y los desarrollos más recientes

Probablemente haya observado en esta lección que mencionamos con frecuencia el término "X" en relación con los gráficos en UNIX.

La organización llamada X Org, un consorcio sin fines de lucro formado por miembros de todo el mundo, desarrolló el protocolo a mediados de los 80 para responder a la necesidad de una interfaz gráfica de usuario transparente a las redes, principalmente para el sistema operativo UNIX.

X proporciona el despliegue y el manejo de información gráfica, en forma muy similar a la GDI (interfaz gráfica de dispositivo) de Microsoft Windows y al Administrador de Presentación (Presentation Manager) de IBM.

La diferencia clave se encuentra en el diseño del protocolo X en sí. Microsoft Windows y el Administrador de Presentación de IBM simplemente despliegan información gráfica en la PC en la que se están ejecutando, mientras que el protocolo X distribuye el procesamiento de las aplicaciones mediante la especificación de una relación cliente-servidor en el nivel de aplicación.

La parte del "qué se va a hacer" de la aplicación se llama cliente X y está lógica y, por lo general, físicamente separada de la parte "cómo hacerlo", que viene siendo la pantalla, y se llama servidor X. Por lo general, los clientes X se ejecutan en un equipo remoto que tiene poder computacional de sobra y se despliega en un servidor X. Ésta es una verdadera relación cliente-servidor entre una aplicación y su pantalla, y tiene toda la secuela de ventajas de esta relación. (Note que el cliente y el servidor se distribuyen en forma inversa al estándar; en realidad el servidor es el que ofrece el despliegue y no el poder de cómputo.)

Para lograr esta separación, la aplicación (cliente X) se divorcia de la pantalla (servidor X), y los dos se comunican mediante un protocolo asíncrono basado en sockets que opera en forma transparente a través de una red.

Además, X proporciona un sistema común de manejo de ventanas mediante la especificación de un nivel dependiente del dispositivo, así como de uno independiente. En esencia, el protocolo X oculta las peculiaridades del sistema operativo y del hardware que lo soporta de las aplicaciones que lo utilizan. En teoría, distintas configuraciones de hardware presentarán una interfaz común de software y aún así tendrán componentes internos bastante distintos.

Esto está muy bien en teoría, pero la realidad es que la API (interfaz de programación de aplicaciones) Xlib, escrita en C, es demasiado complicada, y la programación de X es algo así como un arte oculto.

Recientemente se han escrito varios niveles que están por encima de Xlib para tratar de simplificar la programación de X, pero estos intentos, principalmente Motif y su clon gratuito, LessTif, sólo tienen éxito en parte (la API aún es muy mala) y, de cualquier forma, la "look and feel" se está volviendo obsoleta.

Más recientemente, hemos visto desarrollos como GTK++ (una API de C que se encuentra por encima de GLib y GDK, que a su vez está un nivel arriba de Xlib), wxWindows (un nivel de C++ por encima de GTK++) y la biblioteca Qt de TrollTech (un nivel de C++ por encima de Xlib en UNIX y por encima de la GDI en Windows).

Toda esta división de niveles tal vez le provoque un dolor de cabeza, pero en realidad no es motivo de preocupación: hasta las bibliotecas de gráficos de C++ más conocidas, como

la MFC de Microsoft y las clases C++ Builder de Borland son poco más que niveles de C++ por encima de la API de C nativa de Windows, y la GDI. Evidentemente, GDK++, wxWindows, y en especial Qt, tienen una perspectiva que puede ser correcta. Las últimas dos bibliotecas proveen árboles de código fuente independiente de la plataforma; es decir, usted escribe una vez, compila eso para su plataforma usando el kit de herramientas de bibliotecas nativas, y lo ejecuta. Esto es algo parecido a Java, sólo que la compilación se realiza después de enviar el código y no cuando el programador lo construye.

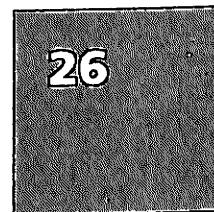
Los ejemplos de Qt y wxWindows que se ven posteriormente en esta lección se compilarán y ejecutarán tanto en plataformas Linux como en Windows.

Pero, he aquí un dilema: tenemos un sistema operativo de primera clase *gratuito* y robusto, que el usuario promedio no querrá utilizar debido a que su GUI se ve como lo que es: aburrida, obsoleta y muy incómoda.

Para fortuna de Linux y de nosotros, los programadores que seremos convocados a escribir las nuevas aplicaciones, hay dos proyectos nuevos que afianzan a Linux en el escritorio, con sus intentos exitosos de convertir la GUI de Linux en una interfaz excelente y eminentemente utilizable.

Estos dos proyectos son el proyecto GNOME (Entorno GNU de Modelo de Objetos de Red), y el KDE (Entorno de Escritorio K). Ambos proyectos son software gratuito y liberado bajo la GPL. KDE tiene uno o dos problemas de licencia con su biblioteca de gráficos de soporte Qt de TrollTech, pero al parecer ya están resueltos; sin embargo, esto parece disuadir a los aficionados empedernidos de GNU de adoptar a KDE como software genuinamente gratuito.

Esta lección examina a GNOME y a KDE y crea algunas aplicaciones básicas para mostrar qué tan simple puede ser escribir software controlado por GUI para estos entornos.



## Qué es GNOME

GNOME es el Entorno GNU de Modelo de Objetos de Red, además de ser el escritorio GUI del proyecto GNU.

Citando del anuncio original del grupo de noticias de usenet `comp.os.linux.announce`, se pretende que GNOME sea

“...un conjunto gratuito y completo de aplicaciones y herramientas de escritorio amigables para el usuario, similar a CDE y a KDE pero basado completamente en software gratuito”.

Y, evidentemente, GNOME es todo lo que se podría esperar de un entorno moderno de programación. A este respecto, es casi igual a CDE (Entorno Común de Escritorio), Win32, NextStep y KDE.

La gran diferencia es que, en contraste con los ejemplos antes mencionados, todos los componentes de GNOME son gratuitos y están liberados ya sea bajo la GPL o la LGPL. Y no sólo eso, además GNOME es extremadamente flexible en comparación con la mayoría de los entornos de escritorio. Como un bono adicional para el usuario, se puede personalizar fácilmente para adaptarse a las necesidades específicas.

GNOME utiliza CORBA (Arquitectura Común de Agente de Solicitud de Objetos) del Grupo de administración de objetos para permitir que los componentes de software operen entre sí sin problemas, sin importar el lenguaje en el que se implementen ni el equipo en el que se ejecuten. Además, los desarrolladores de GNOME están trabajando duro para desarrollar un modelo de objetos llamado Bonobo, con base en CORBA, el cual es similar a OLE2 (Vinculación e Incrustación de Objetos, versión 2) de Microsoft.

Bonobo permitirá a los programadores exportar e importar recursos como componentes y, por ejemplo, permitirá que los usuarios utilicen en su entorno de desarrollo el editor que prefieran, siempre y cuando éste sea soportado mediante una interfaz de editor CORBA estandarizada.

GNOME utiliza el kit de herramientas de Gimp (GTK++) como kit de herramientas de gráficos para todas las aplicaciones gráficas. GTK++ tiene muchas características excelentes y surgió del desarrollo del GIMP (Programa GNU de Procesamiento de Imágenes), el cual merece un libro por sí solo. Como GTK++ sostiene a GNOME, ambos utilizan Imlib, una biblioteca de imágenes para el sistema X Windows, el cual soporta varios formatos de imágenes, desde XPM hasta PNG, y varios fondos de bits, desde color verdadero de 24 bits hasta blanco y negro de 1 bit, y todo es transparente para el programador.

Las aplicaciones GNOME son conscientes de la sesión: por decir, si usted apaga el procesador de palabras de GNOME y luego lo vuelve a iniciar, éste abrirá el documento que usted había abierto anteriormente, y colocará su cursor en el mismo lugar. Esto es posible gracias al sistema Administración de Sesión de X, como se aplica en el Administrador de Sesión de GNOME.

GNOME también ofrece soporte para los métodos de internacionalización y localización de estándares *Uniforum*, permitiendo que se agregue soporte para nuevos lenguajes sin que se necesite volver a compilar la aplicación.

En realidad, GNOME puede ser algo difícil de instalar debido a sus muchas dependencias y algunas cosas más, pero los fabricantes de Linux ahora lo incluyen en la distribución estándar. Por ejemplo, GNOME se incluyó a partir de la versión 5.0 de Red Hat Linux, y lo puede establecer como su escritorio predeterminado al momento de la instalación.

## Cómo obtener GNOME y otros recursos de GNOME

La página Web de GNOME se encuentra en <http://www.gnome.org/>. Si está impaciente por obtener el software, puede ir directo a la página de descargas de Linux en

<http://www.gnome.org/start>. Encontrará FAQs (Preguntas frecuentes) de GNOME en <http://www.gnome.org/faqs/> y mucha más información útil en <http://developer.gnome.org/projects/gdp>, y en <http://www.gnome.org/resources/mailing-lists.html> que tiene un catálogo muy amplio de listas de correo relacionadas con GNOME.

Si está interesado en desarrollar software GNOME o aplicaciones que utilicen GNOME, es conveniente que vea en <http://developer.gnome.org/>, donde encontrará mucha información y recursos adicionales.

La liberación actual de GNOME es 1.2.

## Cómo obtener GTK++ y otros recursos de GTK++

La página Web de GTK++ está en <http://www.gtk.org/>, y usted puede descargar la liberación más reciente desde <ftp://ftp.gtk.org/pub gtk/v1.3/>. La página Web de GTK++ tiene muchos vínculos hacia otros sitios útiles, así como documentación esencial.

Puede obtener GTK—, una biblioteca de envoltura de C++ gratuita relacionada con GTK++, en <http://gtkmm.sourceforge.net/>. La distribución también incluye a gnome—, una envoltura de C++ para las bibliotecas de GNOME. En enero del 2000, gnome— no parecía tener documentación, y GTK— era algo inestable. Tal vez valga la pena que vea estos proyectos otra vez.

Para GTK++ 1.3.2 (versión actual) y posteriores, necesitará la libsigc++ 1.0.1 o posterior. Puede obtenerla en <http://libsigc.sourceforge.net/stable.html>.

## Qué es KDE

KDE solía ser el “Entorno Agradable de Desarrollo”, pero, afortunadamente, le han quitado la palabra “Agradable”.

KDE es un entorno de escritorio moderno transparente para la red para estaciones de trabajo UNIX. Satisface admirablemente la necesidad de un escritorio fácil de usar para las estaciones de trabajo UNIX, similar a los entornos de escritorio que se encuentran en Mac OS y Windows.

Con la llegada de KDE, ahora hay disponible un entorno de escritorio moderno y fácil de usar para UNIX que rivaliza con cualquier otro que haya en el mercado. Al igual que GNOME, KDE puede ser bastante difícil de instalar, pero también se incluye en liberaciones de Linux de varios fabricantes. Red Hat Linux 6.0 y posteriores también vienen con KDE, y se pueden instalar como el escritorio predeterminado en lugar de GNOME. De hecho, puede configurar su equipo para cambiar a una sesión GNOME o a una sesión KDE cada vez que entre al sistema.

**Nota**

Aunque la instalación de las distribuciones estándar de GNOME y KDE desde el CD es una manera fácil y tentadora de tenerlos listos y funcionando, una desventaja es que tal vez algunas de las liberaciones más recientes de software útil, como KDevelop, KOffice y GNOME Office, necesiten versiones más recientes de las bibliotecas centrales que las que están disponibles en el CD. Esto es inevitable, ya que las liberaciones en CD se tienen que congelar en algún punto.

Junto con Linux, KDE proporciona una plataforma completamente abierta, disponible sin costo para cualquiera, incluyendo su código fuente para poder modificarla y así satisfacer las necesidades de cada individuo.

Aunque en general esto es cierto, hay algunas cuestiones relacionadas con las bibliotecas Qt que hacen la interfaz entre KDE con las profundidades de la interfaz Xlib, y para muchos puritanos esto significa que KDE no es software gratuito. Evidentemente, no tiene que pagar por usarlo, por modificarlo o por distribuirlo, pero no puede *vender* software que usted escriba utilizando las bibliotecas de KDE; a menos que compre una licencia profesional de TrollTech.

A pesar de las objeciones de los puritanos, y aunque siempre habrá espacio para mejorar, los desarrolladores de KDE, un grupo no muy acoplado de programadores que están enlazados por medio de Internet, han ideado una alternativa viable para algunas de las combinaciones de sistemas operativos/escritorios más populares y comerciales que se puedan obtener.

Para el usuario, KDE ofrece, entre otras cosas

- Un sistema integrado de ayuda para un acceso conveniente y consistente a la ayuda relacionada con el uso del escritorio KDE y sus aplicaciones.
- Look and feel consistente en todas las aplicaciones KDE.
- Menús y barras de herramientas estandarizados, enlaces de teclas, esquemas de colores, etc.
- Internacionalización: KDE está disponible en más de 25 lenguajes.
- Una gran cantidad de aplicaciones KDE útiles.

En realidad, la look and feel predeterminada de KDE es asombrosamente parecida a la GUI de Windows 95, algo que no creo que sea pura coincidencia.

## Cómo obtener KDE y otros recursos de KDE

La página Web de KDE se encuentra en <http://www.kde.org/>, y puede descargarlo de cualquiera de los sitios espejo enlistados en <http://www.kde.org/mirrors.html>. Los vínculos que vienen en esta página tratan sobre temas que van desde los archivos de listas de correo de KDE y la documentación de KDE, hasta las camisetas y juguetes de peluche de KDE (y no estoy bromeando).

Si está pensando en desarrollar aplicaciones KDE, necesitará la liberación gratuita de las bibliotecas Qt de TrollTech; puede obtenerlas en su sitio Web que está en <http://www.troll.no/>. La versión más reciente de Qt es 2.2.3. Tome nota de que la versión gratuita es sólo para aplicaciones X. Si quiere compilar y enlazar su código en plataformas Windows, o si quiere vender sus programas, necesita comprar su paquete Profesional.

La versión actual de KDE es 2.0.1, pero está disponible la versión 2.1.0 beta.

Puede leer acerca de KDevelop y descargar este software visitando <http://www.kdevelop.org/>. La versión actual es 1.4, pero este número parece ir cambiando casi a diario. Para cuando usted esté leyendo este libro, ya se habrá actualizado varias veces, por lo que será conveniente que consulte la página Web para la última liberación.

## Programación con C++ en el escritorio de Linux

Para ayudar a los programadores a utilizar los servicios ofrecidos por GNOME y KDE, cada uno exporta una API a la que los programadores pueden escribir, para así asegurar que exista compatibilidad y consistencia con otros programas escritos para el escritorio. Las aplicaciones se conforman a un formato fijo, o *marco de trabajo de aplicación*; éste es un término conocido para alguien que haya programado con la MFC de Microsoft o con C++ Builder de Borland.

En KDE, la biblioteca Qt proporciona el marco de trabajo y el mecanismo que utilizan los objetos de Qt para comunicarse entre sí; en GNOME, la API de GNOME proporciona servicios similares. Llamaremos *widgets* a los objetos que están en la GUI de UNIX. Vea el recuadro "Widgets".

Veremos un tercer marco de trabajo en la biblioteca wxGTK. Esta biblioteca es similar al marco de trabajo de Qt y, evidentemente, a los marcos de trabajo de Microsoft y de Borland, en que implementa un marco de trabajo Documento/Vista en el que el Documento representa conceptualmente a los objetos de datos, y la Vista representa conceptualmente la vista que tiene el usuario de esos datos.

26

### Widgets

En términos computacionales, un widget es un elemento de una GUI (interfaz gráfica de usuario) que despliega información o proporciona una manera específica para que un usuario interactúe con el sistema operativo y los programas de aplicaciones. Los widgets incluyen iconos, menús desplegables, botones, cuadros de selección, indicadores de progreso, cuadros de verificación, barras de desplazamiento, ventanas, bordes de ventanas (que nos permiten cambiar el tamaño de la ventana), interruptores, formularios y muchos otros dispositivos para desplegar información y para pedir, aceptar y responder a las acciones del usuario.

En programación, un widget también es el pequeño programa que se escribe para poder describir la apariencia de un widget específico, la forma en que se comporta, y cómo interactúa con el usuario. La mayoría de los sistemas operativos incluyen un conjunto de widgets predefinidos que el programador puede incorporar en una aplicación, especificando la forma en que debe comportarse. Puede crear nuevos widgets extendiendo los ya existentes, o puede escribir sus propios widgets partiendo desde cero.

Este término se aplicó aparentemente por primera vez en sistemas operativos basados en UNIX y en el sistema X Windows. En OOP (programación orientada a objetos), cada tipo de widget se define como una clase (o una subclase bajo una clase widget genérica y amplia) y siempre se asocia con una ventana específica. Por ejemplo, en el kit de herramientas AIX Enhanced X-Window, un widget es el tipo de datos fundamental.

## Fundamentos de la programación en GNOME

GNOME es el escritorio de GNU y, a diferencia de KDE, está escrito completamente en C.

Se basa en la excelente biblioteca GTK++, que es en sí una biblioteca de envoltura de C alrededor de GDK, que a su vez es una envoltura alrededor de las bibliotecas Xlib nativas (vea el recuadro “El protocolo X y los desarrollos más recientes”).

Aunque éste es un libro para enseñar a programar en C++, vale la pena dar un vistazo breve a las APIs GTK++ y C de GNOME para ilustrar algunos conceptos de programación útiles que se basan en las bibliotecas de envoltura de C++ más relevantes que se examinan posteriormente en esta lección.

En primer lugar, como GNOME utiliza GTK++ como su motor de gráficos, daremos un breve vistazo a GTK++. Ésta es una biblioteca para crear interfaces gráficas de usuario. Tiene licencia de la LGPL, por lo que puede desarrollar software abierto, software gratuito o incluso software comercial con GTK++ sin tener que gastar nada en cuanto a licencias o regalías.

Se llama Kit de herramientas de GIMP debido a que fue originalmente escrito para desarrollar el GIMP (Programa General de Manipulación de Imágenes, otra aplicación excelente de software gratuito), pero ahora GTK++ se ha utilizado en un gran número de proyectos de software, incluyendo el proyecto GNOME (Entorno GNU de Modelo de Objetos de Red).

GTK++ está creado por encima de GDK (Kit de Dibujo de GIMP), el cual es básicamente una envoltura alrededor de las funciones de bajo nivel para tener acceso a las funciones de soporte para manejo de ventanas (Xlib, en el caso del sistema X Windows). GTK++ es en esencia una API orientada a objetos. Aunque está escrita completamente en C, se implementa usando la idea de clases y funciones callback (apuntadores a funciones).

Existe una unión de C++ de nivel delgado con GTK++, que se conoce como GTK—, la cual proporciona una interfaz más parecida a C++ para GTK++. En la liberación actual, GTK++ 1.3, hay también una biblioteca llamada gnome—, la cual es una envoltura de C++ alrededor de la API C de GNOME. Pero gnome— parece ser inmadura en la actualidad, o inexperta en el menor de los casos. Ciertamente, esto puede haber cambiado para cuando usted lea esto.

Si está decidido a usar C++, tiene tres opciones:

- En primer lugar, siempre puede utilizar una de las bibliotecas de envoltura especializadas que envuelven a la API de C de GTK++ en clases de C++, en forma muy parecida a como lo hace GTK—. En la lección de hoy, esto es lo que realmente hacemos, ya que se enlaza a la perfección con el entorno de programación KDE, y esto facilita las comparaciones considerables.
- En segundo lugar, si no quiere confiar sus aplicaciones a gnome— o a otra biblioteca de envoltura de C++ por cualquier otra razón, puede utilizar solamente el subconjunto de C incluido en C++ al hacer una interfaz con GTK++, y luego utilizar la interfaz de C. Recuerde que C es un subconjunto válido de C++ y un programa válido de C es un programa válido de C++, generalmente hablando.
- Por último, puede utilizar juntos a GTK++ y a C++ declarando todas las funciones callback como funciones estáticas en clases de C++, y llamando otra vez a GTK++ por medio de su interfaz de C. El ejemplo botones que se muestra en el listado 26.1 hace esto.

Si elige esta tercera opción, puede incluir un apuntador al objeto que se va a manipular (el apuntador `this`) como valor de los datos de la función callback.

Elegir entre estas opciones es principalmente cuestión de preferencia, ya que en todas se cuenta con C++ y GTK++. En lo personal, la segunda solución me parece algo horrenda y nada elegante, y prefiero utilizar una biblioteca de envoltura de C++, pero es cuestión de gustos.

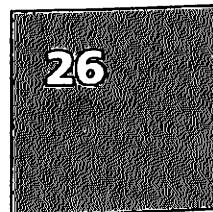
Ninguna de estas opciones requiere del uso de un preprocesador especializado, así que, sin importar cuál sea su elección, puede utilizar C++ estándar con GTK++.

Ahora veamos a GNOME en sí, que se encuentra por encima de GTK++.

Un verdadero programa GNOME es una aplicación GUI de GTK++ que también utiliza las bibliotecas de GNOME. Estas bibliotecas hacen posible tener una look and feel similar en todas las aplicaciones, y que las cosas simples sean simples de programar. Y no sólo eso, también agregan muchos widgets que no pertenecen propiamente a GTK++.

Para infortunio de los programadores novatos de C++, GNOME, al igual que su biblioteca de gráficos de soporte, GTK++, tiene una API de C. Para estar seguros, la API está fuertemente orientada a objetos y utiliza estructuras opacas, funciones de acceso y otras cosas más, pero sin duda es una API de C. Claro que se tienen las mismas tres opciones a elegir tanto en la programación de GNOME como en la programación de GTK++.

Vale la pena que analice un pequeño programa de GNOME (vea el listado 26.1) y que vea cómo encaja en el escritorio de GNOME. También verá cómo puede declarar una clase simple con funciones miembro estáticas que actúan como callbacks. Desde luego que puede hacer la clase tan complicada y rica en funcionalidad como usted quiera.



**ENTRADA****LISTADO 26.1** Un programa básico de GNOME: botones.cxx

```
1: // Listado 26.1 Un programa básico de GNOME: (botones.hxx)
2:
3:
4: #include <gnome.h>
5:
6: class Callback
7: {
8:     public:
9:         static void clicked(GtkWidget * button,
10:             gpointer data);
11:         static gint quit(GtkWidget * widget,
12:             GdkEvent * event,
13:             gpointer data);
14:     };
15:
16: void Callback::clicked(GtkWidget * button,
17:     gpointer data)
18: {
19:     char * string = (char *)data;
20:     g_print(string);
21:     g_print("Uso de Callback de C++\n");
22: }
23:
24: gint Callback::quit(GtkWidget * widget,
25:     GdkEvent * event,
26:     gpointer data)
27: {
28:     gtk_main_quit();
29:     return FALSE;
30: }
31:
32: int main(int argc, char *argv[])
33: {
34:     GtkWidget * app;
35:     GtkWidget * button;
36:     GtkWidget * hbox;
37:
38:     // Inicializar GNOME, esto es muy similar a gtk_init
39:     gnome_init("botones-ejemplo-básico", "0.1", argc, argv);
40:     app = gnome_app_new("botones-ejemplo-básico", "Buttons");
41:     hbox = gtk_hbox_new(FALSE, 5);
42:     gnome_app_set_contents(GNOME_APP (app), hbox);
43:
44:     // enlazar "quit_event" con gtk_main_quit
45:     gtk_signal_connect (GTK_OBJECT (app),
46:         "quit_event",
47:             GTK_SIGNAL_FUNC (Callback::quit),
48:             NULL);
49:
50:     button = gtk_button_new_with_label("Button 1");
51:     gtk_box_pack_start(GTK_BOX(hbox),
52:         button,
53:         FALSE,
```

```
54: FALSE,
55: 0);
56: gtk_signal_connect(GTK_OBJECT(button),
57: "clicked",
58: GTK_SIGNAL_FUNC(Callback::clicked),
59: "Button 1\n");
60:
61: button = gtk_button_new_with_label("Button 2");
62: gtk_box_pack_start(GTK_BOX(hbox),
63: button,
64: FALSE,
65: FALSE,
66: 0);
67: gtk_signal_connect (GTK_OBJECT(button),
68: "clicked",
69: GTK_SIGNAL_FUNC(Callback::clicked),
70: "Button 2\n");
71:
72: gtk_widget_show_all(app);
73: gtk_main ();
74: return 0;
75: }
```

**ANÁLISIS**

La primera parte del código, líneas 6 a 30, contiene la declaración de la clase `Callback` y sus funciones callback estáticas que creamos para responder a eventos que la GUI detecta al hacer clic en los botones desplegados:

Como puede ver, las funciones son simples y directas, `Callback::clicked()` simplemente imprime texto en `stdout` en la línea 21 (compare con `cout` en C++) y `Callback::quit()` sale del programa principal en la línea 28.

La primera llamada que hacemos en `main()` es a `gnome_init()` en la línea 39.

Esto es muy similar a una aplicación GTK++ pura, en la que llamaríamos a `gtk_init()`; de la misma manera, utilizamos una llamada a `gnome_app_new()` en la línea 40 para que nos dé una instancia de una nueva aplicación GNOME. En una aplicación GTK++ pura, la llamada correspondiente sería a `gtk_window_new()`.

Aunque `gnome_app_new()` regresa un objeto `GtkWindow`, después verificamos eso con un `GnomeApp` mediante la macro `GNOME_APP`. `GnomeApp` es el widget principal de cada aplicación. Es la ventana principal de la aplicación que contiene el documento en el que se está trabajando, los menús de aplicaciones, las barras de herramientas y barras de estado, etc. También recuerda las posiciones acopladas de las barras de menús y barras de herramientas, para que el usuario obtenga la ventana de la misma forma en que la aplicación la había dejado la última vez que se cerró.

Crear un widget `GnomeApp` es tan sencillo como llamar a `gnome_app_new()` con el nombre de la aplicación y el título de la ventana principal. Luego puede crear el contenido de la ventana principal y agregarlo al widget `GnomeApp` mediante una llamada a `gnome_app_set_contents()` con el contenido como argumento.

En este caso, creamos un cuadro horizontal para alojar los botones que crearemos en breve y lo agregaremos al widget de la aplicación en las líneas 41 y 42.

El marco de trabajo de aplicación GnomeApp es parte de la biblioteca libgnomeui, y es la parte que en verdad hace que una aplicación GNOME sea lo que es. También hace que la programación de aplicaciones GNOME sea razonablemente simple y directa, y hace que las aplicaciones sean amplias y consistentes en todo el escritorio. Si sólo utiliza GTK++ necesita hacer muchas cosas usted mismo, pero GnomeApp se encarga de la configuración estándar de la GUI de GNOME por usted, y aún así permite que el usuario configure el comportamiento y lo hace consistente para distintas aplicaciones.

Las líneas 45 a 70 enlazan el manejador de eventos `Callback::quit()` con su función `callback` y luego crean dos botones y los enlazan al manejador de eventos `Callback::clicked()`, antes de agregarlos al cuadro horizontal.

Esto significa que cuando hacemos clic en uno de los botones, la función `callback` `Callback::clicked()` dirige y procesa el evento.

Por último, en las líneas 72 y 73 indicamos al widget de la aplicación principal que se muestre a sí mismo y todo su contenido, y luego que entre al ciclo principal de eventos y espere el clic del ratón.

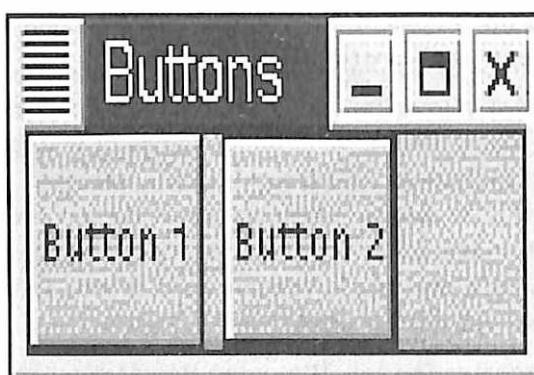
Para crear este programa, puede utilizar el siguiente comando:

```
gcc -g -Wall `gnome-config --cflags gnome gnomeui` LDFLAGS=`gnome-config \
--libs gnome gnomeui` -I/usr/lib/glib/include lstd26-01.cxx -o botones
```

La figura 26.1 le muestra cómo debe lucir su programa botones.

## SALIDA

**FIGURA 26.1**  
*La salida del  
programa botones.*



## Cómo envolver a GTK++ con wxWindows

En la sección anterior vio cómo utilizar C++ con las bibliotecas de GNOME para crear aplicaciones GNOME.

En 1992, un grupo de programadores de la Universidad Edinburgh de Escocia creó la versión 1.0 del kit de herramientas wxWindows. Este kit de herramientas es un conjunto de bibliotecas de C++ puras que facilitan el desarrollo para la GUI. Permite que las aplicaciones de C++ se compilen y ejecuten en distintos tipos de computadoras, sin necesitar más que unos cuantos cambios en el código.

Durante 1997 se originó un esfuerzo para producir un entorno de escritorio estándar en Linux, GNOME. Su conjunto de widgets fue GTK++, creado por encima de X, y parecía que las aplicaciones basadas en GTK++ se convertirían en el estándar en el universo Linux. Esto condujo al equipo de desarrollo de wxWindows a reafirmar su propósito y a liberar wxWindows 2.0.

En realidad, wxWindows no es un entorno de desarrollo para GNOME, ya que no utiliza las bibliotecas de GNOME. Sin embargo, tal vez esta capacidad se incluya en futuras liberaciones ya que, en Linux, wxWindows está enfocada directamente a los entornos GTK++ y GNOME. Aún así, un programa wxWindows se ejecutará sin problemas en KDE y en GNOME, y empleará la look and feel (interfaz gráfica de usuario) del escritorio en el que se encuentre; pero, y esto puede ser importante para algunos usuarios, no tendrá acceso a las funciones de escritorio que proporcionan GNOME y KDE.

### Nota

Cada GUI soportada tiene su propia biblioteca (como GTK++, Motif, o Windows); wxWindows es el nombre genérico para toda la suite de bibliotecas. La que más nos interesa es wxGTK, que es la biblioteca específica que envuelve a GTK++. Sin embargo, al igual que la documentación de wxWindows, utilizaremos los términos wxWindows y wxGTK sin distinción, excepto cuando sea importante diferenciarlas.

26

Además de proveer una API común para funcionalidad de la GUI en un amplio rango de plataformas, wxWindows proporciona la funcionalidad para tener acceso a algunas facilidades del sistema operativo utilizadas comúnmente, como copiar o eliminar archivos, y muy pronto incluirá un conjunto de clases para proporcionar servicios criptográficos.

Por lo tanto, wxWindows es un “marco de trabajo” en el sentido de que proporciona mucha funcionalidad integrada, que puede ser utilizada o reemplazada por la aplicación según sea necesario, ahorrando por consecuencia mucho esfuerzo de codificación. También soporta un conjunto de estructuras básicas de datos, como cadenas, listas enlazadas y tablas de hash.

Sin embargo, wxWindows no es un traductor de una GUI a otra. Por ejemplo, no puede tomar una aplicación Motif y generar una aplicación Windows. Para programar usando una GUI *nativa*, necesita aprender a usar una *nueva* API. No obstante, la API de wxWindows ha sido elogiada por ser intuitiva y simple, y puede ser mucho más fácil aprender a utilizarla que una API de GUI nativa como Motif o Windows. El mensaje es simple: si sabe utilizar wxWindows (o Qt en esta cuestión), no necesita conocer las APIs de GUI nativas.

Este kit de herramientas no es único, hay varios a escoger, pero wxWindows es *completamente* gratuito bajo la GPL, está bien establecido, bien documentado, y tiene una cobertura bastante amplia de funcionalidad de GUI.

El peso que arrastran GNOME y GTK++, gracias a Red Hat Labs y a otros, podría impulsar a wxWindows a una posición bastante importante, como la única herramienta utilizable para producir productos compatibles con GNOME, Windows, Motif, Mac, y tal vez versiones para BeOS. Linux se está convirtiendo en una variante de UNIX cada vez más importante y respetada, y esto producirá algunas aplicaciones wxGTK de corriente principal.

Ahora que sabe lo que es wxWindows, veamos cómo puede utilizarla.

## Creación de su primera aplicación de wxWindows: “¡Hola, mundo!”

El listado 26.2 muestra el código fuente para la aplicación más simple de wxWindows que se pueda tener.

---

### ENTRADA LISTADO 26.2 El programa de wxWindows GNOMEHelloWorld

---

```
1: // Listado 26.2 La aplicación GNOMEHelloWorld
2:
3: #ifdef __GNUG__
4: // #pragma implementation
5: #endif
6:
7: // Para los compiladores que soportan la precompilación,
8: #include "wx/wxprec.h"
9:
10: #ifdef __BORLANDC__
11: #pragma hdrstop
12: #endif
13:
14: #ifndef WX_PRECOMP
15: #include "wx/wx.h"
16: #endif
17:
18: class MyApp: public wxApp
19: {
20:     virtual bool OnInit();
```

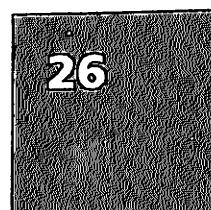
```
21: };
22:
23: class MyFrame: public wxFrame
24: {
25: public:
26:     MyFrame(const wxString & title,
27:             const wxPoint & pos,
28:             const wxSize & size);
29: };
30:
31: IMPLEMENT_APP(MyApp)
32:
33: bool MyApp::OnInit()
34: {
35:     MyFrame * frame = new MyFrame("Hello World",
36:                                     wxPoint(50, 50),
37:                                     wxSize(200, 100));
38:     frame->Show(TRUE);
39:     SetTopWindow(frame);
40:     return TRUE;
41: }
42:
43: MyFrame::MyFrame(const wxString & title,
44:                   const wxPoint & pos,
45:                   const wxSize & size) :
46:     wxFrame((wxFrame *)NULL, -1, title, pos, size)
47: { }
```

**ANÁLISIS**

Lo primero que hacemos es incluir los archivos de encabezado de wxWindows en las líneas 8 y 15 del listado. Observará algunos pragmas e instrucciones `#ifdef` condicionales: éstos son normales y ayudan a que el código sea verdaderamente portable. En alguna parte del kit de herramientas wxWindows hay algunas otras macros que podemos incluir para ayudar con distintos compiladores y otras cosas más, pero por claridad vamos a asumir que sólo tenemos un entorno Linux. Por lo tanto, por el momento puede ignorar sin peligro estas macros.

Desde luego que puede incluir los encabezados de manera individual, por ejemplo `#include "wx/window.h"` o de manera global, que sería `#include "wx/wx.h"`. Esto es especialmente útil en plataformas que soportan encabezados precompilados, como los principales compiladores de la plataforma Windows.

El código en sí empieza en la línea 18 del listado, en donde declaramos nuestra propia clase de aplicación, derivándola de la clase estándar `wxApp`. Prácticamente toda aplicación



wxWindows debe definir una nueva clase derivada de `wxApp`. Entonces podemos redefinir `wxApp::OnInit()` para inicializar el programa como lo hacemos en la línea 20.

En las líneas 23 a 29 derivamos de `wxFrame` la ventana principal; nombramos el marco pasándole una cadena ("Hello World") como parámetro para su constructor, como verá posteriormente en el código.

La siguiente línea de código, la línea 31, puede parecer un poco rara, pero se aclara lo que es al considerar que *todos* los programas de C++ deben tener una función `main()` como el punto de entrada del programa. La macro `IMPLEMENT_APP()` implementa a `main` y crea una instancia de un objeto aplicación, además de empezar el ciclo principal de eventos del programa.

La función miembro `wxApp::OnInit()`, que se implementa en las líneas 33 a 41, se ejecuta al inicio e inicializa el programa al crear la ventana principal y mostrar pantallas instantáneas y otras cosas más. Aquí verá que se da un título al marco en la llamada a su constructor en `MyApp::OnInit()` en la línea 35.

Para compilar el programa, utilice el siguiente comando:

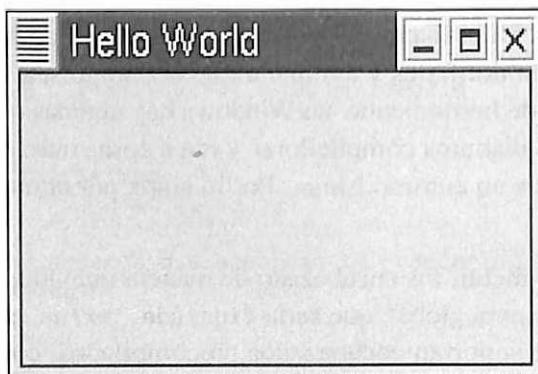
```
g++ 1st26-02.cxx `wx-config --libs` `wx-config --cflags` \  
-o GNOMEHelloWorld
```

**SALIDA**

Al ejecutar el programa, debe ver una ventana sencilla con el título "Hello World" desplegado a lo largo de la parte superior (vea la figura 26.2).

**FIGURA 26.2**

*El primer programa  
GNOMEHelloWorld.*



## Cómo agregar botones a su propia clase de ventana de wxWindows

La primera aplicación, como vio en el listado 26.2, es un programa GNOME completamente funcional, pero es extremadamente aburrido. No permite ninguna interacción por parte del usuario que no sea finalizarla.

Para hacerla más interesante, puede agregarle unas cuantas líneas y colocar algunos botones que le permitan interactuar con ella. En wxWindows, en el marco de trabajo, por lo general se colocan botones y otros widgets dentro de un objeto `wxPanel`; un `wxPanel` es en esencia una `wxWindow` con un poco más de funcionalidad. Por lo general, estas ventanas residen en cuadros de diálogo, pero como verá en el siguiente ejemplo, pueden aparecer casi en cualquier parte.

Primero veamos el listado fuente completo para el nuevo código del listado 26.3.

**ENTRADA****LISTADO 26.3** El programa de wxWindows GNOMEHelloWorld con botones

```
1: // Listado 26.3 Otro ejemplo de GNOMEHelloWorld
2:
3: #ifdef __GNUG__
4: // #pragma implementation
5: #endif
6:
7: // Para compiladores que soporten la precompilación,
8: #include "wx/wxprec.h"
9:
10: #ifndef __BORLANDC__
11: #pragma hdrstop
12: #endif
13:
14: #ifndef WX_PRECOMP
15: #include "wx/wx.h"
16: #endif
17:
18:
19: class MyApp: public wxApp
20: {
21:     virtual bool OnInit();
22: };
23:
24: class MyFrame: public wxFrame
25: {
26: public:
27:     MyFrame(const wxString & title,
28:             const wxPoint & pos,
```

**26***continúa*

**LISTADO 26.3** CONTINUACIÓN

```
29:         const wxSize & size);
30:     void OnQuit(wxCommandEvent & event);
31:     void OnGreet(wxCommandEvent & event);
32:     DECLARE_EVENT_TABLE()
33: private:
34:     wxPanel * m_panel;
35:     wxButton * m_btnGreet;
36:     wxButton * m_btnQuit;
37: };
38:
39: enum { ID_Quit = 1, ID_Greet };
40:
41: BEGIN_EVENT_TABLE(MyFrame, wxFrame)
42:     EVT_BUTTON(ID_Greet, MyFrame::OnGreet)
43:     EVT_BUTTON(ID_Quit, MyFrame::OnQuit)
44: END_EVENT_TABLE()
45:
46: IMPLEMENT_APP(MyApp)
47:
48: bool MyApp::OnInit()
49: {
50:     MyFrame * frame = new MyFrame("Hello World",
51:                                     wxPoint(50,50),
52:                                     wxSize(200,100));
53:     frame->Show(TRUE);
54:     SetTopWindow(frame);
55:     return TRUE;
56: }
57:
58: MyFrame::MyFrame(const wxString & title,
59:                   const wxPoint & pos,
60:                   const wxSize & size) :
61:     wxFrame((wxFrame *)NULL, -1, title, pos, size)
62: {
63:     wxSize panelSize = GetClientSize();
64:     m_panel = new wxPanel(this, -1, wxPoint(0, 0), panelSize);
65:     int h = panelSize.GetHeight();
66:     int w = panelSize.GetWidth();
67:
68:     m_btnGreet = new wxButton(m_panel,
69:                               ID_Greet,_T("Greet"),
70:                               wxPoint(w/2-70, h/2-10),
71:                               wxSize(50, 20));
72:     m_btnQuit = new wxButton(m_panel,
73:                               ID_Quit,_T("Quit"),
74:                               wxPoint(w/2+20, h/2-10),
75:                               wxSize(50, 20));
76: }
```

```
77:  
78: void MyFrame::OnQuit(wxCommandEvent & WXUNUSED(event))  
79: {  
80:     Close(TRUE);  
81: }  
82:  
83: void MyFrame::OnGreet(wxCommandEvent & WXUNUSED(event))  
84: {  
85:     wxMessageBox("Ejemplo de Hello World con wxWindows",  
86:                   "Hello World",  
87:                   wxOK | wxICON_INFORMATION,  
88:                   this);  
89: }
```

**ANÁLISIS** Probablemente, las primeras diferencias que observará se encuentran en la declaración de la clase `MyFrame`. En las líneas 30 a 36 ahora hemos agregado algunas funciones y variables miembro adicionales en la clase `MyFrame`.

Las líneas 30 y 31 declaran dos funciones miembro públicas ordinarias, `OnQuit()` y `OnGreet()`. Verá que ambas toman un apuntador `wxCommandEvent` como su único argumento; los lectores astutos deducirán que estas funciones miembro deben responder de alguna manera a los eventos. Evidentemente, sus nombres son por sí solos una pista.

A continuación verá la macro `DECLARE_EVENT_TABLE()` en la línea 32. Lo que ésta hace es insertar código que actuará como marco de trabajo en la clase para permitirnos asignar eventos a las funciones que deban manejarlos. Verá más sobre esto en un momento.

Por último, el código adicional de esta declaración incluye tres variables miembro, una `wxPanel` y dos `wxButtons`.

Inmediatamente después de la declaración de clase modificada, verá más código nuevo en las líneas 39 a 44.

Este código nuevo primero declara dos nuevos identificadores enumerados para identificar eventos, y luego los agrega a las nuevas funciones miembro que vio declaradas anteriormente. En el código se utilizan las macros `EVENT_TABLE()` para asociar los identificadores de eventos con las funciones que manejarán los objetos de eventos creados. La tabla definida en las líneas 41 a 44 nos está indicando que la función miembro `MyFrame::OnQuit()` manejará los eventos que ocurran con el identificador `ID_Quit`, y que la función miembro `MyFrame::OnGreet()` manejará los eventos que tengan el identificador `ID_Greet`.

A continuación observará que hemos llenado el cuerpo del constructor de `MyFrame` con código para inicializar, posicionar y luego desplegar los botones que prometimos incluir en la aplicación.

La primera línea del constructor, línea 63 del listado, simplemente nos proporciona el tamaño del área cliente del objeto `MyFrame`; un objeto `wxSize` representa las dimensiones `x` (longitud) y `y` (altura) del área cliente. Utilizamos este objeto `wxSize` en la línea 64 para crear un nuevo `wxPanel` que llene completamente el área cliente del objeto `MyFrame`.

Observe que el primer parámetro para el constructor de `wxPanel`, un apuntador al objeto `wxWindows` padre del panel, es `this`, lo que significa que pertenece al objeto `MyFrame`. El constructor predeterminado para la clase `wxPanel` toma varios parámetros que determinan en dónde se coloca, cuál es su tamaño, etc. En la línea 64 le damos un identificador predeterminado de `-1`, establecemos su origen en la esquina superior izquierda del objeto `MyFrame` (`wxPoint(0,0)`), y le damos las dimensiones establecidas en `panelSize`.

También usamos el tamaño del panel para calcular las posiciones de los botones. En este sentido es código “frívolo” ya que su función es meramente cosmética. Reducimos la dimensión del objeto `panelSize` a sus valores componentes, los valores enteros `h` y `w` que se encuentran en las líneas 65 y 66.

Por último, creamos los botones que ha estado esperando pacientemente. Creamos dos de ellos y hacemos que pertenezcan al panel que acabamos de crear. Si analiza los argumentos para el constructor que estamos pasando, es bastante evidente lo que está ocurriendo. Por ejemplo, en la línea 69 se da al botón `m_btnGreet` el identificador de evento `ID_Greet`. Del código anterior, es evidente que este botón generará eventos `ID_Greet` cuando se utilice.

También observará que establecimos el texto del botón en los argumentos del constructor. Los argumentos restantes para el constructor que vemos especifican su posición (que se determina por la altura y el ancho del panel) y su tamaño. Observe que aquí tenemos valores constantes fijos: ésta es, en general, una mala práctica de programación, pero la utilizamos aquí para que el código sea más legible.

Creamos el botón `m_btnQuit` en forma muy similar.

La última sección del código implementa los dos manejadores de eventos, `OnQuit()` y `OnGreet()`, en las líneas 78 a 89.

La única cuestión de interés aquí es la función `wxMessageBox()`, que empieza en la línea 85, y que despliega un mensaje dentro de un cuadro que está en la pantalla.

Para compilar el programa, utilice el siguiente comando:

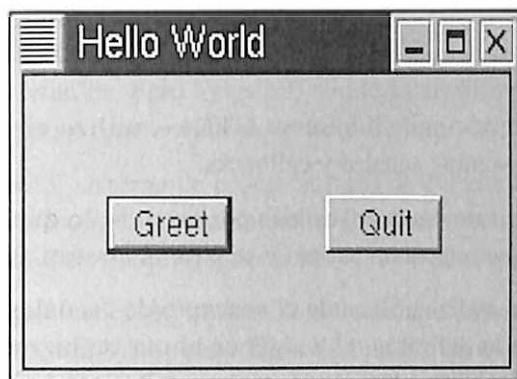
```
g++ lst26-03.cxx `wx-config --libs` `wx-config --cflags` \  
-o GNOMEHelloWorld
```

**SALIDA**

Al ejecutar el programa, deberá ver una ventana sencilla (vea la figura 26.3) con el título "Hello World" mostrado a lo largo de la parte superior de la ventana, y dos botones etiquetados como "Greet" y "Quit" centrados en el área cliente. Hacer clic en el botón Greet mostrará un cuadro de diálogo con un saludo impreso, y hacer clic en Quit hará que salga de la ventana.

**FIGURA 26.3**

*El segundo programa GNOMEHelloWorld, con botones.*



## Interacción de objetos por medio de eventos

Antes de agregar más funcionalidad al programa GNOMEHelloWorld, necesitamos ver la forma en que se comunican los distintos objetos de un programa wxWindows. Al igual que la mayoría de las aplicaciones de GUI, los programas de wxWindows tienden a ser controlados por eventos.

**26**

### Manejo de eventos

Hablando en general, el usuario tiene dos maneras de interactuar con un programa de GUI: el ratón y el teclado. Para ambas formas, una interfaz gráfica de usuario tiene que proporcionar métodos que detecten acciones y métodos que hagan algo como reacción a estas acciones.

Cuando un usuario mueve el ratón, oprime una tecla o hace clic en el botón de un ratón, describimos esto como un evento: es algo que ocurre y brinda información al sistema, información que debe ser identificada, dirigida al objeto que está interesado en recibirla, y luego se debe realizar alguna acción.

Por lo tanto, el sistema de ventanas envía todos los eventos de interacción a la aplicación GUI que tiene el enfoque en la pantalla.

Utilizamos la palabra enfoque para describir a la aplicación que se encuentra actualmente en primer plano, o la que tomará la entrada actual. Por lo general, en la mayoría de los sistemas se cambia la configuración para que la aplicación que tenga el enfoque tenga un borde o marco de color distinto, con lo cual es más fácil saber cuál aplicación se está utilizando actualmente.

Como wxGTK está basado en la biblioteca GTK++, utiliza el mecanismo GTK++ de soporte para manejar eventos: señales y callbacks.

GTK++ es un kit de herramientas controlado por eventos, lo que significa que esperará en `gtk_main()` hasta que ocurra un evento y se pase el control a la función apropiada.

Este paso de control se realiza utilizando el concepto de "señales". Al ocurrir un evento, como el clic de un botón del ratón, el widget en el que se hizo clic "emitirá" la señal apropiada. Así es como GTK++ hace la mayor parte de su trabajo útil. Hay señales que todos los widgets heredan, como `destroy()`, y hay señales específicas de cada widget, como `toggled()` en un interruptor.

### Nota

Observe que las señales del contexto de estas discusiones sobre el desarrollo GUI en GNOME y KDE no tienen nada que ver con las señales tradicionales de UNIX, como SIGHUP; sólo es una "desafortunada" coincidencia de nomenclatura.

Para hacer que un botón realice una acción, configuraremos un manejador de señal para que capture estas señales y llame a la función apropiada. Esto se logra mediante el uso de una función como la que se muestra a continuación:

```
gint gtk_signal_connect(GtkObject *object,
                      gchar *name,
                      GtkSignalFunc func,
                      gpointer func_data);
```

El primer argumento es el widget que estará emitiendo la señal, y el segundo es el nombre de la señal que se quiere capturar. El tercero es la función que se quiere llamar cuando se atrape la señal, y el cuarto son los datos que se quieren pasar a esta función.

La función especificada en el tercer argumento se llama "función callback", y por lo general debe ser de la forma

```
void callback_func(GtkWidget *widget, gpointer callback_data);
```

donde el primer argumento es un apuntador al widget que emitió la señal, y el segundo es un apuntador a los datos proporcionados como último argumento para la función `gtk_signal_connect()`, como se mostró anteriormente.

wxWindows extiende este concepto de señales y callbacks en su propio espacio de nombres y utiliza *tablas de eventos* para asignar acciones a los eventos.

Usted coloca una tabla de eventos en un archivo de implementación para indicar a wxWindows cómo debe asignar eventos a funciones miembro. Estas funciones miembro no son funciones virtuales, pero todas son similares en forma: toman un solo argumento derivado de `wxEvent`, y tienen un tipo de valor de retorno `void`.

Tal vez observe que el sistema de procesamiento de eventos de wxWindows implementa algo muy parecido a los métodos virtuales en C++ normal; es decir, puede alterar el comportamiento de una clase redefiniendo sus funciones de manejo de eventos.

En muchos casos esto funciona incluso para cambiar el comportamiento de los controles nativos. Por ejemplo, también puede filtrar un número de eventos de pulsaciones de teclas enviados por el sistema a un control de texto nativo si redefine a `wxTextCtrl` y define un manejador para eventos de teclas por medio de `EVT_KEY_DOWN`. Esto prevendría sin duda que cualquier evento relacionado con oprimir una tecla fuera enviado al control nativo (que tal vez no sea lo que usted quiere). En este caso, la función manejadora de eventos llama a `Skip()` para indicar que debe continuar la búsqueda del manejador de eventos.

En resumen, en lugar de llamar explícitamente a la versión de la clase base, como lo habría hecho con las funciones virtuales de C++ (por ejemplo, `wxTextCtrl::OnChar()`), debe llamar a `Skip()`.

En la práctica, eso se vería como si el control de texto derivado sólo aceptara las letras de la "a" a la "z" y de la "A" a la "Z":

```
01: void MyTextCtrl::OnChar(wxKeyEvent & event)
02: {
03:     if (isalpha(event.KeyCode()))
04:     {
05:         // El código de tecla está dentro del rango válido. Llamamos a
06:         // event.Skip() para que se pueda procesar el evento ya sea en
07:         // la clase wxWindows base o en el control nativo.
08:         event.Skip();
09:     }
10:     else
11:     {
12:         // pulsación de tecla ilegal. No llamamos a event.Skip() para que
```

```

13:      // el evento no se procese en ninguna otra parte.
14:      wxBell();
15:  }
16: }
```

Verá que la línea 3 comprueba si el código de tecla estaba representando una tecla, y llama a `Skip()` en la línea 8 para continuar la búsqueda de un manejador.

El orden normal de búsqueda en la tabla de eventos que realiza `ProcessEvent` es el siguiente:

1. Si el objeto está deshabilitado, por lo general con una llamada a `wxEvtHandler::SetEvtHandlerEnabled()`, la función salta hasta el paso (6).
2. Si el objeto es `wxWindow`, llamar a `ProcessEvent` en forma recursiva sobre el `wxValidator` de la ventana. Salir de la función si esto regresa `true`.
3. Llamar a `SearchEventTable()` para obtener un manejador de eventos para este evento. Si esto falla, tratar en la clase base, y así sucesivamente hasta que se agoten las tablas o se encuentre una función apropiada, en cuyo caso la función termina.
4. Aplicar la búsqueda descendente por toda la cadena de manejadores de eventos (por lo general, la cadena tiene una longitud de uno). Salir de la función si este paso tiene éxito.
5. Si el objeto es `wxWindow` y el evento es `wxCommandEvent`, aplicar `ProcessEvent` en forma recursiva al manejador de eventos de la ventana padre. Salir si este paso regresa `true`.
6. Llamar a `ProcessEvent` en el objeto `wxApp`.

## Cómo agregar un menú a su propia clase de ventana `wxWindows`

Por último, agregaremos un menú a la aplicación simple que hemos desarrollado. No agregaremos más funcionalidad, excepto proporcionar una segunda forma de invocar la funcionalidad que ya tenemos.

El cambio de código que tenemos que hacer es muy pequeño, así que no tenemos que incluir todo el archivo fuente: todo lo que necesitamos es agregar algo de código adicional (vea el listado 26.4) en el constructor de `MyFrame`.

### **ENTRADA** LISTADO 26.4 El programa de `wxWindows` GNOMEHelloWorld con un menú

```

1: // Listado 26.4  GNOMEHelloWorld
2:
3: #ifdef __GNUG__
4: // #pragma implementation
5: #endif
6:
7: // Para compiladores que soporten la precompilación,
```

```
8: #include "wx/wxprec.h"
9:
10: #ifdef __BORLANDC__
11: #pragma hdrstop
12: #endif
13:
14: #ifndef WX_PRECOMP
15: #include "wx/wx.h"
16: #endif
17:
18:
19: class MyApp: public wxApp
20: {
21:     virtual bool OnInit();
22: };
23:
24: class MyFrame: public wxFrame
25: {
26: public:
27:     MyFrame(const wxString & title,
28:             const wxPoint & pos,
29:             const wxSize & size);
30:     void OnQuit(wxCommandEvent & event);
31:     void OnGreet(wxCommandEvent & event);
32:     DECLARE_EVENT_TABLE()
33: private:
34:     wxPanel * m_panel;
35:     wxButton * m_btnGreet;
36:     wxButton * m_btnQuit;
37: };
38:
39: enum { ID_Quit = 1, ID_Greet };
40:
41: BEGIN_EVENT_TABLE(MyFrame, wxFrame)
42:     EVT_BUTTON(ID_Greet, MyFrame::OnGreet)
43:     EVT_BUTTON(ID_Quit, MyFrame::OnQuit)
44: END_EVENT_TABLE()
45:
46: IMPLEMENT_APP(MyApp)
47:
48: bool MyApp::OnInit()
49: {
50:     MyFrame *frame = new MyFrame("Hello World",
51:                                 wxDefaultPosition,
52:                                 wxDefaultSize);
53:     frame->Show(TRUE);
54:     SetTopWindow(frame);
55:     return TRUE;
56: }
57:
58: MyFrame::MyFrame(const wxString & title,
59:                   const wxPoint& pos,
60:                   const wxSize& size) :
61:     wxFrame((wxFrame *)NULL, -1, title, pos, size)
62: {
63:     wxSize panelSize = GetClientSize();
64:     int h = panelSize.GetHeight();
65:     int w = panelSize.GetWidth();
```

26

**LISTADO 26.4** CONTINUACIÓN

```

66:     m_panel = new wxPanel(this, -1, wxPoint(0, 0), panelSize);
67:
68:     m_btnGreet = new wxButton(m_panel,
69:                               ID_Greet,
70:                               _T("Greet"),
71:                               wxDefaultPosition,
72:                               wxDefaultSize);
73:     m_btnQuit = new wxButton(m_panel,
74:                               ID_Quit,
75:                               _T("Quit"),
76:                               wxDefaultPosition,
77:                               wxDefaultSize);
78:     wxMenu * menuApp = new wxMenu;
79:     menuApp ->Append(ID_Greet, "& Greet...");
80:     menuApp ->AppendSeparator();
81:     menuApp ->Append(ID_Quit, "&Quit");
82:
83:     wxMenuBar * menuBar = new wxMenuBar;
84:     menuBar->Append(menuApp, "&Application");
85:
86:     SetMenuBar(menuBar);
87: }
88:
89: void MyFrame::OnQuit(wxCommandEvent & WXUNUSED(event))
90: {
91:     Close(TRUE);
92: }
93:
94: void MyFrame::OnGreet(wxCommandEvent& WXUNUSED(event))
95: {
96:     wxMessageBox("Ejemplo de Hello World con wxWindows",
97:                 "Hello World",
98:                 wxOK | wxICON_INFORMATION,
99:                 this);
100: }
101:

```

**ANÁLISIS**

Tenemos exactamente el mismo código para empezar, y luego simplemente agregamos al código lo necesario para crear el menú y asociar los eventos del menú con los manejadores de eventos que ya tenemos.

En la línea 78 creamos un nuevo encabezado de menú en la barra de menú principal, y las líneas 79 a 81 comprenden el encabezado del menú y determinan lo que se verá al seleccionar el encabezado de menú y los elementos de menú que se muestren.

Luego creamos en la línea 83 una barra de menús, el objeto actual que se ve a lo largo de la parte superior del área cliente, y agregamos el encabezado de menú y sus elementos en la línea 84.

Por último, podemos configurar el menú principal de la aplicación para que sea la barra de menú que acabamos de crear en la línea 86.

Para compilar el programa, utilice el siguiente comando:

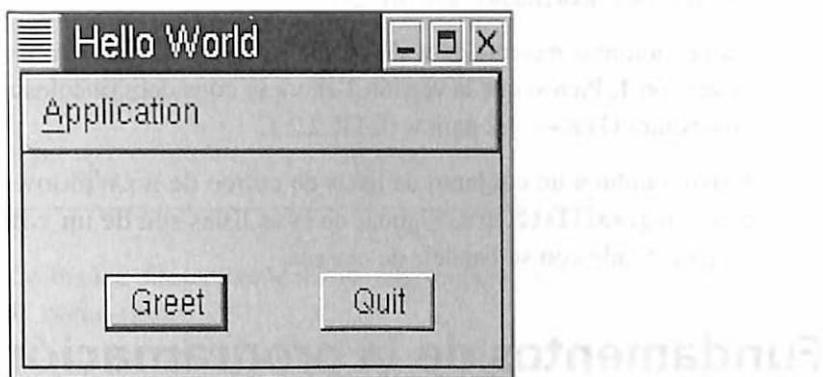
```
g++ l1t26-04.cxx `wx-config --libs` `wx-config --cflags` \ -o GNOMEHelloWorld
```

**SALIDA**

Al ejecutar el programa deberá ver una ventana igual a la que se muestra en la figura 26.3, pero ahora con un menú en la parte superior del área cliente de la ventana, como se muestra en la figura 26.4.

**FIGURA 26.4**

*El tercer programa GNOMEHelloWorld, con menú y botones.*



## Creación de aplicaciones wxWindows más complejas: wxStudio

Hasta ahora hemos creado una aplicación simple con un solo archivo fuente. Las aplicaciones más grandes que tengan varios archivos fuente que requieran un manejo más estrecho necesitarán obviamente algún tipo de entorno de desarrollo formal.

El proyecto wxWindows ha sugerido el wxStudio, su propio IDE (Entorno de Desarrollo Integrado). El trabajo con wxStudio se encuentra aún en sus primeras etapas, pero ya está liberado y está disponible mediante descarga electrónica gracias a CVS (Sistema de Versiones Concurrentes). Es muy interesante, cuando menos en plataformas Windows, que MS DevStudio funcione bien como IDE y como depurador para la biblioteca wxWindows; no sé cómo se comparan otros IDEs.

Como wxStudio se encuentra en una etapa muy temprana, es difícil hacer un comentario sobre este IDE, aparte de decir que el plan es para un IDE independiente de la plataforma y completo con depuradores, editores de recursos, control de versiones y mucho, mucho más. Uno de los dogmas fundamentales del diseño es que se proporcionarán tantos módulos funcionales como sea posible como complementos para permitir que el usuario configure el IDE con el editor, compilador y cualquier otra cosa de su elección.

## Cómo obtener wxWindows y otros recursos de wxWindows

La página Web de wxWindows está en <http://www.wxwindows.org/>. wxWindows viene en varias versiones, y proporciona el rango más amplio de plataformas soportadas que haya visto para un paquete de este tipo. El que usted utiliza con este libro es wxGTK, y la página de descarga para este software se encuentra en [http://www.wxwindows.org/dl\\_gtk.htm](http://www.wxwindows.org/dl_gtk.htm).

La versión más reciente de wxWindows es la 2.2.1 y es una actualización substancial de la versión 1. Pienso que la versión 1 ahora se considera obsoleta, si no es que obsoleta. Necesitará GTK++ 1.2 para wxGTK 2.2.1.

Existe también un conjunto de listas de correo de wxWindows en <http://www.wxwindows.org/mailist2.htm>. Algunas de estas listas son de un volumen muy grande, así que tenga cuidado con su bandeja de entrada.

## Fundamentos de la programación de KDE

La biblioteca Qt es la biblioteca de gráficos que sostiene a todo el KDE. Es un kit de herramientas de C++ desarrollado por la compañía TrollTech de Noruega, y ofrece elementos gráficos que se pueden utilizar para crear aplicaciones GUI para X.

Por añadidura, el kit de herramientas ofrece un conjunto completo de clases y métodos listos para usarse, incluso para código de programación que no involucre gráficos, así como un marco de trabajo sólido de interacción del usuario por medio de métodos virtuales y el exclusivo mecanismo de señal y ranura de Qt, y una biblioteca de elementos de GUI predefinidos, widgets de GUI que usted puede utilizar para crear los elementos visibles. También ofrece cuadros de diálogo predefinidos que se utilizan comúnmente en aplicaciones, como indicadores de progreso y cuadros de diálogo para archivos.

Por lo tanto, es muy conveniente que usted sepa cómo utilizar las clases de Qt, incluso si sólo quiere programar aplicaciones KDE usando el marco de trabajo de aplicación KDE. Vale la pena visitar el sitio de TrollTech que se encuentra en <http://www.troll.no/> para dar un vistazo a sus excelentes tutoriales y documentación.

Pero por ahora, para comprender los conceptos básicos de cómo crear aplicaciones GUI de Qt/KDE, dé un vistazo a un programa KDE de muestra.

## Creación de su primera aplicación KDE: "Hello World"

Para seguir con la tradición, daremos un vistazo (vea el listado 26.5) al programa Hello World estilo KDE y explicaremos cómo funciona.

**ENTRADA****LISTADO 26.5** El programa KDEHelloWorld

```
1: // Listado 26.5 Ejemplo simple de KDEHelloWorld
2:
3: #include <kapp.h>
4: #include <ktmainwindow.h>
5:
6: int main(int argc, char ** argv)
7: {
8:     KApplication MyApp(argc, argv);
9:     KTMainWindow * MyWindow = new KTMainWindow();
10:
11:    MyWindow->setGeometry(50, 50, 200, 100);
12:    MyApp.setMainWidget(MyWindow);
13:    MyWindow->show();
14:    return MyApp.exec();
15: }
```

**ANÁLISIS**

Esta aplicación básica simplemente dibuja una ventana vacía estilo KDE, con "Hello World" como título.

Compile estos archivos con los siguientes comandos en el indicador del sistema:

```
g++ -c -I$KDEDIR/include/kde -I$QTDIR/include -fno-rtti 1st26-25.cxx
g++ -L$KDEDIR/lib -lkdecore -lkdeui -lqt -o \
KDEHelloWorld 1st26-05.o
```

En esta aplicación, como en todas las aplicaciones KDE, primero se debe instanciar un objeto `KApplication`, en este caso representado por `MyApp` en la línea 8.

También pasamos los argumentos del programa, `argv` y `argc`, al constructor, y éste los utiliza antes de regresarlos, sin alteraciones, a `main()`.

Después de esto, instanciamos un objeto `KTMainWindow` y lo llamamos `MyWindow` en la línea 9; como el nombre lo implica, ésta será la ventana que eventualmente se verá desplegada en la pantalla. Damos tamaño y colocamos la ventana en la pantalla con la llamada a `setGeometry()` en la línea 11. La movemos hacia las coordenadas (50, 50) y cambiamos el tamaño a 200 × 100 píxeles (ancho × altura).

Por último, llamamos a la función miembro `setMainWidget()` en `MyApp`, seguida del método `show()` en `MyWindow`, en las líneas 12 y 13, respectivamente. Esto indica a `KApplication` que su widget principal u objeto de despliegue es la ventana que acabamos de crear; al llamar a `show()` indicamos a la ventana que se haga visible.

Para ejecutar el código de `KApplication` y para dibujar la ventana principal, llamamos a la función miembro `exec()` en `MyApp` en la línea 14.

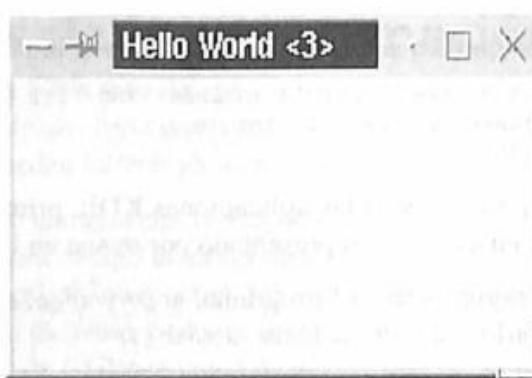
Llamar a `exec()` ocasiona que el programa entre al ciclo principal de eventos y que espere hasta que éste regrese un entero al sistema operativo, señalando así que la aplicación ha terminado.

Lo principal aquí es que la aplicación entra a un "ciclo principal de eventos". Esto significa que el programa tiene que esperar las acciones del usuario y luego reaccionar a ellas; lo que es más, con una aplicación KDE el programa tiene que estar en el ciclo principal de eventos para empezar el manejo de los mismos.

**SALIDA** La figura 26.5 le muestra cómo debe lucir la aplicación `KDEHelloWorld` al ejecutarla. Tenga en cuenta que tal vez haya algunas diferencias de tipo cosmético, pues tal vez usted haya elegido una "composición" del escritorio o combinación de colores diferentes de los que yo tengo.

**FIGURA 26.5**

*El primer programa `KDEHelloWorld`.*



## Creación de su propia clase de ventana de KDE

El ejemplo que acaba de ver es probablemente el programa KDE más simple que se pueda escribir; permítanos extenderlo un poco y derivar nuestra propia clase de ventana de la clase `KTMainWindow`.

Ahora podemos cambiar los archivos fuente del proyecto para que contengan el código que se muestra en el listado 26.6. Tome en cuenta que no podemos utilizar en forma razonable un solo archivo fuente, ya que tenemos que ejecutar el MOC (Compilador de metaobjetos) de Qt en el archivo de encabezado.

**ENTRADA****LISTADO 26.6A** El archivo de encabezado para el programa KDEHelloWorld

```

1: // Listado 26.6a Definición de la clase KDEHelloWorld (lst26-06.hpp)
2:
3: #include <kapp.h>
4: #include <ktmainwindow.h>
5:
6: class KDEHelloWorld : public KTMaiNWindow
7: {
8:     Q_OBJECT;
9: public:
10:    void closeEvent(QCloseEvent * );
11: };

```

**ENTRADA****LISTADO 26.6B** El programa KDEHelloWorld con una clase de ventana derivada

```

1: // Listado 26.6b KDEHelloWorld (lst26-06.cxx)
2:
3: #include "lst26-06.moc"
4: #include "lst26-06.hpp"
5:
6: void KDEHelloWorld::closeEvent(QCloseEvent * )
7: {
8:     kapp->quit();
9: }
10:
11: int main(int argc, char ** argv)
12: {
13:     KApplication MyApp(argc, argv, "Hello World");
14:     KDEHelloWorld * MyWindow = new KDEHelloWorld();
15:
16:     MyWindow ->setGeometry(50, 100, 200, 100);
17:     MyApp.setMainWidget(MyWindow);
18:     MyWindow ->show();
19:     return MyApp.exec();
20: }
21:

```

26

Compile estos archivos con los siguientes comandos en el indicador del sistema:

```

$QTDIR/bin/moc lst26-06.hpp -o lst26-06.moc
g++ -c -I$KDEDIR/include/kde -I$QTDIR/include -fno-rtti lst26-06.cxx
g++ -L$KDEDIR/lib -lkdecore -lkdeui -lqt -o KDEHelloWorld \
lst26-06.o

```

**ANÁLISIS**

Cómo puede ver en el código anterior, el cambio más notable es la macro extraña `Q_OBJECT` en la línea 8 del listado 26.6a, en la declaración de clase de `KDEHelloWorld`, y la directiva `#include` no muy conocida en la línea 3 del listado 26.6b.

Verá exactamente para qué son estos cambios más adelante en la lección. Por ahora, basta con saber que debe incluir la macro `Q_OBJECT` en *todas* sus clases derivadas de KDE. La razón de esto es que el MOC (Compilador de metaobjetos) utiliza la macro `Q_OBJECT` para crear un archivo `.moc` que, al ser incluido mediante la instrucción `#include`, permite la comunicación entre los widgets de KDE por medio del mecanismo de señales y ranuras de Qt.

También observará que en las líneas 6 a 9 del listado 26.6b se implementa el manejador de eventos para el objeto `QCloseEvent`. Los eventos y los manejadores de eventos son lo que la aplicación KDE utiliza para determinar cómo está interactuando el usuario. En el caso anterior, `QCloseEvent *` ocasiona que la aplicación termine. Si quisieramos, podríamos agregar código para preguntar al usuario si ésta era la acción deseada, y así sucesivamente. Vea el recuadro “Los eventos como objetos QEvent”.

La principal diferencia que verá al ejecutar la aplicación es que el título de la ventana ha cambiado de “KDEHelloWorld” a “Hello World” ya que en este segundo ejemplo especificamos el título para la aplicación en el constructor de `KApplication`.

#### **Los eventos como objetos QEvent**

`KApplication` envía eventos a la ventana activa como objetos `QEvent`, y entonces los mismos widgets tienen qué decidir qué hacer con ellos.

Un widget recibe el `QEvent` y llama a `QWidget::event(QEvent*)`, que luego decide cuál evento ha sido detectado y cómo reaccionar; `event()` es por lo tanto el manejador principal de eventos.

La función `event()` pasa el objeto `QEvent` a los filtros de eventos, los cuales determinan qué ocurrió y qué hacer con el evento. Si ningún filtro es responsable del manejo de un evento, se llama a los manejadores especializados de eventos.

En la documentación de Qt verá que todos los manejadores de eventos son funciones virtuales que se declaran como protegidas; por lo tanto, puede redefinir los eventos que necesite en sus propios widgets y especificar cómo tiene que reaccionar su widget.

`QWidget` también contiene algunos otros métodos virtuales que pueden ser útiles en sus programas.

## **Cómo agregar botones a su propia clase de ventana de KDE**

Hasta ahora, estas aplicaciones KDE han sido un poco simples, por no decir más; tal vez tengamos una GUI, pero no la estamos aprovechando mucho.

En este tercer ejemplo expandiremos el código que tenemos y agregaremos un par de botones a la aplicación. Al mismo tiempo, también introduciremos un concepto importante que sostiene todo el marco de trabajo de aplicación de Qt (y también de KDE): las señales y ranuras.

Primero modificaremos nuestra aplicación “Hello World” para utilizar el mecanismo de señales y ranuras antes de profundizar más en su funcionamiento.

Puede ver el código fuente modificado y los archivos de encabezado en el listado 26.7.

**ENTRADA LISTADO 26.7A Los archivos de encabezado para el programa KDEHelloWorld**

```
1: // Listado 26.7a Declaración de la clase KDEHelloWorld
2:
3: #include <kapp.h>
4: #include <ktmainwindow.h>
5: #include <qpushbutton.h>
6:
7: class KDEHelloWorld : public KTMainWindow
8: {
9:     Q_OBJECT
10:    public:
11:        KDEHelloWorld();
12:        void closeEvent(QCloseEvent * );
13:    public slots:
14:        void SlotGreet();
15:        void SlotQuit();
16:    private:
17:        QPushButton * m_btnGreet;
18:        QPushButton * m_btnQuit;
19:    };
20:
```

**ENTRADA LISTADO 26.7B El programa KDEHelloWorld con botones**

```
1: // Listado 26.7b Otro ejemplo de KDEHelloWorld
2:
3: #include "lst26-07.moc"
4: #include <kmsgbox.h>
5:
6: KDEHelloWorld::KDEHelloWorld() : KTMainWindow()
7: {
8:     m_btnGreet = new QPushButton("Greet", this);
9:     m_btnGreet->setGeometry(45, 30, 50, 20);
10:    m_btnGreet->show();
11:    connect(m_btnGreet,
12:            SIGNAL(clicked()),
13:            this,
14:            SLOT(SlotGreet()));
15:
16:    m_btnQuit = new QPushButton("Quit", this);
17:    m_btnQuit->setGeometry(105, 30, 50, 20);
18:    m_btnQuit->show();
19:    connect(m_btnQuit,
20:            SIGNAL(clicked()),
21:            this,
22:            SLOT(SlotQuit()));
23: }
24:
```

**LISTADO 26.7B CONTINUACIÓN**

```
25: void KDEHelloWorld::closeEvent(QCloseEvent *)
26: {
27:     kapp->quit();
28: }
29:
30: void KDEHelloWorld::SlotGreet()
31: {
32:     KMessageBox::message(0,"Hello World con KDE","Hello World");
33: }
34:
35: void KDEHelloWorld::SlotQuit()
36: {
37:     close();
38: }
39:
40: int main(int argc, char ** argv)
41: {
42:     KApplication MyApp(argc, argv, "Hello World");
43:     KDEHelloWorld * MyWindow = new KDEHelloWorld();
44:
45:     MyWindow ->setGeometry(50, 100, 200, 100);
46:     MyApp.setMainWidget(MyWindow);
47:     MyWindow ->show();
48:     return MyApp.exec();
49: }
50:
```

---

Compile estos archivos con los siguientes comandos en el indicador del sistema:

```
$QTDIR/bin/moc lst26-07.hpp -o lst26-07.moc
g++ -c -I$KDEDIR/include/kde -I$QTDIR/include/ -fno-rtti lst26-07.cxx
g++ -L$KDEDIR/lib -lkdecore -lkdeui -lqt -o \
KDEHelloWorld lst26-07.o
```

**ANÁLISIS** Ahora puede ver que todo está empezando a movilizarse. Tenemos en el código varios elementos nuevos que merecen una explicación.

Lo primero y más simple que debe observar es la llamada a `KMessageBox::message()` en la línea 32 del listado 26.7b.

Esto coloca en la pantalla un simple cuadro de diálogo predefinido que da algo de información al usuario; en este caso no dice más que “¡Hola, mundo!”, pero lo puede configurar para que muestre cualquier mensaje que desee. El primer parámetro de cadena, “Hola mundo con KDE”, establece el título para el cuadro, y el segundo “¡Hola, mundo!”, es el mensaje en sí.

Lo segundo y más importante que debe observar es la rara combinación de palabras reservadas `public slots:` en el listado 26.7a. Ésta no es una declaración estándar de C++, sino que forma parte del mecanismo de señales y ranuras de la biblioteca Qt; en esencia, le indica al compilador de meta objetos que estas funciones pueden ser llamadas como

respuesta a las señales que se emitan. Si analiza la implementación del constructor de `KDEHelloWorld`, `KDEHelloWorld::KDEHelloWorld()`, verá que las llamadas a `connect()` de las líneas 11 y 19 del listado 26.7b actúan para asociar la señal `clicked()` con los miembros `SlotGreet()` y `SlotQuit()` de la clase `KDEHelloWorld` (vea la figura 26.6).

En esencia, lo que esto significa es que, al hacer clic en el widget `m_btnGreet`, se envía una señal `clicked()` y la biblioteca Qt invocará a la función miembro `KDEHelloWorld::SlotGreet()` por usted y desplegará un mensaje. De la misma manera, al hacer clic en el widget `m_btnQuit`, se enviará una señal `clicked()` y la biblioteca Qt invocará a la función miembro `KDEHelloWorld::SlotQuit()` por usted y cerrará la ventana.

**FIGURA 26.6**

*El segundo programa `KDEHelloWorld`, con botones.*



Las señales y ranuras son los elementos centrales del marco de trabajo de programación de KDE y por lo tanto merecen una explicación más detallada.

**26**

## Interacción de objetos por medio de señales y ranuras

El mecanismo de señales y ranuras ofrece una solución muy poderosa y útil al problema de interacción de objetos, que por lo general se resuelve mediante el uso de funciones callback en la mayoría de los kits de herramientas de X Windows.

Si las callbacks no se utilizan de manera correcta, pueden propiciar errores y con frecuencia conduce al temible fallo de segmentación, por lo que requieren un estricto protocolo de programación, y algunas veces hacen que la creación de la interfaz de usuario sea muy difícil. TrollTech ideó un nuevo sistema en el que los objetos pueden emitir señales que se pueden conectar con métodos declarados como ranuras.

El mecanismo de señales y ranuras es una característica central de Qt y es probablemente el elemento que más distingue al kit de herramientas Qt de los demás kits.

En la mayoría de los kits de herramientas para GUI los widgets tienen una callback para cada acción que puedan activar. Esta callback es un apuntador a una función. En Qt, las señales y las ranuras se han encargado del trabajo de estos apuntadores a funciones propensos a errores.

Las señales y las ranuras pueden tomar cualquier número de argumentos de cualquier tipo. Ofrecen una completa *seguridad de tipos*: ¡Al menos nos librados de los vacíos del kernel provocados por las callback!

Todas las clases que heredan de `QObject` o de una de sus subclases (por ejemplo, de `QWidget`) pueden contener señales y ranuras. Los objetos emiten señales cuando cambian su estado en una forma que podría ser interesante para el mundo exterior. Esto es todo lo que el objeto hace para comunicarse. No sabe si hay algo que reciba la señal en el otro extremo; además, ni siquiera le importa. Ésta es una verdadera encapsulación de datos, y asegura que se pueda utilizar el objeto como un componente de software.

Las ranuras pueden recibir señales, pero, por lo demás, son como cualquier función miembro normal. Una ranura no sabe ni le importa si tiene una o más señales conectadas; es decir, el objeto no sabe nada acerca del mecanismo de comunicación y se puede utilizar como un verdadero componente de software. Puede conectar tantas señales como desee en una sola ranura, y puede conectar una señal a todas las ranuras que desee.

Un objeto emitirá una señal cuando su estado interno haya cambiado de alguna forma que pueda ser relevante para el cliente o el propietario del objeto. Sólo la clase que define una señal y sus subclases pueden emitir la señal.

Por ejemplo, un widget de cuadro de lista emite las señales `highlighted()` y `activated()`. Probablemente, la mayoría de los objetos sólo estarían interesados en `activated()`, pero tal vez algunos querrían saber cuál elemento del cuadro de lista está resaltado actualmente. Si la señal es relevante para dos objetos diferentes, puede conectarla en las ranuras de ambos objetos.

Cuando un objeto emite una señal, las ranuras conectadas a éste se ejecutan igual que una llamada normal a una función. El mecanismo de señales y ranuras es totalmente independiente de cualquier ciclo de eventos de la GUI. La emisión terminará cuando todas las ranuras hayan terminado; si varias ranuras están conectadas a una señal, éstas se ejecutarán una después de la otra, en orden arbitrario, cuando se emita la señal.

El MOC genera automáticamente el código que emite señales y usted no debe implementarlas por su cuenta en el archivo `.cxx`. Todas las señales tienen el tipo de valor de retorno `void`. Usted, el programador de la aplicación, tiene el trabajo de implementar las ranuras por su propia cuenta.

Como las ranuras son funciones miembro normales que se pueden invocar en formas misteriosas de las que no necesitamos preocuparnos aquí, tienen derechos de acceso al igual que cualquier otro miembro de la clase. No es sorprendente que el derecho de acceso de una ranura determine quién puede conectarse a ella:

- Una sección `public slots`: contiene ranuras a las que cualquiera puede conectar señales. Usted crea objetos que no saben nada unos acerca de otros, conecta sus señales y ranuras, y pasa información entre ellos.

- Una sección **protected slots**: contiene ranuras a las que esta clase y sus subclases pueden conectar señales. Esto se utiliza en las ranuras que son parte de la implementación de la clase, en lugar de su interfaz **public externa**.
- Una sección **private slots**: contiene ranuras a las que sólo los objetos instanciados de esa misma clase pueden conectar señales. Esto se utiliza en clases conectadas en forma muy estrecha, en donde ni siquiera se confía que las subclases hagan bien las conexiones.

Desde luego que también puede definir las ranuras como virtuales. Esta característica es muy útil.

Como ejemplo de una implementación típica de señales y ranuras, considere la siguiente declaración mínima de una clase de C++:

```
01: class MiClase
02: {
03:     public:
04:         MiClase();
05:         char Letter() const { return m_cVal; }
06:         void SetValue(char cVal);
07:     private:
08:         char m_cVal;
09: };
```

Una pequeña clase Qt se podría declarar de la siguiente manera:

```
01: class MiClaseQt : public QObject
02: {
03:     Q_OBJECT // observe que no hay punto y coma después de esta macro
04:     public:
05:         MiClaseQt();
06:         char Letter() const { return m_cVal; }
07:     public slots:
08:         void SetLetter (char cVal);
09:     signals:
10:         void LetterChanged(char cVal);
11:     private:
12:         char m_cVal;
13: };
```

26

Esta clase es esencialmente la misma, pero además de la funcionalidad básica de la clase simple, también tiene soporte para la programación de componentes por medio de señales y ranuras: esta clase puede decir al mundo que ha cambiado su estado emitiendo una señal, **LetterChanged()**, y tiene una ranura a la que los otros objetos pueden enviar señales. Hemos indicado a MOC que proporcione el código para hacer esto por medio de la macro **Q\_OBJECT** en la línea 3 y del especificador **signals** en la línea 9.

Todas las clases que contienen señales, ranuras o ambas, también deben contener la macro **Q\_OBJECT** en su declaración.

He aquí una posible implementación de `MiClaseQt::SetLetter()`:

```

01: void MiClaseQt::SetLetter(char cVal)
02: {
03:     if (cVal != m_cVal)
04:     {
05:         m_cVal = cVal;
06:         emit LetterChanged(cVal);
07:     }
08: }
```

La línea 6 emite la señal `LetterChanged()` del objeto. Como puede ver, se emite una señal usando la llamada a `emit señal(argumentos)`.

Si quisiéramos conectar dos instancias del objeto `MiClase`, podríamos escribir lo siguiente:

```

01: int main(int argc, char** argv)
02: {
03:     MiClaseQt Alice, Bob;
04:     connect(&Alice, SIGNAL(LetterChanged(char)), &Bob, SLOT(SetLetter(
05:     ↳char)));
06:     Bob.SetLetter('a');
07:     Alice.SetLetter('b');
08:     Bob.Letter();           // ¿ qué seria esto ahora? 'b', desde luego
09:     return 0;
09: }
```

Lamar a `Alice.SetLetter('b')` hará que Alice emita una señal, la cual será recibida por Bob; es decir, invoca a `Bob.SetLetter('b')`. A su vez, Bob emitirá la misma señal, la cual nadie recibe ni le presta atención ya que no le hemos conectado una ranura. Por lo tanto, la señal emitida por Bob se desvanece para no ser vista nunca más.

Una analogía muy parecida en el mundo de las funciones callback sería un apuntador NULL a una callback (lo que, por lo general, produciría un vaciado del kernel y un programador deprimido).

Observe que la función `SetLetter()` establece el valor y emite la señal sólo si `cVal != m_cVal`. Esto previene ciclos infinitos en el programa donde, por ejemplo, `Bob..LetterChanged()` estaba conectada con `Alice.SetLetter()`.

Las señales y ranuras son eficientes en una forma razonable, pero son nominalmente más lentas que las “verdaderas” funciones callback debido a la naturaleza flexible y dividida en niveles de la llamada; sin embargo, la perdida es aceptablemente pequeña para todas las aplicaciones, excepto las que requieran de un uso más crítico del tiempo.

Para implementar el mecanismo de señales y ranuras de Qt, tiene que compilar el código fuente con el MOC (Compilador de metaobjetos), el cual forma parte de la biblioteca Qt. El MOC analiza sintácticamente la declaración de la clase de un archivo de C++ y genera código de C++ que inicializa el metaobjeto. Éste contiene los nombres de todos los miembros de señales y ranuras, así como de apuntadores a estas funciones. El preprocesador se encarga en forma transparente de las palabras reservadas `signal`, `slot` y `emit`, para que el compilador de C++ no vea algo que no pueda digerir.

## Cómo agregar un menú a su propia clase de ventana de KDE

Por último, daremos a nuestra aplicación KDE simple una última capa de respeto, agregando un menú a la ventana.

Primero necesitamos modificar el código fuente para que se vea igual que el listado 26.8.

### ENTRADA LISTADO 26.8A El archivo de encabezado para el programa KDEHelloWorld

```
1: // Listado 26.8a Declaración de la clase KDEHelloWorld
2:
3: #include <kapp.h>
4: #include <ktmainwindow.h>
5: #include <qpushbutton.h>
6: #include <kmenubar.h>
7: #include <qpopupmenu.h>
8:
9: class KDEHelloWorld : public KTMainWindow
10: {
11:     Q_OBJECT
12: public:
13:     KDEHelloWorld();
14:     void closeEvent(QCloseEvent * );
15: public slots:
16:     void SlotGreet();
17:     void SlotQuit();
18: private:
19:     QPushButton * m_btnGreet;
20:     QPushButton * m_btnQuit;
21:     KMenuBar * m_Menu;
22:     QPopupMenu * m_MenuApp;
23: };
```

26

### ENTRADA LISTADO 26.8B El programa KDEHelloWorld con botones y un menú

```
1: // Listado 26.8b KDEHelloWorld
2:
3: #include "lst26-08.moc"
4: #include <kmsgbox.h>
5:
6: KDEHelloWorld::KDEHelloWorld() : KTMainWindow()
7: {
8:     m_btnGreet = new QPushButton("Greet", this);
9:     m_btnGreet->setGeometry(45, 30, 50, 20);
10:    m_btnGreet->show();
11:    connect(m_btnGreet,
12:             SIGNAL(clicked()),
13:             this,
```

**LISTADO 26.8B CONTINUACIÓN**

---

```
14:         SLOT(SlotGreet()));
15:
16:     m_btnQuit = new QPushButton("Quit", this);
17:     m_btnQuit->setGeometry(105, 30, 50, 20);
18:     m_btnQuit->show();
19:     connect(m_btnQuit,
20:             SIGNAL(clicked()),
21:             this,
22:             SLOT(SlotQuit()));
23:
24:     m_MenuApp = new QPopupMenu();
25:     m_MenuApp->insertItem("&Greet",
26:                           this,
27:                           SLOT(SlotGreet()));
28:     m_MenuApp->insertItem("&Quit",
29:                           this,
30:                           SLOT(SlotQuit()));
31:     m_Menu = new KMenuBar(this);
32:     m_Menu->insertItem("&Application", m_MenuApp);
33: }
34:
35: void KDEHelloWorld::closeEvent(QCloseEvent *)
36: {
37:     kapp->quit();
38: }
39:
40: void KDEHelloWorld::SlotGreet()
41: {
42:     KMessageBox::message(0,"Hello World con KDE","Hello World");
43: }
44:
45: void KDEHelloWorld::SlotQuit()
46: {
47:     close();
48: }
49:
50: int main(int argc, char ** argv)
51: {
52:     KApplication MyApp(argc, argv, "Hello World");
53:     KDEHelloWorld * MyWindow = new KDEHelloWorld();
54:
55:     MyWindow ->setGeometry(50, 100, 200, 100);
56:     MyApp.setMainWidget(MyWindow);
57:     MyWindow ->show();
58:     return MyApp.exec();
59: }
60:
```

---

Compile estos archivos con los siguientes comandos en el indicador del sistema:

```
g++ -c -I$KDEDIR/include/kde/ -I$QTDIR/include/ -fno-rtti main.cxx
$QTDIR/bin/moc lst26-08.hpp -o lst26-08.moc
```

```
g++ -c -I$KDEDIR/include/kde -ISQTDIR/include -fno-rtti lst26-08.cxx
g++ -L$KDEDIR/lib -lkdecore -lkdeui -lqt -o KDEHelloWorld main.o \
lst26-08.o
```

**ANÁLISIS**

Lo primero que debe observar son las variables miembro adicionales que representan los objetos del menú que hemos agregado a la clase `KDEHelloWorld` en las líneas 21 y 22 del listado 26.8a.

Los nombres de las variables se explican por sí solos, al igual que el código de inicialización en el constructor `KDEHelloWorld` que abarca las líneas 24 a 32 del listado 26.8b.

**SALIDA**

Al ejecutar este programa, deberá ver algo parecido a la pantalla que se muestra en la figura 26.7.

**FIGURA 26.7**

*El tercer programa `KDEHelloWorld`, con menú y botones.*



## Creación de aplicaciones KDE más complejas: KDevelop

Las aplicaciones de ejemplo que ha visto hasta ahora son simples y sirven sólo para ilustrar los conceptos esenciales del marco de trabajo de aplicaciones KDE. Pero para aplicaciones más complicadas, alternar entre muchos archivos fuente y navegar por las clases y otras cosas más es mucho trabajo; de cualquier forma, ningún escritorio GUI sería respetable sin un IDE (Entorno de Desarrollo Integrado) de GUI, y KDE no es la excepción.

El IDE KDevelop proporciona muchas características que los desarrolladores necesitan. Además, envuelve la funcionalidad de proyectos de terceros, como `make` y los compiladores GNU de C++, y los convierte en una parte invisible e integrada del proceso de desarrollo. KDevelop ofrece las siguientes características:

- Todas las herramientas de desarrollo necesarias para la programación en C++, como el compilador, el enlazador, automake y autoconf.
- KAppWizard, que genera aplicaciones de ejemplo completas.
- ClassGenerator, que crea nuevas clases y las integra en el proyecto actual.
- Administración de archivos para archivos fuente, archivos de encabezado, documentación del proyecto, etc.

- La creación de manuales de usuario escritos con SGML y la generación automática de salida en HTML con la look and feel de KDE.
- Documentación automática de API, basada en HTML para las referencias cruzadas de las clases de su proyecto con las bibliotecas utilizadas.
- Soporte internacional para su aplicación, lo cual le permite agregar fácilmente sus nuevos lenguajes a un proyecto.
- Creación WYSIWYG de interfaces de usuario con un editor de cuadros de diálogo integrado.
- Administración de proyectos mediante CVS y la provisión de una interfaz fácil de usar para las funciones más necesarias.
- Depuración de programas por medio de KDbg.
- Edición de iconos por medio de KIconEdit.
- La inclusión de cualquier otro programa que necesite para el desarrollo, agregándolo al menú Herramientas de acuerdo con sus necesidades individuales.

KDevelop facilita el trabajo con todos los programas en un solo lugar y ahorra tiempo al automatizar los procesos estándar de desarrollo, además de proporcionar un acceso directo y transparente a toda la información que necesite para controlar su proyecto. Los mecanismos de exploración integrados están diseñados para soportar solicitudes de documentación que tengan los desarrolladores junto con su proyecto.

El visor de clases y el buscador de errores lo llevan a cualquier parte del código del proyecto con sólo hacer clic con el ratón, sin necesidad de buscar archivos. Los árboles de archivos le proporcionan un acceso directo a los archivos del proyecto, y el sistema de ayuda integrado le ofrece un acceso excelente a la documentación en línea desde cualquier parte del IDE.

En resumen, KDevelop es un IDE poderoso y completo en características. ¿Ya le mencioné que es gratuito?

Al momento de escribir esto, y debido a que KDevelop 1.3 utiliza KDE 2.0, me estoy refiriendo al estado de las bibliotecas de KDE en relación con esa liberación. Las principales bibliotecas de KDE que usted utilizará para crear sus propias aplicaciones KDE son

- La biblioteca Core de KDE, que contiene todas las clases que son elementos no visibles y que proporcionan la funcionalidad que su aplicación puede utilizar.
- La biblioteca UI de KDE, que contiene elementos de interfaz de usuario, como barras de menús, barras de herramientas y demás cosas relacionadas.
- La biblioteca KFile, que contiene los cuadros de diálogo de selección de archivos.

Además, KDE ofrece las siguientes bibliotecas para soluciones específicas:

- La biblioteca KHTMLW, que ofrece un widget completo para interpretar HTML que se utiliza en varios programas, como KDEHelp, KFM y Kdevelop.
- La biblioteca KFM, que le permite utilizar el administrador de archivos de KDE desde el interior de su aplicación.
- La biblioteca KAb, o KAddressBook. Proporciona acceso a la libreta de direcciones, por ejemplo para aplicaciones de correo electrónico.
- La biblioteca KSpell, que ofrece widgets y funcionalidad para integrar el uso de Ispell (el revisor ortográfico común) en aplicaciones tales como editores; se utiliza para la aplicación KEdit

## Resumen

Finalmente Linux ha crecido. Los dos nuevos escritorios que ha visto hoy, GNOME y KDE, así como sus bibliotecas y aplicaciones de soporte, le proporcionan una selección de GUIs gratuitas, modernas, intuitivas y robustas para el sistema operativo Linux. También le proporcionan un conjunto de poderosos marcos de trabajo de desarrollo, entornos y herramientas que eliminan el trabajo pesado de la escritura de aplicaciones de GUI en X y le permiten concentrarse en lo que su aplicación debe estar haciendo, en lugar de involucrarlo en una lucha que no puede ganar contra una API intratable.

Con las nuevas interfaces GUI para Linux, tiene una opción genuina, y eso sólo puede ser bueno para todos, ya que se verá una notable mejora en la calidad y el precio (especialmente porque los escritorios Linux son gratuitos) en todo el mercado de sistemas operativos.

Todos los paquetes y bibliotecas que he descrito en esta sección están siendo extendidos y mejorados casi a diario por voluntarios, y están apareciendo nuevos casi con la misma frecuencia.

Sus esfuerzos y habilidad han colocado firmemente a Linux donde pertenece: en el escritorio.

26

## Preguntas y respuestas

**P ¿Para qué necesitamos KDE y GNOME?**

**R** La razón es que Linux, a diferencia de Windows y MacOS, no tiene una GUI nativa: usted no puede interactuar con ella por medio de comandos textuales. La computadora en sí tiene capacidad para gráficos (por lo general), pero Linux por sí solo no sabe cómo manejar esta capacidad. GNOME y KDE le proporcionan un marco de trabajo para que interactúe con la computadora en forma gráfica, y para que sus comandos se dirijan en forma transparente al sistema operativo. En realidad, no necesitamos ambos: son lo suficientemente distintos como para coexistir, y el tiempo dirá cuál de los dos se hará más popular.

- P ¿Por qué no puedo simplemente utilizar CDE (Entorno Común de Escritorio) o Motif como mi GUI?**
- R** Puede hacerlo si lo desea. La belleza de Linux es que es gratuito, y si utiliza software de GNU o de KDE, éste también es gratuito. *Puede* obtener versiones de CDE y de Motif para Linux, pero son software propietario y tiene que pagar por ellos. No sólo eso, sino que para muchas personas, las GUIs proporcionadas por CDE y Motif se están empezando a ver un poco anticuadas.
- P Si estoy escribiendo una aplicación GUI para GNOME, ¿podré ejecutarla en KDE, y viceversa?**
- R** Tal vez. Si escribe aplicaciones para un escritorio, podrá ejecutarlas en el otro si y sólo si tiene en su equipo las bibliotecas correctas de tiempo de ejecución. Por otra parte, si funcionan tal vez no se comporten como usted espera y quiera que se comporten, ya que la información de sesión persistente de una aplicación GNOME será ignorada por el escritorio KDE (y esto se aplica también en forma inversa). Lo que es más importante, en algunas instalaciones, las aplicaciones GNOME, al igual que GIMP, perturban la combinación de colores de KDE cuando tienen el enfoque. En mi equipo, cuando ejecuto Red Hat Linux 6.1, si ejecuto GIMP en KDE, toda la pantalla se vuelve azul cuando GIMP tiene el enfoque. Si lo que quiere es una aplicación genérica, siempre podrá utilizar wxWindows, que utiliza GTK++ pero no utiliza GNOME.
- P ¿Qué es el código multiplataforma? ¿Por qué podría ser importante?**
- R** Es código que se escribe una sola vez y se compila usando el kit de herramientas adecuado en el equipo de destino. La biblioteca Qt y wxWindows son “independientes de la plataforma”, lo que significa que el código fuente para una aplicación dada es el mismo, sin importar en qué plataforma se compila; obviamente, las bibliotecas de los kits de herramientas son considerablemente distintas, pero eso no debe preocuparle. Ésta es el área en la que algunas personas piensan que KDE se derrumba: Las bibliotecas Qt tienen una licencia gratuita sólo para Linux y otros sistemas UNIX. Si quiere crear software para Windows usando las bibliotecas Qt, tiene que comprar una licencia.
- P Quiero utilizar C++, pero no quiero usar wxWindows ni GTK, ni cualquier otra envoltura. ¿Puedo utilizar C++ directamente en las bibliotecas de GNOME/GTK++?**
- R** Sí. Aunque esto probablemente produzca código más revuelto, puede declarar sus funciones callback como miembros estáticos de sus clases. Entonces puede implementar en cualquier forma que guste las clases que tienen los miembros callback estáticos y hacer que hagan lo que usted quiera.

## Taller

El taller le proporciona un cuestionario para ayudarlo a afianzar su comprensión del material tratado, así como ejercicios para que experimente con lo que ha aprendido. Trate de responder el cuestionario y los ejercicios antes de ver las respuestas en el apéndice D, “Respuestas a los cuestionarios y ejercicios”, y asegúrese de comprender las respuestas antes de pasar al siguiente día.

### Cuestionario

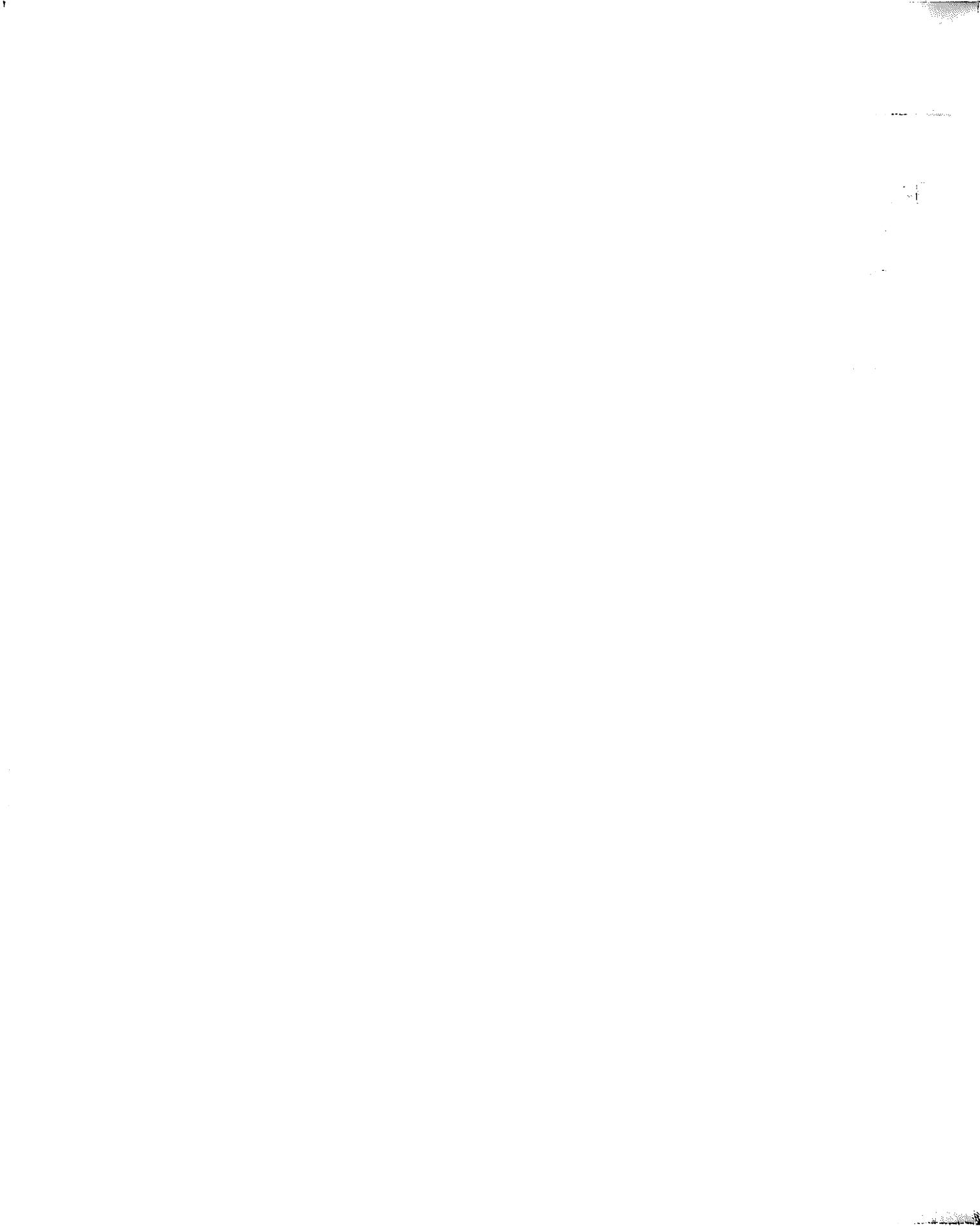
1. ¿Cuál es la diferencia entre programas controlados por eventos y programas controlados por procedimientos?
2. ¿Qué es un widget, en términos computacionales?
3. ¿Qué son las funciones callback, y por qué son propensas a errores?
4. ¿Qué es una ranura Qt, y cómo reacciona a las señales Qt?
5. Las señales y las ranuras ofrecen seguridad de tipos. ¿Qué significa esto y por qué es algo bueno?
6. ¿Cómo enlazaría el siguiente manejador de eventos de wxWindows a un evento EVT\_MENU desde un elemento de menú con el identificador de ID\_MY\_HELP? Asuma que `OnHelp()` es miembro de la clase MyFrame, y que MyFrame se deriva de wxFrame.

```
void OnHelp(wxCommandEvent & WXUNUSED(event));
```

7. ¿Qué hace la macro `IMPLEMENT_APP()`?
8. ¿Qué hace la macro `Q_OBJECT`?

### Ejercicios

1. Usando el archivo fuente `lst26-01.cxx` como base, extienda el programa para que la clase `callback` muestre un cuadro de diálogo al hacer clic en los botones, en lugar de escribir directamente en `stdout`.
2. Tomando como base el ejemplo final de wxWindows o el de KDE, extienda el código para que despliegue un botón de exploración de archivos que le permita seleccionar un archivo por medio de un cuadro de diálogo estándar de selección de archivos.
3. Extienda el código del ejercicio 2 de forma que, al hacer clic en OK y seleccionar el archivo, el programa lo despliegue en un control de texto en la ventana principal.
4. Analice los resultados del ejercicio 1 y considere cómo podría envolver toda la aplicación en una sola clase. Considere por qué podría necesitar implementar las funciones callback como miembros estáticos privados de la clase, y exponer sólo “envolturas” que las llamen cuando el usuario active eventos. Vea por ejemplo la función `do_message()` del archivo `lst26-01.cxx`. Piense cómo podría extender el concepto para crear una envoltura genérica para la aplicación, y utilizar funciones virtuales para configurar la ventana principal y conectar señales.



22

# SEMANA 4

23

## Repaso

¡Debe estar orgulloso de usted mismo! No sólo completó los 21 días para aprender C++ para Linux, sino que también completó los cinco días de la semana adicional. Esa semana cubrió los siguientes temas:

- El entorno de programación de Linux
- Programación shell
- Programación de sistemas
- Comunicación entre procesos
- Programación de la GUI

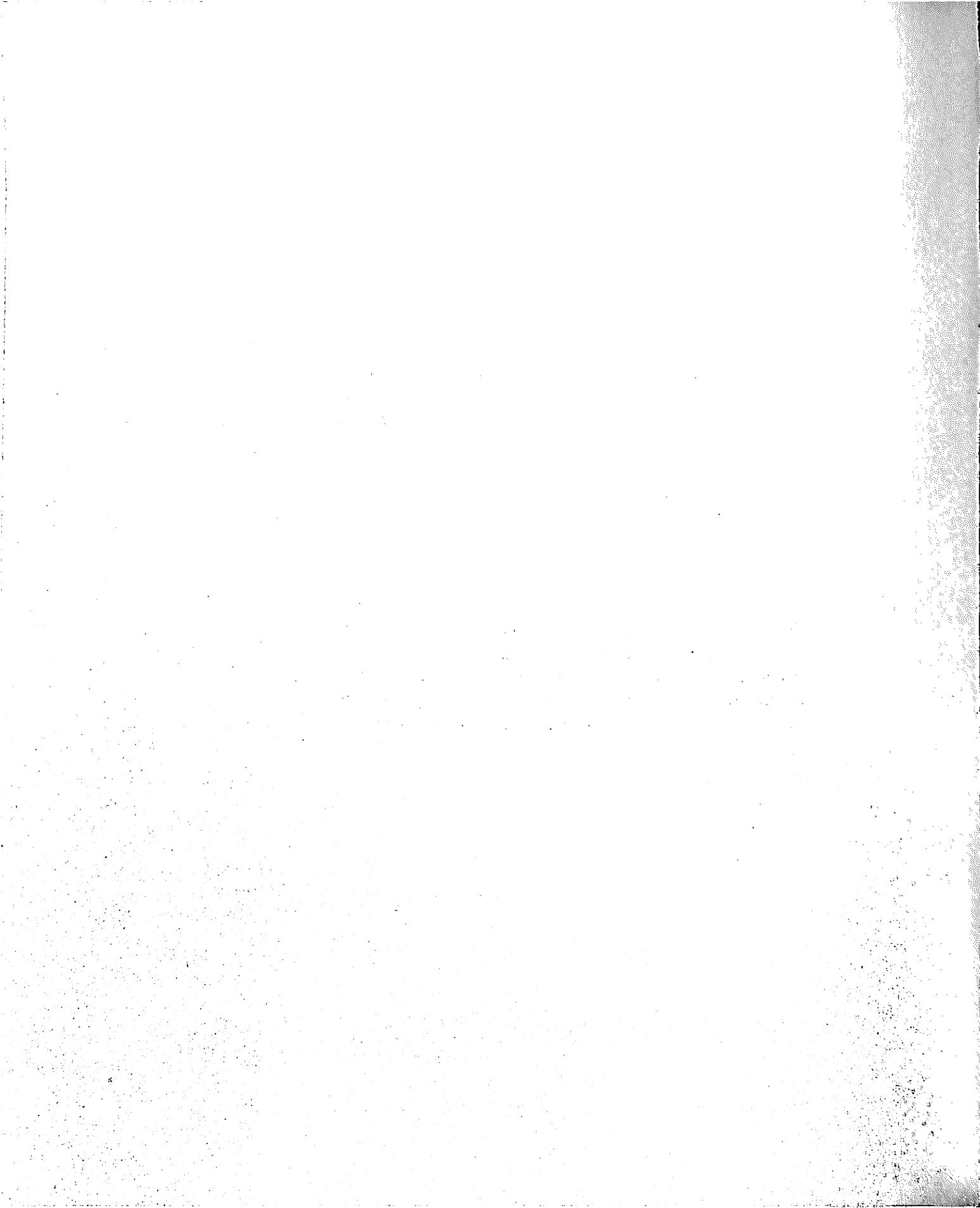
24

Ciertamente, estas lecciones sólo le presentaron una introducción a estos temas. Hay mucho más que usted debe aprender acerca de estos temas y del lenguaje C++. La mejor forma de aprender es practicando. Escriba programas, pruebe herramientas y trabaje con varias bibliotecas de funciones y objetos.

25

¡Buena suerte!

26



# APÉNDICE A

## Precedencia de operadores

Es importante comprender que los operadores tienen una precedencia, pero no es esencial memorizarla.

La *precedencia* es el orden en el que un programa realiza las operaciones de una fórmula matemática. Si un operador tiene precedencia sobre otro operador, se evalúa primero.

Los operadores con mayor precedencia tienen un “lazo más estrecho” que los operadores con menor precedencia; por ende, los operadores con mayor precedencia se evalúan primero. Entre menor sea el rango en la tabla A.1, será menor la precedencia.

**TABLA A.1** Precedencia de operadores

| Rango | Nombre                                                                                                                                                                                                                                                                                                     | Operador                                  |
|-------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------|
| 1     | Resolución de ámbito (binario, unario)                                                                                                                                                                                                                                                                     | ::                                        |
| 2     | Llamadas a funciones, paréntesis, subíndice, selección de miembro, posfixos de incremento y decremento                                                                                                                                                                                                     | () [] . .-> . .++ . .-                    |
| 3     | <code>sizeof</code> , conversión de tipos en C++, prefijos de incremento y decremento, signos más y menos unarios, negación, complemento, conversión explícita en C, <code>sizeof()</code> , dirección, desreferencia, <code>new</code> , <code>new[]</code> , <code>delete</code> , <code>delete[]</code> | . .++ . .- ! - (conversión explícita) & * |
| 4     | Selección de miembro mediante apuntador                                                                                                                                                                                                                                                                    | . * .->*                                  |
| 5     | Multiplicación, división, residuo                                                                                                                                                                                                                                                                          | * / %                                     |
| 6     | Suma, resta                                                                                                                                                                                                                                                                                                | + -                                       |
| 7     | Desplazamiento a nivel de bits                                                                                                                                                                                                                                                                             | << >>                                     |
| 8     | Desigualdad relacional                                                                                                                                                                                                                                                                                     | < <= > >=                                 |
| 9     | Igualdad, desigualdad                                                                                                                                                                                                                                                                                      | == !=                                     |
| 10    | AND a nivel de bits                                                                                                                                                                                                                                                                                        | &                                         |
| 11    | OR exclusivo a nivel de bits (XOR)                                                                                                                                                                                                                                                                         | ^                                         |
| 12    | OR a nivel de bits                                                                                                                                                                                                                                                                                         |                                           |
| 13    | Lógico AND                                                                                                                                                                                                                                                                                                 | &&                                        |
| 14    | Lógico OR                                                                                                                                                                                                                                                                                                  |                                           |
| 15    | Condicional                                                                                                                                                                                                                                                                                                | ? :                                       |
| 16    | Operadores de asignación= *= /= %= += -= <<= >>= &=  = ^=                                                                                                                                                                                                                                                  |                                           |
| 17    | Coma                                                                                                                                                                                                                                                                                                       |                                           |

# APÉNDICE B

## Palabras reservadas de C++

El compilador utiliza las palabras reservadas (también llamadas palabras clave) como parte del lenguaje de C++. No puede definir clases, variables o funciones que tengan como nombre estas palabras reservadas. La lista es un poco arbitraria, ya que contiene las palabras reservadas estándar, así como algunas de las palabras reservadas que son específicas de g++. Algunas de las palabras reservadas no están disponibles en todos los compiladores o en versiones anteriores del compilador g++. Puede haber ligeras variaciones.

|              |                  |             |
|--------------|------------------|-------------|
| asm          | float            | static      |
| auto         | for              | static_cast |
| bool         | friend           | struct      |
| break        | goto             | switch      |
| case         | if               | template    |
| catch        | inline           | this        |
| char         | int              | throw       |
| class        | long             | true        |
| const        | mutable          | try         |
| const_cast   | namespace        | typedef     |
| continue     | new              | typeof      |
| default      | operator         | typeid      |
| delete       | private          | typename    |
| do           | protected        | union       |
| double       | public           | unsigned    |
| dynamic_cast | register         | using       |
| else         | reinterpret_cast | virtual     |
| enum         | return           | void        |
| explicit     | short            | volatile    |
| extern       | signed           | while       |
| false        | sizeof           |             |

# APÉNDICE C

## Números binarios, octales, hexadecimales y una tabla de valores ASCII

Aprendió los fundamentos de la aritmética hace ya tanto tiempo que es difícil imaginar cómo serían las cosas si no tuviera esos conocimientos. Al ver el número 145, inmediatamente ve “ciento cuarenta y cinco” sin necesidad de mucha reflexión.

La comprensión de distintos sistemas de notación numérica requiere que reexamine el número 145 y que lo vea no como un número, sino como un código para (o la *representación de*) un número.

Empiece con cosas pequeñas: examine la relación que existe entre el número tres y “3”. El símbolo 3 es un garabato en un pedazo de papel; el número tres es un concepto. El símbolo se utiliza para representar el concepto de un número.

Esta distinción se puede aclarar si tomamos en cuenta que tres, 3, III y \*\*\* se pueden utilizar para representar el mismo concepto.

En el sistema numérico con base 10 (decimal), se utilizan los números 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9 para representar a todos los números. ¿Cómo se representa el número diez?

Imaginemos que hubiéramos podido desarrollar una estrategia para utilizar la letra A para representar diez; o hubiéramos podido usar IIIIIIIII para representar el concepto. Los romanos utilizaban X. El sistema arábigo, que es el que utilizamos, utiliza la posición junto con los números para representar los valores. La primera columna (la que se encuentra a la derecha) se utiliza para las “unidades”, y la siguiente columna (hacia la izquierda) se utiliza para las “decenas”. Por lo tanto, el número quince se representa como 15 (se lee “uno, cinco”); es decir, una decena (10) y cinco unidades (5).

Surgen ciertas reglas, de las que se pueden hacer algunas generalizaciones:

1. La base 10 utiliza los dígitos del 0 al 9.
2. Las columnas son potencias de diez: unidades, decenas, centenas, y así sucesivamente.
3. Si la tercera columna es 100, el número más grande que puede representar con dos columnas es 99. Dicho en forma más general, con  $n$  columnas puede representar del 0 al  $(10^n - 1)$ . Por lo tanto, con tres columnas puede representar del 0 al  $(10^3 - 1)$  o del 0 al 999.

## Más allá de la base 10

No es una coincidencia que utilicemos la base 10; tenemos 10 dedos. Sin embargo, podríamos imaginarnos una base diferente. Usando las reglas encontradas en la base 10, podríamos describir la base 8:

1. Los dígitos utilizados en la base 8 son del 0 al 7.
2. Las columnas son potencias de 8: unos, ochos, sesenta y cuatro, y así sucesivamente.
3. Con  $n$  columnas, puede representar del 0 al  $8^n - 1$ .

Para distinguir los números escritos en cada base, se escribe la base como subíndice en seguida del número. El número quince en base 10 se escribe como  $15_{10}$  y se lee como “uno, cinco, base diez”.

Por lo tanto, para representar el número  $15_{10}$  en base 8, se escribe  $17_8$ . Esto se lee como “uno, siete, base ocho”. Observe que también se puede leer como “quince octal” ya que ése es el número que sigue representando.

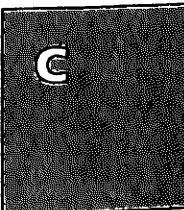
¿Por qué 17? El 1 significa 8 y el 7 significa siete unidades. Un 8 más siete unidades es igual a 15. Considere 15 asteriscos:

\*\*\*\*\*      \*\*\*\*\*  
\*\*\*\*\*

La tendencia natural es formar dos grupos: un grupo de diez asteriscos y otro de cinco. Esto se representaría en decimal como 15 (una decena y cinco unidades). También puede agrupar los asteriscos como

\*\*\*\* \*  
\*\*\*\*

Es decir, un grupo de ocho asteriscos y otro de siete. En base 8, esto se representaría como 17. Es decir, un 8 y siete unidades.



## Un vistazo a las bases

Puede representar el número quince en base 10 como 15, en base 9 como 16, en base 8 como 17, en base 7 como 21. ¿Por qué 21? En base 7 no hay número 8. Para poder representar el quince, necesita dos sietes y un 1.

¿Cómo se generaliza el proceso? Para convertir un número base 10 a base 7, piense en las columnas: en base 7 hay unos, sietes, cuarenta y nueve, trescientos cuarenta y tres, y así sucesivamente. ¿Por qué estas columnas? Representan  $7^0$ ,  $7^1$ ,  $7^2$ ,  $7^3$ , y así sucesivamente.

Puede crear una tabla usted mismo:

|       |       |       |       |
|-------|-------|-------|-------|
| 4     | 3     | 2     | 1     |
| $7^3$ | $7^2$ | $7^1$ | $7^0$ |
| 343   | 49    | 7     | 1     |

La primera fila representa el número de la columna. La segunda fila representa la potencia de 7. La tercera fila representa el valor decimal de cada número de esa fila.

He aquí el procedimiento para convertir un valor decimal a base 7: examine el número y decida cuál columna utilizar primero. Por ejemplo, si el número es 200, sabe que la columna 4 (343) es 0, y no tiene que preocuparse por ella.

Para averiguar cuántos cuarenta y nueve hay, divida 200 entre 49. La respuesta es 4, así que coloque 4 en la columna 3 y examine el residuo: 4. No hay sietes en 4, por lo que debe colocar un cero en la columna de los sietes. Hay cuatro unos en 4, así que coloque un 4 en la columna de los unos. La respuesta es 404.

Para convertir el número 968 a base 6:

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
| 5     | 4     | 3     | 2     | 1     |
| $6^4$ | $6^3$ | $6^2$ | $6^1$ | $6^0$ |
| 1296  | 216   | 36    | 6     | 1     |

No hay 1296s en 968, así que la columna 5 tiene 0. Dividir 968 entre 216 da como resultado 4 con un residuo de 104. La columna 4 tiene 4. Dividir 104 entre 36 da como resultado 2 con un residuo de 32. La columna 3 tiene 2. Dividir 32 entre 6 da como resultado 5 con un residuo de 2. Por lo tanto, la respuesta es 4252.

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
| 5     | 4     | 3     | 2     | 1     |
| $6^4$ | $6^3$ | $6^2$ | $6^1$ | $6^0$ |
| 1296  | 216   | 36    | 6     | 1     |
| 0     | 4     | 2     | 5     | 2     |

Hay un método más sencillo para convertir de una base a otra (por ejemplo, de base 6 a base 10). Puede multiplicar:

$$4 * 216 = 864$$

$$2 * 36 = 72$$

$$5 * 6 = 30$$

$$2 * 1 = 2$$

$$968$$

## Números binarios

La base 2 es la extensión definitiva de este concepto. Sólo hay dos dígitos: 0 y 1. Las columnas son:

|           |       |       |       |       |       |       |       |       |
|-----------|-------|-------|-------|-------|-------|-------|-------|-------|
| Col:      | 8     | 7     | 6     | 5     | 4     | 3     | 2     | 1     |
| Potencia: | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| Valor:    | 128   | 64    | 32    | 16    | 8     | 4     | 2     | 1     |

Para convertir el número 88 a base 2, se sigue el mismo procedimiento: no hay 128s, por lo que la columna 8 es 0.

Hay un 64 en 88, por lo que la columna 7 es 1 y el residuo es 24. No hay 32s en 24, por lo que la columna 6 es 0.

Hay un 16 en 24, por lo que la columna 5 es 1. El residuo es 8. Hay un 8 en 8, así que la columna 4 es 1. No hay residuo, por lo que las columnas restantes son 0.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

Para probar esta respuesta, conviértala otra vez:

$$1 * 64 = 64$$

$$0 * 32 = 0$$

$$1 * 16 = 16$$

$$1 * 8 = 8$$

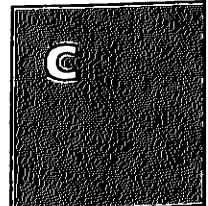
$$0 * 4 = 0$$

$$0 * 2 = 0$$

$$0 * 1 = 0$$

88

Para convertir de binario a decimal, sólo necesita seguir el método utilizado para volver a comprobar nuestra conversión de decimal a binario, es decir, multiplicar el resultado binario por el valor de las columnas respectivas.



## Por qué base 2

El poder de la base 2 es que corresponde a la perfección a lo que una computadora necesita representar. Las computadoras realmente no saben nada acerca de las letras, los números, las instrucciones o los programas. En su interior sólo hay circuitos, y en una unión dada hay mucha energía, o muy poca.

Para mantener clara la lógica, los ingenieros no tratan esto como una escala relativa (poca energía, algo de energía, más energía, mucha energía o toneladas de energía), sino como una escala binaria (“energía suficiente” o “energía insuficiente”). En lugar de decir “suficiente” o “insuficiente”, lo simplifican como “sí” o “no”. Sí o no, o verdadero o falso, se puede representar como 1 o 0. Por convención, 1 significa verdadero o sí, pero esto es sólo una convención; podría, igual de fácil, significar falso o no.

Al dar este gran salto en intuición, el poder del sistema binario se vuelve claro: con unos y ceros se puede representar la verdad fundamental de todo circuito (hay energía o no hay). He aquí todo lo que una computadora sabe: “¿Es o No es?” Es = 1; No es = 0.

## Bits, Bytes, y Nibbles

Cuando se toma la decisión de representar la verdad y la falsedad con 1s y 0s, los dígitos binarios (o bits) se vuelven muy importantes. Debido a que las primeras computadoras podían enviar 8 bits a la vez, era común empezar a escribir código usando números de 8 bits, conocidos como *bytes*.

### Nota

La mitad de un byte (4 bits) se conoce como nibble.

Con 8 dígitos binarios se pueden representar hasta 256 valores distintos. ¿Por qué? Examine las columnas: si los 8 bits están encendidos (1), el valor es 255. Si ninguno está encendido (todos los bits están apagados o valen cero), el valor es 0. El rango 0–255 le proporciona 256 posibles estados.

## Qué es un KB

Resulta que  $2^{10}$  (1,024) es aproximadamente igual a  $10^3$  (1,000). Esta coincidencia era demasiado buena como para pasarla por alto, por lo que los científicos de la computación empezaron a referirse a  $2^{10}$  bytes como 1 KB o 1 kilobyte, basándose en el prefijo científico de kilo para representar mil.

De la misma manera,  $1024 * 1024$  (1,048,576) se encuentra lo suficientemente cerca de un millón para recibir la designación de 1 MB o de 1 megabyte, y 1,024 megabytes se conocen como 1 gigabyte (giga implica mil millones).

## Números binarios

Las computadoras utilizan combinaciones de 1s y 0s para codificar todo lo que hacen. Las instrucciones en lenguaje de máquina se codifican como una serie de 1s y 0s y son interpretadas por los circuitos fundamentales. Los científicos de la computación pueden convertir otra vez en números estos conjuntos arbitrarios de 1s y 0s, pero sería un error pensar que estos números tienen un significado intrínseco.

Por ejemplo, el juego de chips Intel 80 × 86 interpreta la combinación de bits 1001 0101 como una instrucción. Evidentemente, este valor se puede convertir a decimal (149), pero ese número no tiene significado por sí mismo.

Algunas veces los números son instrucciones, otras veces son valores, y algunas otras son códigos. ASCII (Código Estándar Estadounidense para el Intercambio de Información) es un importante conjunto de código estandarizado. En ASCII, cada letra y carácter de puntuación tiene una representación de 7 dígitos binarios. Por ejemplo, la letra “a” minúscula se representa con 0110 0001. Éste no es un número, aunque puede convertirlo en el número 97 ( $64 + 32 + 1$ ). Por esta razón, se dice que en ASCII la letra “a” está representada por el 97; pero lo cierto es que la representación binaria de 97, 0110001, es la codificación de la letra “a”, y el valor decimal 97 es una conveniencia humana.

## Números octales

Debido a que los números binarios son difíciles de leer, buscamos una manera más sencilla de representar los mismos valores. Traducir de binario a base 10 implica una manipulación de números bastante compleja; es más sencillo traducir de octal a base 10.

Para comprender esto, primero debe comprender la base 8, conocida como *octal*. En base 8 hay ocho números: 0, 1, 2, 3, 4, 5, 6 y 7. Las columnas en octal son:

|           |       |       |       |       |       |
|-----------|-------|-------|-------|-------|-------|
| Col:      | 5     | 4     | 3     | 2     | 1     |
| Potencia: | $8^4$ | $8^3$ | $8^2$ | $8^1$ | $8^0$ |
| Valor:    | 4096  | 512   | 64    | 8     | 1     |

Para convertir el número 88 a octal, se siguen los mismos procedimientos que con los números binarios: no hay 4096s ni 512s, por lo que las columnas 5 y 4 son 0, respectivamente.

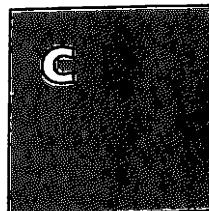
Hay un 64 en el 88, por lo que la columna 3 es 1 con 24 como residuo. Hay tres ochoes en 24, por lo que la columna 2 es 3 con un residuo de cero (por lo que la columna restante es 0).

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 0 | 1 | 3 | 0 |
|---|---|---|---|---|

Para probar esta respuesta, la convertimos en forma inversa:

$$\begin{array}{r}
 1 * 64 = 64 \\
 3 * 8 = 24 \\
 0 * 1 = 0
 \end{array}$$

88



Para convertir de octal a decimal sólo hay que seguir el método utilizado para comprobar nuestra conversión de decimal a octal, es decir, multiplicar el resultado en octal por el valor de las columnas respectivas.

## Por qué octal

El poder de los números octales (base 8) reside en que están en un nivel más alto que los binarios. Cada dígito octal representa 3 dígitos binarios. Históricamente, los sistemas computacionales originales en los que se desarrollaron UNIX y C eran equipos de 12 bits. Se requerían 4 dígitos octales para describir una palabra de memoria.

Cada dígito octal equivale a 3 dígitos binarios. Es muy sencillo alinear los dígitos octales y convertirlos, uno a la vez, en binarios. Y la conversión a la inversa también es fácil.

## Números hexadecimales

Como los números binarios son difíciles de leer, necesitamos una manera más sencilla de representar los mismos valores. La traducción de binario a base 10 requiere una manipulación bastante compleja de números; pero resulta que traducir de base 2 a base 16 es muy sencillo, pues existe un muy buen atajo.

Para comprender esto, primero debe comprender lo que es la base 16, conocida como hexadecimal. En base 16 hay dieciséis números: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F. Los últimos seis son arbitrarios; se eligieron las letras de la A a la F ya que son fáciles de representar en un teclado. Las columnas en hexadecimal son:

|        |        |        |        |
|--------|--------|--------|--------|
| 4      | 3      | 2      | 1      |
| $16^3$ | $16^2$ | $16^1$ | $16^0$ |
| 4096   | 256    | 16     | 1      |

Para convertir de hexadecimal a decimal se puede multiplicar. Por ejemplo, el número F8C representa:

$$\begin{array}{r}
 F * 256 = 15 * 256 = 3840 \\
 8 * 16 = 128 \\
 C * 1 = 12 * 1 = 12 \\
 \hline
 3980
 \end{array}$$

Traducir el número FC a binario se puede hacer traduciendo primero a base 10 y luego a binario:

$$\begin{array}{r}
 F * 16 = 15 * 16 = 240 \\
 C * 1 = 12 * 1 = 12 \\
 \hline
 252
 \end{array}$$

Convertir el número 252<sub>10</sub> a binario requiere la tabla:

|           |       |       |       |       |       |       |       |       |       |
|-----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| Col:      | 9     | 8     | 7     | 6     | 5     | 4     | 3     | 2     | 1     |
| Potencia: | $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| Valor:    | 256   | 128   | 64    | 32    | 16    | 8     | 4     | 2     | 1     |

No hay 256s.

1 128 quedan 124

1 64 quedan 60

1 32 quedan 28

1 16 quedan 12

1 8 quedan 4

1 4 quedan 0

0

0

1 1 1 1 1 1 0 0

Por lo tanto, la respuesta en binario es 1111 1100.

Ahora, resulta que si se trata este número binario como dos conjuntos de 4 dígitos, se puede hacer una transformación mágica.

El conjunto de la derecha es 1100. En decimal esto es 12, o en hexadecimal es C.

El conjunto de la izquierda es 1111, que en base 10 es 15, o en hexadecimal es F.

Por lo tanto, tenemos:

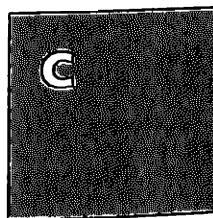
|      |      |
|------|------|
| 1111 | 1100 |
| F    | C    |

Si se colocan los dos números hexadecimales se forma FC, que es el valor real de 1111 1100. Este atajo siempre funciona. Puede tomar cualquier número binario de cualquier longitud, y reducirlo en conjuntos de 4, convertir cada conjunto de 4 a hexadecimal y colocar los números hexadecimales juntos para obtener el resultado en hexadecimal. He aquí un número más grande:

1011 0001 1101 0111

Las columnas son 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 y 32768.

|             |        |
|-------------|--------|
| 1 x 1 =     | 1      |
| 1 x 2 =     | 2      |
| 1 x 4 =     | 4      |
| 0 x 8 =     | 0      |
| <br>        |        |
| 1 x 16 =    | 16     |
| 0 x 32 =    | 0      |
| 1 x 64 =    | 64     |
| 1 x 128 =   | 128    |
| <br>        |        |
| 1 x 256 =   | 256    |
| 0 x 512 =   | 0      |
| 0 x 1024 =  | 0      |
| 0 x 2048 =  | 0      |
| <br>        |        |
| 1 x 4096 =  | 4,096  |
| 1 x 8192 =  | 8,192  |
| 0 x 16384 = | 0      |
| 1 x 32768 = | 32,768 |
| Total:      | 45,527 |



Para convertir esto a hexadecimal se requiere una tabla con los valores hexadecimales.

65536      4096      256      16      1

No hay 65,536s en 45,527, por lo que la primera columna es 4096. Hay once 4096s (45,056), con un residuo de 471. Hay un 256 en 471 con un residuo de 215. Hay trece 16s (208) en 215 con un residuo de 7. Por lo tanto, el número hexadecimal es B1D7.

Comprobemos las matemáticas:

|                 |        |
|-----------------|--------|
| B (11) * 4096 = | 45,056 |
| 1 * 256 =       | 256    |
| D (13) * 16 =   | 208    |
| 7 * 1 =         | 7      |
| Total           | 45,527 |

La versión con atajo sería tomar el número binario original, 1011000111010111, y dividirlo en grupos de 4: 1011 0001 1101 0111. Cada uno de los cuatro se evalúa como número hexadecimal:

1011 =  
1 x 1 = 1  
1 x 2 = 2  
0 x 4 = 0  
1 x 8 = 8  
Total 11  
Hex: B

0001 =  
1 x 1 = 1  
0 x 2 = 0  
0 x 4 = 0  
0 \* 8 = 0  
Total 1  
Hex: 1

1101 =  
1 x 1 = 1  
0 x 2 = 0  
1 x 4 = 4  
1 x 8 = 8  
Total 13  
Hex = D

0111 =  
1 x 1 = 1  
1 x 2 = 2  
1 x 4 = 4  
0 x 8 = 0  
Total 7  
Hex: 7

Total Hex: B1D7

El sistema hexadecimal es muy popular en los sistemas con múltiplos pares de 4 bits. En la actualidad, la mayoría de los sistemas entra en esta categoría: 16, 32 y 64. Para cuando este libro se imprima, probablemente habrá sistemas de 128 bits en el mercado.

## ASCII

ASCII es la representación de caracteres más utilizada actualmente. La tabla C.1 muestra valores binarios, octales, decimales hexadecimales y sus equivalentes en ASCII.

**TABLA C.1** Sistemas numéricicos y sus equivalentes en ASCII

C

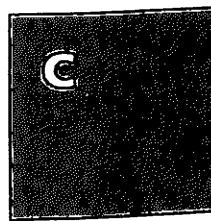
| <i>Binario</i> | <i>Octal</i> | <i>Hex</i> | <i>Decimal</i> | <i>ASCII</i> | <i>Octal</i> | <i>Hex</i> | <i>Decimal</i> |
|----------------|--------------|------------|----------------|--------------|--------------|------------|----------------|
| 00000000       | 0            | 0          | 0              | <nul>        | 00           | 00000000   | 0              |
| 00000001       | 1            | 1          | 1              | <soh>        | 01           | 00000001   | 1              |
| 00000010       | 2            | 2          | 2              | <stx>        | 02           | 00000002   | 2              |
| 00000011       | 3            | 3          | 3              | <etx>        | 03           | 00000003   | 3              |
| 00000100       | 4            | 4          | 4              | <eot>        | 04           | 00000004   | 4              |
| 00000101       | 5            | 5          | 5              | <enq>        | 05           | 00000005   | 5              |
| 00000110       | 6            | 6          | 6              | <ack>        | 06           | 00000006   | 6              |
| 00000111       | 7            | 7          | 7              | <bel>        | 07           | 00000007   | 7              |
| 00001000       | 10           | 8          | 8              | <bs>         | 10           | 00000008   | 8              |
| 00001001       | 11           | 9          | 9              | <tab>        | 11           | 00000009   | 9              |
| 00001010       | 12           | a          | 10             | <nuevalínea> | 12           | 0000000a   | 10             |
| 00001011       | 13           | b          | 11             | <vt>         | 13           | 0000000b   | 11             |
| 00001100       | 14           | c          | 12             | <ff>         | 14           | 0000000c   | 12             |
| 00001101       | 15           | d          | 13             | <cr>         | 15           | 0000000d   | 13             |
| 00001110       | 16           | e          | 14             | <so>         | 16           | 0000000e   | 14             |
| 00001111       | 17           | f          | 15             | <si>         | 17           | 0000000f   | 15             |
| 00010000       | 20           | 10         | 16             | <dle>        | 20           | 00000010   | 16             |
| 00010001       | 21           | 11         | 17             | <dc1>        | 21           | 00000011   | 17             |
| 00010010       | 22           | 12         | 18             | <dc2>        | 22           | 00000012   | 18             |
| 00010011       | 23           | 13         | 19             | <dc3>        | 23           | 00000013   | 19             |
| 00010100       | 24           | 14         | 20             | <dc4>        | 24           | 00000014   | 20             |
| 00010101       | 25           | 15         | 21             | <nak>        | 25           | 00000015   | 21             |
| 00010110       | 26           | 16         | 22             | <syn>        | 26           | 00000016   | 22             |
| 00010111       | 27           | 17         | 23             | <etb>        | 27           | 00000017   | 23             |
| 00011000       | 30           | 18         | 24             | <can>        | 30           | 00000018   | 24             |
| 00011001       | 31           | 19         | 25             | <em>         | 31           | 00000019   | 25             |
| 00011010       | 32           | 1a         | 26             | <sub>        | 32           | 0000001a   | 26             |
| 00011011       | 33           | 1b         | 27             | <esc>        | 33           | 0000001b   | 27             |
| 00011100       | 34           | 1c         | 28             | <fs>         | 34           | 0000001c   | 28             |
| 00011101       | 35           | 1d         | 29             | <gs>         | 35           | 0000001d   | 29             |
| 00011110       | 36           | 1e         | 30             | <rs>         | 36           | 0000001e   | 30             |

continúa

**TABLA C.1** CONTINUACIÓN

| <i>Binario</i> | <i>Octal</i> | <i>Hex</i> | <i>Decimal</i> | <i>ASCII</i> |
|----------------|--------------|------------|----------------|--------------|
| 00011111       | 37           | 1f         | 31             | <us>         |
| 00100000       | 40           | 20         | 32             | <espacio>    |
| 00100001       | 41           | 21         | 33             | !            |
| 00100010       | 42           | 22         | 34             | "            |
| 00100011       | 43           | 23         | 35             | #            |
| 00100100       | 44           | 24         | 36             | \$           |
| 00100101       | 45           | 25         | 37             | %            |
| 00100110       | 46           | 26         | 38             | &            |
| 00100111       | 47           | 27         | 39             | '            |
| 00101000       | 50           | 28         | 40             | (            |
| 00101001       | 51           | 29         | 41             | )            |
| 00101010       | 52           | 2a         | 42             | *            |
| 00101011       | 53           | 2b         | 43             | +            |
| 00101100       | 54           | 2c         | 44             | .            |
| 00101101       | 55           | 2d         | 45             | -            |
| 00101110       | 56           | 2e         | 46             | ,            |
| 00101111       | 57           | 2f         | 47             | /            |
| 00110000       | 60           | 30         | 48             | 0            |
| 00110001       | 61           | 31         | 49             | 1            |
| 00110010       | 62           | 32         | 50             | 2            |
| 00110011       | 63           | 33         | 51             | 3            |
| 00110100       | 64           | 34         | 52             | 4            |
| 00110101       | 65           | 35         | 53             | 5            |
| 00110110       | 66           | 36         | 54             | 6            |
| 00110111       | 67           | 37         | 55             | 7            |
| 00111000       | 70           | 38         | 56             | 8            |
| 00111001       | 71           | 39         | 57             | 9            |
| 00111010       | 72           | 3a         | 58             | :            |
| 00111011       | 73           | 3b         | 59             | ;            |
| 00111100       | 74           | 3c         | 60             | <            |
| 00111101       | 75           | 3d         | 61             | =            |
| 00111110       | 76           | 3e         | 62             | >            |

| <i>Binario</i> | <i>Octal</i> | <i>Hex</i> | <i>Decimal</i> | <i>ASCII</i> |
|----------------|--------------|------------|----------------|--------------|
| 00111111       | 77           | 3f         | 63             | ?            |
| 01000000       | 100          | 40         | 64             | @            |
| 01000001       | 101          | 41         | 65             | A            |
| 01000010       | 102          | 42         | 66             | B            |
| 01000011       | 103          | 43         | 67             | C            |
| 01000100       | 104          | 44         | 68             | D            |
| 01000101       | 105          | 45         | 69             | E            |
| 01000110       | 106          | 46         | 70             | F            |
| 01000111       | 107          | 47         | 71             | G            |
| 01001000       | 110          | 48         | 72             | H            |
| 01001001       | 111          | 49         | 73             | I            |
| 01001010       | 112          | 4a         | 74             | J            |
| 01001011       | 113          | 4b         | 75             | K            |
| 01001100       | 114          | 4c         | 76             | L            |
| 01001101       | 115          | 4d         | 77             | M            |
| 01001110       | 116          | 4e         | 78             | N            |
| 01001111       | 117          | 4f         | 79             | O            |
| 01010000       | 120          | 50         | 80             | P            |
| 01010001       | 121          | 51         | 81             | Q            |
| 01010010       | 122          | 52         | 82             | R            |
| 01010011       | 123          | 53         | 83             | S            |
| 01010100       | 124          | 54         | 84             | T            |
| 01010101       | 125          | 55         | 85             | U            |
| 01010110       | 126          | 56         | 86             | V            |
| 01010111       | 127          | 57         | 87             | W            |
| 01011000       | 130          | 58         | 88             | X            |
| 01011001       | 131          | 59         | 89             | Y            |
| 01011010       | 132          | 5a         | 90             | Z            |
| 01011011       | 133          | 5b         | 91             | [            |
| 01011100       | 134          | 5c         | 92             | \            |
| 01011101       | 135          | 5d         | 93             | ]            |
| 01011110       | 136          | 5e         | 94             | ^            |



**TABLA C.1** CONTINUACIÓN

| <i>Binario</i> | <i>Octal</i> | <i>Hex</i> | <i>Decimal</i> | <i>ASCII</i> |
|----------------|--------------|------------|----------------|--------------|
| 01011111       | 137          | 5f         | 95             |              |
| 01100000       | 140          | 60         | 96             |              |
| 01100001       | 141          | 61         | 97             | a            |
| 01100010       | 142          | 62         | 98             | b            |
| 01100011       | 143          | 63         | 99             | c            |
| 01100100       | 144          | 64         | 100            | d            |
| 01100101       | 145          | 65         | 101            | e            |
| 01100110       | 146          | 66         | 102            | f            |
| 01100111       | 147          | 67         | 103            | g            |
| 01101000       | 150          | 68         | 104            | h            |
| 01101001       | 151          | 69         | 105            | i            |
| 01101010       | 152          | 6a         | 106            | j            |
| 01101011       | 153          | 6b         | 107            | k            |
| 01101100       | 154          | 6c         | 108            | l            |
| 01101101       | 155          | 6d         | 109            | m            |
| 01101110       | 156          | 6e         | 110            | n            |
| 01101111       | 157          | 6f         | 111            | o            |
| 01110000       | 160          | 70         | 112            | p            |
| 01110001       | 161          | 71         | 113            | q            |
| 01110010       | 162          | 72         | 114            | r            |
| 01110011       | 163          | 73         | 115            | s            |
| 01110100       | 164          | 74         | 116            | t            |
| 01110101       | 165          | 75         | 117            | u            |
| 01110110       | 166          | 76         | 118            | v            |
| 01110111       | 167          | 77         | 119            | w            |
| 01111000       | 170          | 78         | 120            | x            |
| 01111001       | 171          | 79         | 121            | y            |
| 01111010       | 172          | 7a         | 122            | z            |
| 01111011       | 173          | 7b         | 123            | {            |
| 01111100       | 174          | 7c         | 124            |              |
| 01111101       | 175          | 7d         | 125            | }            |
| 01111110       | 176          | 7e         | 126            | ~            |
| 01111111       | 177          | 7f         | 127            | <del>        |

Los valores del 0 al 127 (decimal) son parte del estándar ASCII. Algunos sistemas soportan lo que se conoce como "ASCII extendido" (128 a 255) para caracteres imprimibles especiales. Todas las entradas ASCII de la tabla C.1 que se muestran encerradas entre el signo mayor que (<) y el signo menor que (>) (como <espacio>) son caracteres especiales o son difíciles de representar al imprimirlas.

Existen otros conjuntos de caracteres además de ASCII: EBCDIC y Unicode son los más comunes. EBCDIC (Código Extendido de Caracteres Decimales Codificados en Binario para el Intercambio de Información) es el conjunto de caracteres utilizado en mainframes de IBM. Unicode es un nuevo estándar que permite la representación de caracteres en la mayoría de los lenguajes del mundo; ASCII es un subconjunto de Unicode.

El listado C.1 muestra el programa que produjo la tabla C.1 (exceptuando los caracteres especiales encerrados entre los signos < y >, como <espacio>; tuve que editarlos manualmente).

### ENTRADA LISTADO C.1 Programa creador de la tabla C.1

```

1: // listado C.1 (lstxc-01.cxx) - Creación de la tabla C.1.
2: #include <iostream.h>
3: #include <iomanip.h>
4: #include <math.h>
5:
6: void bin( int in, char resultados[] );
7:
8: int main()
9: {
10:     int ciclo;
11:     char resultadobin[ 9 ];
12:
13:     for ( ciclo = 0; ciclo < 128; ciclo++ )
14:     {
15:         bin(ciclo, resultadobin);
16:         cout << resultadobin << "\t" << oct << ciclo << "\t" <<
17:             hex << ciclo << "\t" << dec << ciclo << "\t" << (char) ciclo;
18:         cout << endl;
19:     }
20:
21:     return 0;
22: }
23:
24: void bin( int in, char resultados[] )
25: {
26:     int ciclo;
27:     for ( ciclo = 0; ciclo < 8; ciclo++ )
28:     {
29:         resultados[ ciclo ] = in / ( int ) pow( 2, 7 - ciclo );
30:         in %= ( int ) pow( 2, 7 - ciclo );
31:         resultados[ ciclo ] += '0';

```

continúa

**LISTADO C.1** CONTINUACIÓN

---

```
32:     }
33:     resultados[ 8 ] = '\0';
34:
35:     return;
36: }
```

---

**SALIDA**

La salida de este programa se muestra en la tabla C.1.

**ANÁLISIS**

C++ me permite especificar el formato para la salida (octal, hexadecimal y decimal). Pero no me permitirá especificar binario como un formato de salida. Así que tuve que utilizar una función de conversión, cuyo prototipo se muestra en la línea 6, y la función aparece en las líneas 24 a 36.

El programa principal es muy sencillo: un ciclo `for`. Dentro de ese ciclo hay una llamada a `bin()` y mi instrucción de salida. En la línea 17, para que `cout` muestre la copia final de la variable `ciclo` como carácter, tuve que convertirla a carácter y no a entero.

La función `bin()` implementa la conversión de decimal a binario descrita en la sección “Números binarios” de este apéndice. Las líneas 29 y 30 utilizan la función `pow()` para determinar el valor decimal de una potencia específica de 2. La función `pow()` se define en `math.h` (línea 4) y regresa un tipo `double`; como se requiere la división y el módulo de enteros, el resultado de `pow()` se convierte a tipo `int`.

El único truco verdadero de esta función está en la línea 31, en donde se convierten unos y ceros enteros en caracteres imprimibles (sumando el valor de un cero ASCII). Un uno ASCII es un número mayor (61 en comparación con 60) que el cero ASCII, por lo que esto funciona correctamente.

La línea 33 agrega un terminador nulo para que el arreglo `resultados` se pueda tratar como una cadena.

En este caso opté por tener un número de 8 posiciones (8 bits) con la posición de valor más alto en primer lugar.

Éste es un buen ejemplo de algunos de los trucos que C y C++ le permiten hacer. El ejemplo del listado C.1 es difícil de leer, pero la conversión de números en sí puede ser compleja.

# APÉNDICE D

## Respuestas a los cuestionarios y ejercicios

### Día 1

#### Cuestionario

1. ¿Cuál es la diferencia entre un intérprete y un compilador?

Los intérpretes leen el código fuente y traducen un programa, convirtiendo el código del programador, o instrucciones del programa, directamente en acciones. Los compiladores convierten el código fuente en un programa ejecutable que puede ejecutarse posteriormente.

2. ¿Cómo compila el código fuente con su compilador?

Cada compilador es distinto. Asegúrese de revisar la documentación incluida con su compilador. Con GNU se pueden utilizar los comandos `man` (manual) o `info` para ver la documentación.

### 3. ¿Para qué sirve el enlazador?

El trabajo del enlazador es unir su código compilado con las bibliotecas proporcionadas con su compilador GNU y de otras fuentes. El enlazador le permite crear su programa en piezas y luego enlazar las piezas entre sí para formar un gran programa.

### 4. ¿Cuáles son los pasos del ciclo normal de desarrollo?

Editar el código fuente, compilar, enlazar, probar y repetir.

## Ejercicios

1. Inicializa dos variables de tipo entero y luego imprime su suma y su producto.
2. Vea la documentación del compilador de GNU (o el día 2, "Los componentes de un programa de C++").
3. Debe colocar el símbolo # antes de la palabra `include` en la primera línea.
4. Este programa imprime las palabras "¡Hola, mundo!" en la pantalla, seguidas de un carácter de nueva línea (retorno de carro).

## Día 2

### Cuestionario

#### 1. ¿Cuál es la diferencia entre el compilador y el preprocesador?

Cada vez que ejecuta su compilador, el preprocesador se ejecuta primero. Lee su código fuente e incluye los archivos que usted pide, y realiza otras tareas de mantenimiento. El preprocesador se describe detalladamente en el día 18, "Análisis y diseño orientados a objetos".

#### 2. ¿Por qué es especial la función `main()`?

`main()` se llama automáticamente cada vez que se ejecuta su programa.

#### 3. ¿Cuáles son los dos tipos de comentarios, y en qué se diferencian?

Los comentarios estilo C++ empiezan con dos barras diagonales (//), y convierten en comentario cualquier texto hasta llegar al final de la línea. Los comentarios estilo C vienen en pares /\* \* /, y todo lo que se encuentre entre estos símbolos se convierte en comentario. Debe asegurarse de que haya pares relacionados.

#### 4. ¿Se pueden anidar los comentarios?

Sí, los comentarios estilo C++ pueden anidarse dentro de comentarios estilo C. De hecho, se pueden anidar comentarios estilo C dentro de comentarios estilo C++, siempre y cuando recuerde que éstos terminan al final de la línea. La excepción es que no se pueden anidar comentarios estilo C. Si usted crea un comentario estilo /\* \_\_\_\_\_ \*/ \* \_\_\_\_\_ \* / \_\_\_\_\_ \* /, el primer cierre de comentarios (\*) cierra todo el comentario y obtendrá un error por los siguientes cierres de comentario. Esto se debe a que el compilador no crea una pila para los delimitadores de comentarios.

5. ¿Pueden los comentarios ser de más de una línea?

Los comentarios estilo C sí. Si quiere extender los comentarios estilo C++ hasta otra línea, debe colocar otro par de barras diagonales (//).

## Ejercicios

1. Escriba un programa que imprima en la pantalla el mensaje “Me gusta C++”.

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:     cout << "Me gusta C++";
6:     return 0;
7: }
```

2. Escriba el programa más pequeño que se pueda compilar, enlazar y ejecutar.

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

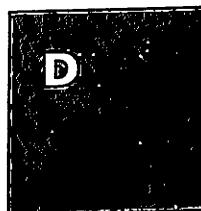
3. CAZA ERRORES: Escriba este programa y compílelo. ¿Por qué falla? ¿Cómo puede arreglarlo?

```
1: #include <iostream.h>
2: int main()
3: {
4:     cout << ¿Hay un error aquí?";
5:     return 0;
6: }
```

En la línea 4 falta un par de comillas al principio de la cadena.

4. Encuentre el error del ejercicio 3 y vuelva a compilar, enlazar y ejecutar el programa.

```
1: #include <iostream.h>
2: main()
3: {
4:     cout << "¿Hay un error aquí?";
5: }
```



## Día 3

### Cuestionario

1. ¿Cuál es la diferencia entre una variable de tipo entero y una de punto flotante?

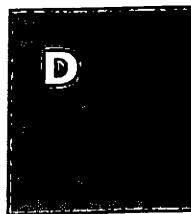
Las variables de tipo entero son números enteros; las variables de punto flotante son decimales y tienen un punto decimal “flotante”. Los números de punto flotante se representan por medio de una mantisa y un exponente. El exponente indicará el lugar en el que se debe colocar el punto, de aquí el nombre de *punto flotante*.

2. ¿Cuáles son las diferencias entre un entero corto sin signo y un entero largo?  
La palabra reservada `unsigned` significa que el valor entero sólo guardará números positivos. En la mayoría de las computadoras, los enteros cortos son de 2 bytes y los enteros largos son de 4. Pero en general, una variable de tipo `long` tiene el doble de capacidad que una variable de tipo `short`.
3. ¿Cuáles son las ventajas de usar una constante simbólica en lugar de una constante literal?  
Una constante simbólica se explica a sí misma; el nombre de la constante indica su función. Además, las constantes simbólicas pueden redefinirse en una sola ubicación en el código fuente, en lugar de que el programador tenga que editar el código en cualquier parte en que se utilice la literal.
4. ¿Cuáles son las ventajas de usar la palabra reservada `const` en lugar de `#define`?  
Las variables `const` están “tipificadas”; por lo tanto, el compilador puede comprobar que no haya errores en la forma en que se utilizan. Además, sobreviven al preprocesador; por lo tanto, el nombre está disponible en el depurador.
5. ¿Qué hace que el nombre de una variable sea bueno o malo?  
Un buen nombre de variable indica para lo que sirve la variable; un nombre malo no da información. `miEdad` y `PersonasEnElAutobus` son buenos nombres de variables, pero `xjk` y `prndl` probablemente son menos útiles.
6. Dado el siguiente enum, ¿cuál es el valor de Azul?  
`enum COLOR { BLANCO, NEGRO = 100, ROJO, AZUL, VERDE = 300 };`  
`AZUL = 102`
7. ¿Cuáles de los siguientes nombres de variables son buenos, cuáles son malos y cuáles no son válidos?
  - a. `Edad`  
`Bueno`
  - b. `!ex`  
`No es válido`
  - c. `R79J`  
`Válido, pero es una mala elección`
  - d. `IngresosTotal`  
`Bueno`
  - e. `__Invalido`  
`Válido, pero es una mala elección`

## Ejercicios

1. Cuál sería el tipo de variable correcto para guardar la siguiente información?

- a. Su edad.  
Entero de tipo `unsigned short`
  - b. El área de su patio.  
Entero de tipo `unsigned long` o flotante de tipo `unsigned float`
  - c. El número de estrellas en la galaxia.  
`unsigned double`
  - d. La cantidad promedio de lluvia para el mes de enero.  
Entero de tipo `unsigned short`
2. Cree nombres buenos de variables para la información de la pregunta 1.
- a. `miEdad`
  - b. `areaPatio`
  - c. `EstrellasEnGalaxia`
  - d. `lluviaPromedio`
3. Declare una constante para *pi* como 3.14159.
- ```
const float PI = 3.14159;
```
4. Declare una variable de tipo float e inicialícela usando su constante *pi*.
- ```
float miPi = PI;
```



## Día 4

### Cuestionario

1. ¿Qué es una expresión?  
Cualquier instrucción que regrese un valor.
2. ¿Es  $x = 5 + 7$  una expresión? ¿Cuál es su valor?  
Sí. 12
3. ¿Cuál es el valor de  $201 / 4$ ?  
50
4. ¿Cuál es el valor de  $201 \% 4$ ?  
1
5. Si `miEdad`, `a` y `b` son variables de tipo `int`, ¿cuáles son sus valores después de ejecutar las siguientes instrucciones?  
`miEdad = 39;`  
`a = miEdad++;`  
`b = ++miEdad;`  
`miEdad: 41, a: 39, b: 41`

6. ¿Cuál es el valor de  $8+2*3$ ?

14

7. ¿Cuál es la diferencia entre  $x = 3$  y  $x == 3$ ?

La primera instrucción asigna 3 a x y regresa true. La segunda prueba si x es igual a 3; regresa true si el valor de x es igual a 3 y false si no lo es.

8. ¿Qué son los siguientes valores, verdaderos o falsos?

a. 0

false

b. 1

verdadero

c. -1

verdadero

d.  $x = 0$

false

e.  $x == 0$  // asuma que x vale 0

verdadero

## Ejercicios

1. Escriba una instrucción if sencilla que examine dos variables de tipo entero y que cambie la más grande a la más pequeña, usando sólo una cláusula else.

```
if (x > y)
    x = y;
else          // y > x || y == x
    y = x;
```

2. Examine el siguiente programa. Imagine que escribe tres números y escriba la salida que espera obtener.

```
1: #include <iostream.h>
2: int main()
3: {
4:     int a, b, c;
5:     cout << "Escriba tres números\n";
6:     cout << "a:      ";
7:     cin >> a;
8:     cout << "\nb:      ";
9:     cin >> b;
10:    cout << "\nc:      ";
11:    cin >> c;
12:
13:    if (c = (a-b))
14:    {
15:        cout << "a:      ";
16:        cout << a;
```

```
17:     cout << "menos b:    ";
18:     cout << b;
19:     cout << "igual a c:    ";
20:     cout << c << endl;
21: }
22: else
23:     cout << "a-b no es igual a c:    " << endl;
24: return 0;
25: }
```

3. Escriba el programa del ejercicio 2; compílelo, enlácelo y ejecútelo. Escriba los números 20, 10, y 50. ¿Obtuvo la salida que esperaba? ¿Por qué no?

Se escribe 20, 10, 50.

Se obtiene a: 20 b: 10 c: 10.

La línea 13 está asignando, no probando igualdad.

4. Examine este programa y trate de adivinar la salida:

```
1: #include <iostream.h>
2: int main()
3: {
4:     int a = 1, b = 1, c;
5:     if (c = (a-b))
6:         cout << "El valor de c es: " << c;
7:     return 0;
8: }
```

5. Escriba, compile, enlace y ejecute el programa del ejercicio 4. ¿Cuál fue la salida? ¿Por qué?

Como en la línea 5 se asigna el valor de a-b a c, el valor de la asignación es a (1) menos b (1), o 0. Ya que 0 se evalúa como false, la instrucción if falla y no se imprime nada.

## Día 5

### Cuestionario

1. ¿Cuáles son las diferencias entre el prototipo de la función y la definición de la función?

El prototipo de la función declara a la función; la definición contiene la implementación. El prototipo termina con punto y coma; la definición no lo necesita. La declaración puede incluir la palabra reservada `inline` y valores predeterminados para los parámetros; la definición no. La declaración no necesita incluir nombres para los parámetros; la definición debe incluirlos.

2. ¿Tienen que concordar los nombres de los parámetros en el prototipo, la definición y la llamada a la función?

No. Todos los parámetros se identifican por su posición, no por su nombre.

3. ¿Cómo se declara una función si no regresa un valor?  
Declare la función de forma que regrese `void`.
4. Si no declara un valor de retorno, ¿qué tipo de valor de retorno se asume?  
Cualquier función que no declare explícitamente un tipo de valor de retorno regresa `int`.
5. ¿Qué es una variable local?  
Una variable local es una variable que se pasa a o se declara dentro de un bloque, que por lo general es una función. Está visible sólo dentro del bloque.
6. ¿Qué es el alcance?  
El alcance se refiere a la visibilidad y el tiempo de vida de las variables locales y globales. El alcance se establece generalmente con un conjunto de llaves.
7. ¿Qué es la recursión?  
La recursión se refiere generalmente a la habilidad de una función para llamarse a sí misma.
8. ¿Cuándo debe utilizar variables globales?  
Las variables globales se utilizan generalmente cuando muchas funciones necesitan tener acceso a los mismos datos. Éstas variables son muy raras en C++; cuando sepa cómo utilizar apuntadores y pasar valor por referencia, casi nunca tendrá que crear variables globales.
9. ¿Qué es la sobrecarga de funciones?  
Es la habilidad de escribir más de una función con el mismo nombre, que se distingan por el número o tipo de sus parámetros.
10. ¿Qué es polimorfismo?  
Es la habilidad de tratar muchos objetos de tipos distintos pero relacionados, sin importar sus diferencias. En C++, el polimorfismo se logra mediante el uso de derivación de clases y funciones virtuales.

## Ejercicios

1. Escriba el prototipo para una función de nombre `Perimetro()` la cual, regresa un valor de tipo `unsigned long int` y acepta dos parámetros, ambos de tipo `unsigned short int`.  
`unsigned long int Perimetro(unsigned short int, unsigned short int);`
2. Escriba la definición de la función `Perimetro()` como se describe en el ejercicio 1. Los dos parámetros representan la longitud y el ancho de un rectángulo. Haga que la función regrese el perímetro (dos veces la longitud más dos veces el ancho).  
`unsigned long int Perimetro(unsigned short int longitud, unsigned short int ancho)`  
{  
    return 2\*longitud + 2\*ancho;  
}

**3. CAZA ERRORES:** ¿Qué está mal en la función del siguiente código?

```
#include <iostream.h>
void miFunc(unsigned short int x);
int main()
{
    unsigned short int x, y;
    y = miFunc(int);
    cout << "x: " << x << " y: " << y << "\n";
}
void miFunc(unsigned short int x)
{
    return (4*x);
}
```

D

La función se declara para regresar void y no puede regresar un valor. Se debe pasar x a miFunc en lugar de int.

**4. CAZA ERRORES:** ¿Qué está mal en la función del siguiente código?

```
#include <iostream.h>;
int miFunc(unsigned short int x);
int main()
{
    unsigned short int x, y;
    y = miFunc(x);
    cout << "x: " << x << " y: " << y << "\n";
}

int miFunc(unsigned short int x)
{
    return (4*x);
}
```

Esta función estaría bien, pero hay un punto y coma al final del encabezado de definición de la función. Además, se utiliza el valor x antes de ser asignado, esto provocará resultados inesperados.

**5. Escriba una función que acepte dos argumentos de tipo unsigned short int y que regrese el resultado de la división del primero entre el segundo. No haga la división si el segundo número es igual a 0, pero regrese el valor -1.**

```
short int Divisor(unsigned short int valUno, unsigned short int
valDos)
{
    if (valDos == 0)
        return -1;
    else
        return valUno / valDos;
}
```

6. Escriba un programa que pida dos números al usuario y que llame a la función que escribió en el ejercicio 5. Imprima la respuesta o imprima un mensaje de error si obtiene -1.

```
#include <iostream.h>
short int Divisor(
    unsigned short int valuno,
    unsigned short int valdos);
int main()
{
    short int uno, dos;
    short int respuesta;
    cout << "Escriba dos números.\n Número uno: ";
    cin >> uno;
    cout << "Número dos: ";
    cin >> dos;
    if ((uno + dos) == 0)
        cout << "Respuesta: " << (-1);
    else
    {
        respuesta = Divisor(uno, dos);
        if (respuesta > -1)
            cout << "Respuesta: " << respuesta;
        else
            cout << "¡Error, no se puede dividir entre cero!";
    }
    return 0;
}
```

7. Escriba un programa que pida un número y una potencia. Escriba una función recursiva que eleve el número a esa potencia. Por ejemplo, si el número es 2 y la potencia es 4, la función debe regresar 16.

```
#include <iostream.h>
typedef unsigned short USHORT;
typedef unsigned long ULONG;
ULONG ObtenerPotencia(USHORT n, USHORT potencia);
int main()
{
    USHORT numero, potencia;
    ULONG respuesta;
    cout << "Escriba un número: ";
    cin >> numero;
    cout << "¿A qué potencia se va a elevar? ";
    cin >> potencia;
    respuesta = ObtenerPotencia(numero, potencia);
    cout << numero << " a la potencia " << power << "es " <<
    respuesta << endl;
    return 0;
}

ULONG ObtenerPotencia(USHORT n, USHORT potencia)
{
    if(potencia == 1)
        return n;
```

```
    else
        return (n * ObtenerPotencia(n,potencia-1));
}
```

## Día 6

### Cuestionario

1. ¿Qué es el operador de punto, y para qué se utiliza?

El operador de punto es el punto (.). Se utiliza para tener acceso a los miembros de la clase.

2. ¿Cuál de las dos acciones reserva memoria, la declaración o la creación?

Las declaraciones de variables reservan memoria. Las declaraciones de clases no reservan memoria (tiene que crear el objeto). Cuando se crea un objeto, se reserva la memoria necesaria para almacenar los datos miembros pero de ninguna forma se reserva memoria para las funciones miembro.

3. ¿Qué es la declaración de una clase, su interfaz o su implementación?

La declaración de una clase es su interfaz; indica a los clientes de la clase cómo interactuar con la misma. La implementación de la clase es el conjunto de funciones miembro guardadas (por lo general, en un archivo CPP relacionado) que dan funcionalidad a la clase.

4. ¿Cuál es la diferencia entre datos miembro públicos y privados?

Los datos miembro públicos pueden ser accedidos por clientes de la clase. Los datos miembro privados pueden ser accedidos sólo por funciones miembro de la clase.

5. ¿Se pueden establecer métodos privados ?

Sí. Tanto los métodos como los datos miembro pueden ser privados.

6. ¿Se pueden establecer datos miembro o públicos?

Aunque los datos miembro pueden ser públicos, es una buena práctica de programación hacerlos privados y proporcionar métodos de acceso públicos a los datos.

7. Si declara dos objetos Gato, ¿pueden éstos tener distintos valores en sus datos miembro suEdad?

Sí. Cada objeto de una clase tiene sus propios datos miembro.

8. ¿Terminan con un punto y coma las declaraciones de clases? ¿Y las definiciones de los métodos de clases?

Las declaraciones terminan con un punto y coma después de la llave de cierre; las definiciones de funciones no.

9. ¿Cuál sería el encabezado para un método de la clase Gato, llamado Maullar, que no toma parámetros y regresa void?

Sería uno como el siguiente:

```
void Gato::Maullar()
```

10. ¿Qué función se llama para inicializar una clase?

El constructor se llama para inicializar cada objeto (cada instancia) de una clase.

## Ejercicios

1. Escriba el código que declare una clase llamada **Empleado** con estos datos miembro:

```
edad, aniosDeServicio, y salario.  
class Empleado  
{  
    int edad;  
    int aniosDeServicio;  
    int salario;  
};
```

2. Vuelva a escribir la clase **Empleado** para hacer los datos miembro privados, y proporcione métodos de acceso públicos para obtener y asignar un valor para cada uno de los datos miembro.

```
class Empleado  
{  
public:  
    int ObtenerEdad() const;  
    void AsignarEdad(int edad);  
    int ObtenerAniosDeServicio() const;  
    void AsignarAniosDeServicio(int anios);  
    int ObtenerSalario() const;  
    void AsignarSalario(int salario);  
  
private:  
    int edad;  
    int aniosDeServicio;  
    int salario;  
};
```

3. Escriba un programa con la clase **Empleado** que cree dos **Empleados**; que asigne un valor a los datos miembro **edad**, **aniosDeServicio** y **salario**, y que imprima sus valores.

```
int main()  
{  
    Empleado John;  
    Empleado Sally;  
    John.AsignarEdad(30);  
    John.AsignarAniosDeServicio(5);  
    John.AsignarSalario(50000);  
  
    Sally.AsignarEdad(32);  
    Sally.AsignarAniosDeServicio(8);  
    Sally.AsignarSalario(40000);  
  
    cout << "En la compañía AcmeCorp, John y Sally tienen el mismo  
trabajo.\n";
```

```
    cout << "John tiene " << John.ObtenerEdad() << " años de edad y ha  
estado en";  
    cout << "la compañía durante " << John.ObtenerAniosDeServicio  
<< "años.\n";  
    cout << "John gana $" << John.ObtenerSalario << " pesos por año.\n\n";  
    cout << "Sally, por otro lado, tiene " << Sally.ObtenerEdad()  
<< "años de edad y ha";  
    cout << "estado con la compañía " << Sally.ObtenerAniosDeServicio;  
    cout << " años. ¡Pero Sally sólo gana $" << Sally.ObtenerSalario();  
    cout << " pesos por año! Aquí hay algo injusto.";  
    return 0;  
}
```

4. Como continuación del ejercicio 3, proporcione un método de la clase Empleado que reporte cuántos miles de pesos gana el empleado, redondeados al múltiplo de 1000 más cercano.

```
long int Empleado::ObtenerMilesRedondeados(int salario) const  
{  
    return (salario+500) / 1000;  
}
```

5. Cambie la clase Empleado de forma que pueda inicializar edad, aniosDeServicio, y salario al crear el empleado.

```
class Empleado  
{  
public:  
  
    Empleado(int edad, int aniosDeServicio, int salario);  
    int ObtenerEdad() const;  
    void AsignarEdad(int edad);  
    int ObtenerAniosDeServicio() const;  
    void AsignarAniosDeServicio(int anios);  
    int ObtenerSalario() const;  
    void AsignarSalario(int salario);  
  
private:  
    int edad;  
    int aniosDeServicio;  
    int salario;  
};
```

6. **CAZA ERRORES:** ¿Qué está mal en la siguiente declaración?

```
class Cuadrado  
{  
public:  
    int lado;  
}
```

Las declaraciones de clases deben terminar con un punto y coma.

7. **CAZA ERRORES:** ¿Por qué no es muy útil la siguiente declaración de clase?

```
class Gato
{
    int ObtenerEdad() const;
private:
    int suEdad;
};
```

El método de acceso `ObtenerEdad()` es privado. Recuerde: todos los miembros de una clase son privados, a menos que se indique lo contrario.

8. **CAZA ERRORES:** ¿Cuáles son los tres errores encontrará el compilador en este código?

```
class TV
{
public:
    void AsignarEstacion(int estacion);
    int ObtenerEstacion() const;
private:
    int suEstacion;
};
int main()
{
    TV miTV;
    miTV.suEstacion = 9;
    TV.AsignarEstacion(10);
    TV miOtraTv(2);
    return 0;
}
```

No se puede tener acceso directamente a `suEstacion`. Es privado.

No se puede llamar a `AsignarEstacion()` en la clase. Puede llamar a `AsignarEstacion()` sólo en objetos.

No se puede inicializar `suEstacion` con el valor 2 ya que no hay constructor relacionado.

## Día 7

### Cuestionario

1. ¿Cómo podemos inicializar más de una variable en un ciclo `for`?

Separando las inicializaciones con comas, como

```
for (x = 0, y = 10; x < 100; x++, y++)
```

2. ¿Por qué se evita el uso de la instrucción `goto`?

`goto` salta en cualquier dirección a cualquier línea arbitraria de código. Esto produce código difícil de entender y por lo tanto difícil de mantener.

3. Es posible escribir un ciclo `for` que tenga un cuerpo que nunca se ejecute?

Sí. Si la condición es `false` después de la inicialización, el cuerpo del ciclo `for` nunca se ejecutará. He aquí un ejemplo:

```
for (int x = 100; x < 100; x++)
```

4. ¿Es posible anidar ciclos `while` dentro de ciclos `for`?

Sí. Cualquier ciclo puede anidarse dentro de cualquier otro ciclo.

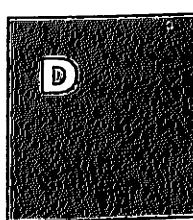
5. ¿Es posible crear un ciclo que nunca termine? Dé un ejemplo.

Sí. A continuación se muestran ejemplos para un ciclo `for` y para un ciclo `while`:

```
for(;;)
{
    // ¡Este ciclo for nunca termina!
}
while(1)
{
    // ¡Este ciclo while nunca termina!
}
```

6. ¿Qué pasa si crea un ciclo que nunca termine?

El programa no se detendrá y pueden pasar muchas cosas. En el mejor de los casos, ocupará mucho tiempo de procesador y reducirá el desempeño de la computadora (en Linux se puede utilizar el comando `kill -9 PID` para terminar la ejecución del programa). En el peor de los casos, se llenarán la pila, el heap y el segmento de datos, habrá un desbordamiento y el sistema se congelará; si la versión del kernel que utiliza no es muy estable o ejecuta el programa como superusuario, deberá reiniciar el equipo.



## Ejercicios

1. ¿Cuál es el valor de `x` cuando el siguiente ciclo `for` finaliza su ejecución?

```
for (int x = 0; x < 100; x++)
100
```

2. Escriba un ciclo `for` anidado que imprima ceros en un patrón de  $10 \times 10$ .

```
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 10; j++)
        cout << "0";
    cout << "\n";
}
```

3. Escriba una instrucción `for` que cuente del 100 al 200 de dos en dos.

```
for (int x = 100; x <= 200; x+=2)
```

4. Escriba un ciclo `while` que cuente del 100 al 200 de dos en dos.

```
int x = 100;
while (x <= 200)
    x+= 2;
```

5. Escriba un ciclo `do...while` que cuente del 100 al 200 de dos en dos.

```
int x = 100;
do
{
    x+=2;
} while (x <= 200);
```

6. CAZA ERRORES: ¿Qué está mal en el siguiente código?

```
int contador = 0
while (contador < 10)
{
    cout << "contador: " << contador;
}
```

`contador` nunca se incrementa y el ciclo `while` nunca terminará.

7. CAZA ERRORES: ¿Qué está mal en el siguiente código?

```
for (int contador = 0; contador < 10; contador++);
    cout << contador << "\n";
```

Hay un punto y coma después del ciclo, y el ciclo no hace nada. Tal vez éste sea el objetivo del programador, pero si se supone que `contador` iba a imprimir cada valor, no lo hará.

8. CAZA ERRORES: ¿Qué está mal en el siguiente código?

```
int contador = 100;
while (contador < 10)
{
    cout << "contador ahora: " << contador;
    contador--;
}
```

`contador` se inicializa con 100, pero la condición de prueba es que si es menor que 10, la prueba fallará y el cuerpo nunca se ejecutará. Si la línea 1 se cambia a `int contador = 5;`, el ciclo no terminará hasta que haya contado más allá del `int` más pequeño posible. Como `int` es de tipo `signed` de manera predeterminada, esto no funcionará como se esperaba.

9. CAZA ERRORES: ¿Qué está mal en el siguiente código?

```
cout << "Escriba un número entre 0 y 5: ";
cin >> elNumero;
switch (elNumero)
{
    case 0:
        hacerCero();
    case 1:           // pasar a la siguiente cláusula case
    case 2:           // pasar a la siguiente cláusula case
    case 3:           // pasar a la siguiente cláusula case
    case 4:           // pasar a la siguiente cláusula case
    case 5:
        hacerUnoHastaCinco();
        break;
    default:
```

```
    hacerPredeterminado();  
    break;  
}
```

Case 0 probablemente necesita una instrucción `break`. De no ser así, debería documentarse con un comentario.

D

## Día 8

### Cuestionario

1. ¿Qué operador se utiliza para determinar la dirección de una variable?

El operador de dirección (`&`) se utiliza para determinar la dirección de cualquier variable.

2. ¿Qué operador se utiliza para encontrar el valor guardado en una dirección que se guarda en un apuntador?

El operador de indirección (`*`) se utiliza para tener acceso al valor en una dirección guardada en un apuntador.

3. ¿Qué es un apuntador?

Es una variable que guarda la dirección de otra variable.

4. ¿Cuál es la diferencia entre la dirección que se guarda en un apuntador y el valor que se encuentra en esa dirección?

La dirección guardada en el apuntador es la dirección de otra variable. El valor guardado en esa dirección es cualquier valor guardado en cualquier variable. El operador de indirección (`*`) regresa el valor guardado en la dirección, que a su vez se guarda en el apuntador.

5. ¿Cuál es la diferencia entre el operador de indirección y el operador de dirección?

El operador de indirección regresa el valor que se encuentra en la dirección guardada en un apuntador. El operador de dirección (`&`) regresa la dirección de memoria de la variable.

6. ¿Cuál es la diferencia entre `const int * apuntUno` e `int * const apuntDos`?

`const int * apuntUno` declara que `apuntUno` es un apuntador a una constante de tipo entero. El entero en sí no puede cambiarse por medio de este apuntador.

`int * const apuntDos` declara que `apuntDos` es un apuntador constante a un entero. Despues de inicializarse, este apuntador no puede ser reasignado.

### Ejercicios

1. ¿Qué hacen estas declaraciones?

- `int * apUno;`
- `int varDos;`
- `int * apTres = &varDos;`

- a. `int * apUno;` declara un apuntador a un entero.
  - b. `int varDos;` declara una variable de tipo entero.
  - c. `int * apTres = &varDos;` declara un apuntador a un entero y lo inicializa con la dirección de otra variable.
2. Si tiene una variable de tipo entero corto sin signo llamada `suEdad`, ¿cómo declararía un apuntador para que manipule a la variable `suEdad`?  
`unsigned short * apEdad = &suEdad;`
3. Asigne el valor **50** a la variable `suEdad` usando el apuntador que declaró en el ejercicio 2.  
`*apEdad = 50;`
4. Escriba un pequeño programa que declare un entero y un apuntador a ese entero. Asígnele al apuntador la dirección del entero. Utilice el apuntador para asignarle un valor a la variable de tipo entero.

```
int elEntero;
int *apEntero = &elEntero;
*apEntero = 5;
```

5. ¿Qué está mal en este código?

```
#include <iostream.h>
int main()
{
    int * apInt;
    *apInt = 9;
    cout << "El valor en apInt: " << *apInt;
    return 0;
}
```

`apInt` debería haber sido inicializado. Lo que es más importante, como no se inicializó y no se le asignó la dirección de ninguna memoria, apunta a un lugar aleatorio en la memoria. Asignar un **9** a ese lugar aleatorio es un error peligroso.

6. **CAZA ERRORES:** ¿Qué está mal con este código?

```
int main()
{
    int UnaVariable = 5;
    cout << "UnaVariable: " << UnaVariable << "\n";
    int * apVar = &UnaVariable;
    apVar = 9;
    cout << "UnaVariable: " << *apVar << "\n";
    return 0;
}
```

Tal vez el programador quiso asignar un **9** al valor contenido en el apuntador `apVar`. Desafortunadamente, el **9** se asignó como valor de `apVar` debido a que se omitió el operador de indirección (`*`). Esto provocará un desastre si `*apVar` se utiliza para asignar un valor.

## Día 9

### Cuestionario

1. ¿Cuál es la diferencia entre una referencia y un apuntador?

Una referencia es un alias, y un apuntador es una variable que guarda una dirección.  
Las referencias no pueden ser nulas y no pueden ser reasignadas.

2. ¿Cuándo debe utilizar un apuntador en vez de una referencia?

Cuando se necesite reasignar a lo que se está apuntando, o cuando el apuntador pueda ser nulo.

3. ¿Qué regresa new si no hay memoria suficiente para crear el nuevo objeto?

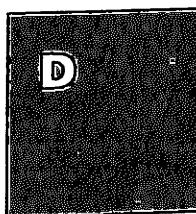
Un apuntador nulo (`0`).

4. ¿Qué es una referencia constante?

Es una forma abreviada de decir “una referencia a un objeto constante”.

5. ¿Cuál es la diferencia entre pasar por referencia y pasar una referencia?

Pasar por referencia significa no hacer una copia local. Puede lograrse al pasar una referencia o al pasar un apuntador. Pasar una referencia es una forma del paso por referencia, pero no es la única.



### Ejercicios

1. Escriba un programa que declare un `int`, una referencia a un `int`, y un apuntador a un `int`. Use el apuntador y la referencia para manipular el valor contenido en el `int`.

```
int main()
{
    int varUno;
    int & rVar = varUno;
    int * apVar = &varUno;
    rVar = 5;
    *apVar = 7;
    return 0;
}
```

2. Escriba un programa que declare un apuntador constante a un entero constante. Inicialice el apuntador a una variable entera, `varUno`. Asigne un 6 a `varUno`. Use el apuntador para asignar un 7 a `varUno`. Cree otra variable de tipo entero que se llame `varDos`. Reasigne el apuntador a `varDos`. No compile todavía este ejercicio.

```
int main()
{
    int varUno;
    const int * const apVar = &varUno;
    varUno = 6;
    *apVar = 7;
```

```
    int varDos;
    apVar = &varDos;
return 0;
}
```

3. Ahora compile el programa del ejercicio 2. ¿Qué es lo que produce errores? ¿Qué es lo que produce advertencias?

No se puede asignar un valor a un objeto constante, y no se puede reasignar un apuntador constante.

4. Escriba un programa que origine un apuntador perdido.

```
int main()
{
    int * apVar;
    *apVar = 9;
    return 0;
}
```

5. Corrija el programa del ejercicio 4.

```
int main()
{
    int varUno;
    int * apVar = &varUno;
    *apVar = 9;
    return 0;
}
```

6. Escriba un programa que origine una fuga de memoria.

```
#include <iostream.h>

int * FuncUno();
int main()
{
    int * apInt = FuncUno();
    cout << "el valor de apInt estando de regreso en main es:
    " << *apInt << endl;
    return 0;
}

int * FuncUno()
{
    int * apInt = new int (5);
    cout << "el valor de pInt en FuncUno es: " << *apInt << endl;
    return apInt;
}
```

7. Corrija el programa del ejercicio 6.

```
#include <iostream.h>

int FuncUno();
int main()
```

```
{  
    int elInt = FuncUno();  
    cout << "el valor de apInt estando de regreso en main es:  
    " << elInt << endl;  
    return 0;  
}  
  
int FuncUno()  
{  
    int * apInt = new int (5);  
    int temp;  
    cout << "el valor de apInt en FuncUno es: " << *apInt << endl;  
    temp = *apInt;  
    delete apInt;  
    return temp;  
}
```

8. ¿Qué está mal en este programa?

```
1:      #include <iostream.h>  
2:  
3:      class GATO  
4:      {  
5:          public:  
6:              GATO(int edad) { suEdad = edad; }  
7:              ~GATO(){}  
8:              int ObtenerEdad() const { return suEdad; }  
9:          private:  
10:             int suEdad;  
11:      };  
12:  
13:      GATO & CrearGato(int edad);  
14:      int main()  
15:      {  
16:          int edad = 7;  
17:          GATO Silvestre = CrearGato(edad);  
18:          cout << "Silvestre tiene " << Silvestre.ObtenerEdad()  
19:          << " años de edad\n";  
20:          return 0;  
21:      }  
22:  
23:      GATO & CrearGato(int edad)  
24:      {  
25:          GATO * apGato = new GATO(edad);  
26:          return *apGato;  
27:      }
```

CrearGato regresa una referencia al objeto GATO creado en el heap. No hay manera de liberar esa memoria, y esto produce una fuga de memoria.

9. Corrija el programa del ejercicio 8.

```
1:      #include <iostream.h>  
2:  
3:      class GATO
```

```
4:      {
5:          public:
6:              GATO(int edad) { suEdad = edad; }
7:              ~GATO(){}
8:              int ObtenerEdad() const { return suEdad; }
9:          private:
10:             int suEdad;
11:         };
12:
13:     GATO * CrearGato(int edad);
14:     int main()
15:     {
16:         int edad = 7;
17:         GATO * Silvestre = CrearGato(edad);
18:         cout << "Silvestre tiene " << Silvestre->ObtenerEdad()
19:             << " años de edad\n";
20:         delete Silvestre;
21:         return 0;
22:     }
23:
24:     GATO * CrearGato(int edad)
25:     {
26:         return new GATO(edad);
27:     }
```

## Día 10

### Cuestionario

1. Al sobrecargar funciones miembro de una clase, ¿de qué manera deben diferir?

Las funciones miembro sobrecargadas son funciones en una clase que comparten un nombre pero difieren en el número o en el tipo de sus parámetros.

2. ¿Cuál es la diferencia entre una declaración y una definición?

Cuando se habla de funciones y clases, la declaración nos muestra su prototipo o interfaz, y la definición establece la implementación. Cuando se habla de variables y tipos, la definición especifica las características particulares de cada tipo (y por consecuencia de cada variable creada). Las definiciones de los tipos integrados son intrínsecas a la plataforma y al compilador. Por otra parte, la declaración de una variable reserva memoria para su uso posterior.

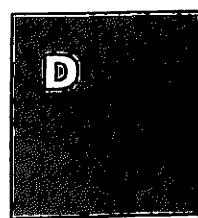
3. ¿Cuándo se llama al constructor de copia?

Siempre que se crea una copia temporal de un objeto. Esto ocurre cada vez que un objeto se pasa por valor.

4. ¿Cuándo se llama al destructor?

El destructor se llama cada vez que se destruye un objeto, ya sea porque queda fuera de alcance o porque se llama a `delete` para que actúe sobre un apuntador que está apuntando a ese objeto.

5. ¿Qué diferencia hay entre el constructor de copia y el operador de asignación (=)?  
El operador de asignación actúa sobre un objeto existente; el constructor de copia crea uno nuevo.
6. ¿Qué es el apuntador `this`?  
Es un parámetro oculto en cada función miembro que apunta al objeto en sí.
7. ¿Cómo puede diferenciar entre la sobrecarga de los operadores de incremento de prefijo y los de posfijo?  
El operador de prefijo no toma parámetros. El operador de posfijo toma un solo parámetro `int`, el cual se utiliza como señal para el compilador que le indica que ésta es la variante de posfijo.
8. ¿Puede sobrecargar el `operator+` para el tipo de datos `short int`?  
No, no es posible sobrecargar ningún operador para los tipos integrados.
9. ¿Es válido en C++ sobrecargar el `operator++` para que decremente un valor de su clase?  
Es válido, pero no es buena idea. Los operadores deben sobrecargarse en una forma que pueda ser entendida por cualquier persona que lea el código.
10. ¿Qué valor de retorno deben tener los operadores de conversión en sus declaraciones?  
Ninguno. Al igual que los constructores y los destructores, no tienen valor de retorno.



## Ejercicio

1. Escriba la declaración de una clase llamada `CirculoSencillo` (únicamente la declaración) con una variable miembro: `suRadio`. Incluya un constructor predeterminado, un destructor y métodos de acceso para la variable `suRadio`.

```
class CirculoSencillo
{
public:
    CirculoSencillo();
    ~CirculoSencillo();
    void AsignarRadio(int);
    int ObtenerRadio();
private:
    int suRadio;
};
```

2. Usando la clase que creó en el ejercicio 1, escriba la implementación del constructor predeterminado, e inicialice `suRadio` con el valor 5.

```
CirculoSencillo::CirculoSencillo():
    suRadio(5)
{}
```

3. Usando la misma clase, agregue un segundo constructor que tome un valor como parámetro y asigne ese valor a suRadio.

```
CirculoSencillo::CirculoSencillo(int radio):
    suRadio(radio)
{}
```

4. Cree un operador de incremento de prefijo y uno de posfijo para su clase CirculoSencillo, que incremente suRadio.

```
const CirculoSencillo & CirculoSencillo::operator++()
{
    ++(suRadio);
    return *this;
}

// Operator ++(int) posfijo.
// Obtener y luego incrementar
const CirculoSencillo CirculoSencillo::operator++ (int)
{
    // declarar CirculoSencillo como local e inicializar al valor de *this
    CirculoSencillo temp(*this);
    ++(suRadio);
    return temp;
}
```

5. Cambie la clase CirculoSencillo para que guarde el dato miembro suRadio en el heap, y corrija los métodos existentes.

```
class CirculoSencillo
{
public:
    CirculoSencillo();
    CirculoSencillo(int);
    ~CirculoSencillo();
    void AsignarRadio(int);
    int ObtenerRadio();
    const CirculoSencillo & operator++();
    const CirculoSencillo operator++(int);
private:
    int * suRadio;
};

CirculoSencillo::CirculoSencillo()
{
    suRadio = new int(5);
}

CirculoSencillo::CirculoSencillo(int radio)
{
    suRadio = new int(radio);
}

CirculoSencillo::~CirculoSencillo()
{
```

```
        delete suRadio;
    }

const CirculoSencillo& CirculoSencillo::operator++()
{
    ++(*suRadio);
    return *this;
}

// Operator ++(int) posfijo.
// Obtener y luego incrementar
const CirculoSencillo CirculoSencillo::operator++ (int)
{
    // declarar CirculoSencillo como local e inicializar al valor de *this
    CirculoSencillo temp(*this);
    ++(*suRadio);
    return temp;
}
```

D

6. Proporcione un constructor de copia para CirculoSencillo.

```
CirculoSencillo::CirculoSencillo(const CirculoSencillo & rhs)
{
    int val = rhs.ObtenerRadio();
    suRadio = new int(val);
}
```

7. Proporcione un operador de asignación para CirculoSencillo.

```
CirculoSencillo& CirculoSencillo::operator=(const CirculoSencillo & rhs)
{
    if (this == &rhs)
        return *this;
    delete suRadio;
    suRadio = new int;
    *suRadio = rhs.ObtenerRadio();
    return *this;
}
```

8. Escriba un programa que cree dos objetos CirculoSencillo. Utilice el constructor predeterminado en uno y cree una instancia con el otro que tenga el valor 9. Llame al operador de incremento para que actúe sobre cada uno y luego imprima sus valores. Por último, asigne el segundo al primero e imprima sus valores.

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

```
class CirculoSencillo
{
public:
    // constructores
    CirculoSencillo();
    CirculoSencillo(int);
    CirculoSencillo(const CirculoSencillo &);
```

```
-CirculoSencillo() {}

// funciones de acceso
void AsignarRadio(int);
int ObtenerRadio()const;

// operadores
const CirculoSencillo& operator++();
const CirculoSencillo operator++(int);
CirculoSencillo& operator=(const CirculoSencillo &);

private:
    int *suRadio;
};

CirculoSencillo::CirculoSencillo()
{suRadio = new int(5);}

CirculoSencillo::CirculoSencillo(int radio)
{suRadio = new int(radio);}

CirculoSencillo::CirculoSencillo(const CirculoSencillo & rhs)
{
    int val = rhs.ObtenerRadio();
    suRadio = new int(val);
}
CirculoSencillo::~CirculoSencillo()
{
    delete suRadio;
}
CirculoSencillo& CirculoSencillo::operator=(const CirculoSencillo & rhs)
{
    if (this == &rhs)
        return *this;
    *suRadio = rhs.ObtenerRadio();
    return *this;
}

const CirculoSencillo& CirculoSencillo::operator++()
{
    ++(*suRadio);
    return *this;
}

// Operator ++(int) posfijo.
// Obtener y luego incrementar
const CirculoSencillo CirculoSencillo::operator++ (int)
{
    // declarar CirculoSencillo como local e inicializar al valor de *this
    CirculoSencillo temp(*this);
```

```

        ++(*suRadio);
        return temp;
    }
    int CirculoSencillo::ObtenerRadio() const
    {
        return *suRadio;
    }
    int main()
    {
        CirculoSencillo CirculoUno, CirculoDos(9);
        CirculoUno++;
        ++CirculoDos;
        cout << "CirculoUno: " << CirculoUno.ObtenerRadio() << endl;
        cout << "CirculoDos: " << CirculoDos.ObtenerRadio() << endl;
        CirculoUno = CirculoDos;
        cout << "CirculoUno: " << CirculoUno.ObtenerRadio() << endl;
        cout << "CirculoDos: " << CirculoDos.ObtenerRadio() << endl;
    return 0;
}

```

- 9. CAZA ERRORES:** ¿Qué está mal en esta implementación del operador de asignación?

```

CUADRADO CUADRADO ::operator=(const CUADRADO & rhs)
{
    suLado = new int;
    *suLado = rhs.ObtenerLado();
    return *this;
}

```

Debe verificar si rhs es igual a this, o la llamada a a = a hará que su programa falle.

- 10. CAZA ERRORES:** ¿Qué está mal en esta implementación del operador de suma?

```

MuyCorto MuyCorto::operator+ (const MuyCorto& rhs)
{
    suVal += rhs.ObtenerSuVal();
    return *this;
}

```

Este operator+ está cambiando el valor de uno de los operandos, en lugar de crear un nuevo objeto MuyCorto con la suma. La manera correcta de hacer esto es la siguiente:

```

MuyCorto MuyCorto::operator+ (const MuyCorto& rhs)
{
    return MuyCorto(suVal + rhs.ObtenerSuVal());
}

```

## Día 11

### Cuestionario

1. ¿Qué es una tabla-v?

Una tabla-v, o tabla de funciones virtuales, es una forma común de que los compiladores manejen las funciones virtuales en C++. La tabla mantiene una lista de las direcciones de todas las funciones virtuales y, dependiendo del tipo del objeto al que se apunte en tiempo de ejecución, invoca a la función apropiada.

2. ¿Qué es un destructor virtual?

Un destructor de cualquier clase puede declararse como virtual. Cuando se elimina el apuntador, el tipo del objeto es valorado en tiempo de ejecución y se invoca al destructor derivado apropiado.

3. ¿Cómo se puede mostrar la declaración de un constructor virtual?

No hay constructores virtuales.

4. ¿Cómo se puede crear un constructor virtual de copia?

Creando un método virtual en la clase, que a su vez llame al constructor de copia.

5. ¿Cómo se invoca a una función miembro de la clase base desde una clase derivada en la que se haya redefinido esa función?

`ClaseBase::NombreDeFuncion();`

6. ¿Cómo se invoca a una función miembro de la clase base desde una clase derivada en la que no se haya redefinido esa función?

`NombreDeFuncion();`

7. Si una clase base declara una función como virtual, y una clase derivada no utiliza el término virtual cuando redefina esa clase, ¿seguirá siendo virtual cuando la herede una clase de tercera generación?

Sí, la virtualidad es heredada y no puede desactivarse.

8. ¿Para qué se utiliza la palabra reservada `protected`?

Los miembros `protected` son accesibles para las funciones miembro de los objetos derivados.

### Ejercicios

1. Muestre la declaración de una función virtual que tome un parámetro entero y regrese `void`.

`virtual void UnaFuncion(int);`

2. Muestre la declaración de una clase llamada Cuadrado, la cual se deriva de Rectangulo, que a su vez se deriva de Figura.

```
class Cuadrado : public Rectangulo  
{};
```

3. Si, en el ejercicio 2, Figura no toma parámetros, Rectangulo toma dos (longitud y ancho), pero Cuadrado toma sólo un parámetro (longitud), muestre la inicialización del constructor para Cuadrado.

```
Cuadrado::Cuadrado(int longitud);  
    Rectangulo(longitud, longitud){}
```

4. Escriba un constructor de copia virtual para la clase Cuadrado (del ejercicio 3).

```
class Cuadrado  
{  
public:  
    // ...  
    virtual Cuadrado * clone() const { return new Cuadrado(*this); }  
    // ...  
};
```

5. **CAZA ERRORES:** ¿Qué está mal en este segmento de código?

```
void UnaFuncion (Figura);  
Figura * apRect = new Rectangulo;  
UnaFuncion(*apRect);
```

Tal vez nada. UnaFuncion espera un objeto Figura. Le ha pasado un Rectangulo "rebanado" como una Figura. Siempre y cuando no necesite ninguna de las partes de Rectangulo, esto estará bien. Si necesita las partes de Rectangulo, necesita modificar a UnaFuncion para que tome un apuntador o una referencia a una Figura.

6. **CAZA ERRORES:** ¿Qué está mal en este segmento de código?

```
class Figura()  
{  
public:  
    Figura();  
    virtual ~Figura();  
    virtual Figura(const Figura &);  
};
```

No se puede declarar un constructor de copia como virtual.

D

## Día 12

### Cuestionario

1. ¿Cuáles son el primero y último elementos en `UnArreglo[25]`?  
`UnArreglo[0], UnArreglo[24]`
2. ¿Cómo se declara un arreglo multidimensional?  
 Se escribe un conjunto de subíndices para cada dimensión. Por ejemplo, `UnArreglo[2][3][2]` es un arreglo de tres dimensiones. La primera dimensión tiene dos elementos, la segunda tiene tres, y la tercera tiene dos.
3. Inicialice los miembros del arreglo de la pregunta 2.  
`UnArreglo[2][3][2] = { { {1,2},{3,4},{5,6} } , { {7,8},{9,10},{11,12} } };`
4. ¿Cuántos elementos hay en el arreglo `UnArreglo[10][5][20]`?  
 $10 \times 5 \times 20 = 1,000$
5. ¿Cuál es el número máximo de elementos que se pueden agregar a una lista enlazada?  
 No hay un máximo fijo. Depende de cuánta memoria haya disponible.
6. ¿Puede utilizar notación de subíndice en una lista enlazada?  
 La puede utilizar sólo si escribe su propia clase para que contenga la lista enlazada y se sobrecargue el operador de subíndice.
7. ¿Cuál es el último carácter de la cadena “Brad es una buena persona”?  
 El carácter nulo.

### Ejercicios

1. Declare un arreglo de dos dimensiones que represente un tablero del juego tic-tac-toe.  
`int Tablero[3][3];`
2. Escriba el código que inicialice con 0 todos los elementos del arreglo que creó en el ejercicio 1.  
`int Tablero[3][3] = { {0,0,0},{0,0,0},{0,0,0} }`
3. Escriba la declaración de una clase `Nodo` que guarde enteros.  

```
class Nodo
{
public:
    Nodo ();
    Nodo (int);
    ~Nodo();
    void AsignarSiguiente(Nodo * node) { suSiguiente = node; }
```

```
Nodo * ObtenerSiguiente() const { return suSiguiente; }
int ObtenerValor() const { return suVal; }
void Insertar(Nodo *);
void Mostrar();
private:
    int suVal;
    Nodo * suSiguiente;
};
```

4. **CAZA ERRORES:** ¿Qué está mal en este fragmento de código?

```
unsigned short UnArreglo[5][4];
for (int i = 0; i<4; i++)
    for (int j = 0; j<5; j++)
        UnArreglo[i][j] = i+j;
```

El arreglo es de 5 elementos por 4 elementos, pero el código inicializa  $4 \times 5$ .

D

5. **CAZA ERRORES:** ¿Qué está mal en este fragmento de código?

```
unsigned short UnArreglo[5][4];
for (int i = 0; i<=5; i++)
    for (int j = 0; j<=4; j++)
        UnArreglo[i][j] = 0;
```

Quería escribir  $i < 5$ , pero en vez de eso escribió  $i \leq 5$ . El código se ejecutará cuando  $i == 5$  y  $j == 4$ , pero no existe el elemento  $\text{UnArreglo}[5][4]$ .

## Día 13

### Cuestionario

1. ¿Qué es una conversión descendente?

Una conversión descendente (también conocida como “convertir hacia abajo”) es una declaración que indica que un apuntador a una clase base debe tratarse como un apuntador a una clase derivada.

2. ¿Qué es el *aptrv*?

El *aptrv*, o apuntador a función virtual, es un detalle de implementación de las funciones virtuales. Cada objeto de una clase con funciones virtuales tiene un *aptrv*, el cual apunta a la tabla de funciones virtuales para esa clase.

3. Si un rectángulo “redondo” tiene bordes rectos y esquinas redondeadas, y su clase *RectRedondo* hereda tanto de *Rectangulo* como de *Circulo*, y éstos a su vez heredan de *Figura*, ¿cuántas Figuras se crearán cuando cree un *RectRedondo*?

Si ninguna clase hereda usando la palabra reservada *virtual*, se crean dos Figuras: una para *Rectangulo* y otra para *Circulo*. Si se utiliza la palabra reservada *virtual* para ambas clases, sólo se crea una Figura compartida.

4. Si **Caballo** y **Ave** heredan de **Animal** usando herencia virtual pública, ¿inicializan sus constructores el constructor de **Animal**? Si **Pegaso** hereda tanto de **Caballo** como de **Ave**, ¿cómo inicializa el constructor de **Animal**?

Tanto **Caballo** como **Ave** inicializan su clase base **Animal** en sus constructores. **Pegaso** lo hace también, y cuando se crea un **Pegaso**, se ignoran las inicializaciones de **Caballo** y **Ave** para **Animal**.

5. Declare una clase llamada **Vehiculo** y conviértala en un tipo de datos abstracto.

```
class Vehiculo
{
    virtual void Mover() = 0;
}
```

6. Si una clase base es un ADT, y tiene tres funciones virtuales puras, ¿cuántas de estas funciones se deben redefinir en sus clases derivadas?

Ninguna debe redefinirse, a menos que quiera hacer que la clase no sea abstracta, en cuyo caso las tres deben redefinirse.

## Ejercicios

1. Muestre la declaración de una clase llamada **AvionJet**, que herede de **Cohete** y de **Avion**.

```
class AvionJet : public Cohete, public Avion
```

2. Muestre la declaración de una clase llamada **777**, que herede de la clase **AvionJet** descrita en el ejercicio 1.

```
class 777 : public AvionJet
```

3. Escriba un programa que derive a **Auto** y a **Camion** de la clase **Vehiculo**. Convierta a **Vehiculo** en un ADT que tenga dos funciones virtuales puras. Haga que **Auto** y **Camion** no sean ADTs.

```
class Vehiculo
{
    virtual void Mover() = 0;
    virtual void Remolcar() = 0;
};

class Auto : public Vehiculo
{
    virtual void Mover();
    virtual void Remolcar();
};

class Camion : public Vehiculo
{
    virtual void Mover();
    virtual void Remolcar();
};
```

4. Modifique el programa del ejercicio 3 de forma que Auto sea un ADT, y derive de Auto a AutoDeportivo, Vagoneta, y Sedan. En la clase Auto, proporcione una implementación para una de las funciones virtuales puras de Vehiculo y hágala no pura.

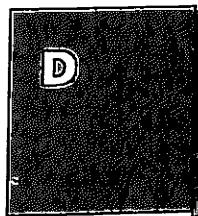
```
class Vehiculo
{
    virtual void Mover() = 0;
    virtual void Remolcar() = 0;
};

class Auto : public Vehiculo
{
    virtual void Mover();
};

class Camion : public Vehiculo
{
    virtual void Mover();
    virtual void Remolcar();
};

class AutoDeportivo : public Auto
{
    virtual void Remolcar();
};

class Sedan : public Auto
{
    virtual void Remolcar();
};
```



## Día 14

### Cuestionario

1. ¿Pueden las variables miembro estáticas ser privadas?

Sí. Son variables miembro, y su acceso puede controlarse igual que cualquier otra. Si son privadas, se puede tener acceso a ellas solamente mediante el uso de funciones miembro o, lo que es más común, por medio de funciones miembro estáticas.

2. Muestre la declaración de una variable miembro estática.

```
static int suEstatica;
```

3. Muestre la declaración de una función estática.

```
static int UnaFuncion();
```

4. Muestre la declaración de un apuntador a una función que regrese un long y que tome un parámetro entero.

```
long (* Funcion)(int);
```

5. Modifique el apuntador de la pregunta 4 para que sea un apuntador a una función miembro de la clase Auto.

```
long (Auto::*Funcion)(int);
```

6. Muestre la declaración de un arreglo de 10 apuntadores como los de la pregunta 5.

```
typedef long (Auto::*Funcion)(int);
Funcion elArreglo[10];
```

Otros compiladores (que no sean de GNU) tal vez prefieran lo siguiente:

```
long (Auto::*Funcion)(int) elArreglo [10];
```

## Ejercicios

1. Escriba un programa corto que declare una clase con una variable miembro y una variable miembro estática. Haga que el constructor inicialice la variable miembro e incremente la variable miembro estática. Haga que el destructor decremente la variable miembro estática.

```
1:      class miClase
2:      {
3:      public:
4:          miClase();
5:          ~miClase();
6:      private:
7:          int suMiembro;
8:          static int suEstatica;
9:      };
10:
11:     miClase::miClase():
12:         suMiembro(1)
13:     {
14:         suEstatica++;
15:     }
16:
17:     miClase::~miClase()
18:     {
19:         suEstatica--;
20:     }
21:
22:     int miClase::suEstatica = 0;
23:
24:     int main()
25:     {}
```

2. Usando el programa del ejercicio 1, escriba un programa controlador corto que cree tres objetos y luego despliegue sus variables miembro y la variable miembro

estática. Luego destruya cada objeto y muestre el efecto en la variable miembro estática.

```
1:      #include <iostream.h>
2:
3:      class miClase
4:      {
5:          public:
6:              miClase();
7:              ~miClase();
8:              void MostrarMiembro();
9:              void MostrarEstatica();
10:         private:
11:             int suMiembro;
12:             static int suEstatica;
13:     };
14:
15:     miClase::miClase():
16:         suMiembro(1)
17:     {
18:         suEstatica++;
19:     }
20:
21:     miClase::~miClase()
22:     {
23:         suEstatica--;
24:         cout << "En destructor. suEstatica: " << suEstatica << endl;
25:     }
26:
27:     void miClase::MostrarMiembro()
28:     {
29:         cout << "suMiembro: " << suMiembro << endl;
30:     }
31:
32:     void miClase::MostrarEstatica()
33:     {
34:         cout << "suEstatica: " << suEstatica << endl;
35:     }
36:     int miClase::suEstatica = 0;
37:
38:     int main()
39:     {
40:         miClase obj1;
41:         obj1.MostrarMiembro();
42:         obj1.MostrarEstatica();
43:
44:         miClase obj2;
45:         obj2.MostrarMiembro();
46:         obj2.MostrarEstatica();
47:
48:         miClase obj3;
49:         obj3.MostrarMiembro();
```

```
50:     obj3.MostrarEstatica();
51:     return 0;
52: }
```

3. Modifique el programa del ejercicio 2 para utilizar una función miembro estática que permita el acceso a la variable miembro estática. Haga que la variable miembro estática sea privada.

```
1: #include <iostream.h>
2:
3: class miClase
4: {
5: public:
6:     miClase();
7:     ~miClase();
8:     void MostrarMiembro();
9:     static int ObtenerEstatica();
10: private:
11:     int suMiembro;
12:     static int suEstatica;
13: };
14:
15: miClase::miClase():
16:     suMiembro(1)
17: {
18:     suEstatica++;
19: }
20:
21: miClase::~miClase()
22: {
23:     suEstatica--;
24:     cout << "En destructor. suEstatica: " << suEstatica << endl;
25: }
26:
27: void miClase::MostrarMiembro()
28: {
29:     cout << "suMiembro: " << suMiembro << endl;
30: }
31:
32: int miClase::suEstatica = 0;
33:
34: void miClase::ObtenerEstatica()
35: {
36:     return suEstatica;
37: }
38:
39: int main()
40: {
41:     miClase obj1;
42:     obj1.MostrarMiembro();
43:     cout << "Estática: " << miClase::ObtenerEstatica() << endl;
44: }
```

```
45:     miClase obj2;
46:     obj2.MostrarMiembro();
47:     cout << "Estática: " << miClase::ObtenerEstatica() << endl;
48:
49:     miClase obj3;
50:     obj3.MostrarMiembro();
51:     cout << "Estática: " << miClase::ObtenerEstatica() << endl;
52:     return 0;
53: }
```

4. Escriba un apuntador a una función miembro para que tenga acceso a los datos miembro que no sean estáticos del programa del ejercicio 3, y utilice ese apuntador para imprimir el valor de esos datos.

```
1: #include <iostream.h>
2:
3: class miClase
4: {
5: public:
6:     miClase();
7:     ~miClase();
8:     void MostrarMiembro();
9:     static int ObtenerEstatica();
10: private:
11:     int suMiembro;
12:     static int suEstatica;
13: };
14:
15: miClase::miClase():
16:     suMiembro(1)
17: {
18:     suEstatica++;
19: }
20:
21: miClase::~miClase()
22: {
23:     suEstatica--;
24:     cout << "En destructor. suEstatica: " << suEstatica << endl;
25: }
26:
27: void miClase::MostrarMiembro()
28: {
29:     cout << "suMiembro: " << suMiembro << endl;
30: }
31:
32: int miClase::suEstatica = 0;
33:
34: int miClase::ObtenerEstatica()
35: {
36:     return suEstatica;
37: }
38:
```

D

```
39:     int main()
40:     {
41:         void (miClase::*AFM) ();
42:
43:         AFM=miClase::MostrarMiembro;
44:
45:         miClase obj1;
46:         (obj1.*AFM)();
47:         cout << "Estática: " << miClase::ObtenerEstatica() << endl;
48:
49:         miClase obj2;
50:         (obj2.*AFM)();
51:         cout << "Estática: " << miClase::ObtenerEstatica() << endl;
52:
53:         miClase obj3;
54:         (obj3.*AFM)();
55:         cout << "Estática: " << miClase::ObtenerEstatica() << endl;
56:         return 0;
57:     }
```

5. Agregue dos variables miembro más a la clase de las preguntas anteriores. Agregue funciones de acceso que obtengan el valor de estos valores y proporcione a todas las funciones miembro los mismos valores de retorno y firmas. Utilice el apuntador a una función miembro para tener acceso a estas funciones.

```
1:     #include <iostream.h>
2:
3:     class miClase
4:     {
5:     public:
6:         miClase();
7:         ~miClase();
8:         void MostrarMiembro();
9:         void MostrarSegunda();
10:        void MostrarTercera();
11:        static int ObtenerEstatica();
12:    private:
13:        int suMiembro;
14:        int suSegunda;
15:        int suTercera;
16:        static int suEstatica;
17:    };
18:
19:    miClase::miClase():
20:        suMiembro(1),
21:        suSegunda(2),
22:        suTercera(3)
23:    {
24:        suEstatica++;
25:    }
```

```
26: 
27:     miClase::~miClase()
28:     {
29:         suEstatica--;
30:         cout << "En destructor. suEstatica: " << suEstatica << endl;
31:     }
32: 
33: void miClase::MostrarMiembro()
34: {
35:     cout << "suMiembro: " << suMiembro << endl;
36: }
37: 
38: void miClase::MostrarSegunda()
39: {
40:     cout << "suSegunda: " << suSegunda << endl;
41: }
42: 
43: void miClase::MostrarTercera()
44: {
45:     cout << "suTercera: " << suTercera << endl;
46: }
47: int miClase::suEstatica = 0;
48: 
49: int miClase::ObtenerEstatica()
50: {
51:     return suEstatica;
52: }
53: 
54: int main()
55: {
56:     void (miClase::*AFM) ();
57: 
58:     miClase obj1;
59:     AFM=miClase::MostrarMiembro;
60:     (obj1.*AFM)();
61:     AFM=miClase::MostrarSegunda;
62:     (obj1.*AFM)();
63:     AFM=miClase::MostrarTercera;
64:     (obj1.*AFM)();
65:     cout << "Estática: " << miClase::ObtenerEstatica() << endl;
66: 
67:     miClase obj2;
68:     AFM=miClase::MostrarMiembro;
69:     (obj2.*AFM)();
70:     AFM=miClase::MostrarSegunda;
71:     (obj2.*AFM)();
72:     AFM=miClase::MostrarTercera;
73:     (obj2.*AFM)();
74:     cout << "Estática: " << miClase::ObtenerEstatica() << endl;
75: 
76:     miClase obj3;
```

```
77:     AFM=miClase::MostrarMiembro;
78:     (obj3.*AFM)();
79:     AFM=miClase::MostrarSegunda;
80:     (obj3.*AFM)();
81:     AFM=miClase::MostrarTercera;
82:     (obj3.*AFM)();
83:     cout << "Estática: " << miClase::ObtenerEstatica() << endl;
84:     return 0;
85: }
```

## Día 15

### Cuestionario

1. ¿Cómo se establece una relación del tipo *es un*?  
Mediante la herencia pública.
2. ¿Cómo se establece una relación del tipo *tiene un*?  
Mediante la contención; es decir, una clase tiene un miembro que es un objeto de otro tipo.
3. ¿Cuál es la diferencia entre contención y delegación?  
La contención describe el concepto de que una clase tiene un dato miembro que es un objeto de otro tipo. La delegación expresa el concepto de que una clase utiliza a otra clase para realizar una tarea u objetivo. La delegación se logra por lo general mediante la contención.
4. ¿Cuál es la diferencia entre delegación e *implementación con base en*?  
La delegación expresa el concepto de que una clase utiliza a otra clase para realizar una tarea u objetivo. *Implementación con base en* expresa el concepto de heredar la implementación de otra clase.
5. ¿Qué es una función friend?  
Una función friend (amiga) es una función declarada para tener acceso a los miembros privados y protegidos de su clase.
6. ¿Qué es una clase friend?  
Una clase friend (amiga) es una clase declarada de forma que todas sus funciones miembro sean funciones amigas de su clase.
7. Si Perro es amigo de Muchacho, ¿Muchacho es amigo de Perro?  
No, la amistad no es commutativa.
8. Si Perro es amigo de Muchacho y Terrier se deriva de Perro, ¿Terrier es amigo de Muchacho?  
No, la amistad no se hereda.
9. Si Perro es amigo de Muchacho y Muchacho es amigo de Casa, ¿Perro es amigo de Casa?

No, la amistad no es transitiva.

10. ¿Dónde debe aparecer la declaración de una función friend?

En cualquier lugar dentro de la declaración de la clase. No importa si se coloca la declaración dentro del área de acceso public:, protected: o private:.

## Ejercicios

1. Muestre la declaración de una clase llamada Animal, que contenga un dato miembro que sea un de objeto tipo cadena.

```
class Animal:  
{  
private:  
    String suNombre;  
};
```

2. Muestre la declaración de una clase llamada ArregloLimitado, que sea un arreglo.

```
class arregloLimitado : public Arreglo  
{  
// ...  
}
```

3. Muestre la declaración de una clase llamada Conjunto, que se declare con base en un arreglo.

```
class Conjunto : private Arreglo  
{  
// ...  
}
```

4. Modifique el listado 15.1 para proporcionar a la clase Cadena un operador de inserción (<<).

```
1: #include <iostream.h>  
2: #include <string.h>  
3:  
4: class Cadena  
5: {  
6:     public:  
7:         // constructores  
8:         Cadena();  
9:         Cadena(const char *const);  
10:        Cadena(const Cadena &);  
11:        ~Cadena();  
12:  
13:        // operadores sobrecargados  
14:        char & operator[](int desplazamiento);  
15:        char operator[](int desplazamiento) const;  
16:        Cadena operator+(const Cadena&);  
17:        void operator+=(const Cadena&);  
18:        Cadena & operator=(const Cadena &);  
19:        friend ostream& operator<<  
20:            (ostream& _elFlujo, Cadena& laCadena);  
21:        friend istream& operator>>
```

```
22:             (istream& _elFlujo,Cadena& laCadena);
23:             // Métodos generales de acceso
24:             int ObtenerLongitud()const { return suLongitud; }
25:             const char * ObtenerCadena() const { return suCadena; }
26:             // static int ConstructorCuenta;
27:
28:         private:
29:             Cadena (int);           // constructor privado
30:             char * suCadena;
31:             unsigned short suLongitud;
32:
33:     };
34:
35:     ostream& operator<<(ostream& elFlujo,Cadena& laCadena)
36:     {
37:         elFlujo << laCadena.ObtenerCadena();
38:         return elFlujo;
39:     }
40:
41:     istream& operator>>(istream& elFlujo,Cadena& laCadena)
42:     {
43:         elFlujo >> laCadena.ObtenerCadena();
44:         return elFlujo;
45:     }
46:
47:     int main()
48:     {
49:         Cadena laCadena("Hola, mundo.");
50:         cout << laCadena;
51:         return 0;
52:     }
```

##### 5. CAZA ERRORES: ¿Qué está mal en este programa?

```
1: #include <iostream.h>
2:
3: class Animal;
4:
5: void asignarValor(Animal & , int);
6:
7:
8: class Animal
9: {
10: public:
11:     int ObtenerPeso()const { return suPeso; }
12:     int ObtenerEdad() const { return suEdad; }
13: private:
14:     int suPeso;
15:     int suEdad;
16: };
17:
```

```

18:     void asignarValor(Animal & elAnimal, int elPeso)
19:     {
20:         friend class Animal;
21:         elAnimal.suPeso = elPeso;
22:     }
23:
24:     int main()
25:     {
26:         Animal peppy;
27:         asignarValor(peppy,5);
28:     }

```

No se puede colocar la declaración `friend` dentro de la función. Debe declarar la función como amiga en la clase.

6. Corrija el listado del ejercicio 5 para que pueda compilarse.

```

1:     #include <iostream.h>
2:
3:     class Animal;
4:
5:     void asignarValor(Animal& , int);
6:
7:
8:     class Animal
9:     {
10:    public:
11:        friend void asignarValor(Animal&, int);
12:        int ObtenerPeso()const { return suPeso; }
13:        int ObtenerEdad() const { return suEdad; }
14:    private:
15:        int suPeso;
16:        int suEdad;
17:    };
18:
19:    void asignarValor(Animal& elAnimal, int elPeso)
20:    {
21:        elAnimal.suPeso = elPeso;
22:    }
23:
24:    int main()
25:    {
26:        Animal peppy;
27:        asignarValor(peppy,5);
28:        return 0;
29:    }

```

7. **CAZA ERRORES:** ¿Qué está mal en este código?

```

1:     #include <iostream.h>
2:
3:     class Animal;

```

```
4:
5:     void asignarValor(Animal& , int);
6:     void asignarValor(Animal& , int, int);
7:
8:     class Animal
9:     {
10:         friend void asignarValor(Animal & , int); // i aquí está el cambio!
11:     private:
12:         int suPeso;
13:         int suEdad;
14:     };
15:
16:     void asignarValor(Animal& elAnimal, int elPeso)
17:     {
18:         elAnimal.suPeso = elPeso;
19:     }
20:
21:
22:     void asignarValor(Animal& elAnimal, int elPeso, int laEdad)
23:     {
24:         elAnimal.suPeso = elPeso;
25:         elAnimal.suEdad = laEdad;
26:     }
27:
28:     int main()
29:     {
30:         Animal peppy;
31:         asignarValor(peppy,5);
32:         asignarValor(peppy,7,9);
33:     }
```

La función `asignarValor(Animal & , int)` fue declarada como amiga, pero la función sobrecargada `asignarValor(Animal & , int, int)` no se declaró como amiga.

8. Corrija el ejercicio 7 para que se pueda compilar.

```
1:     #include <iostream.h>
2:
3:     class Animal;
4:
5:     void asignarValor(Animal& , int);
6:     void asignarValor(Animal& ,int,int); // i aquí está el cambio!
7:
8:     class Animal
9:     {
10:         friend void asignarValor(Animal& ,int);
11:         friend void asignarValor(Animal& ,int,int);
12:     private:
13:         int suPeso;
14:         int suEdad;
```

```
15:     };
16:
17:     void asignarValor(Animal& elAnimal, int elPeso)
18:     {
19:         elAnimal.suPeso = elPeso;
20:     }
21:
22:
23:     void asignarValor(Animal& elAnimal, int elPeso, int laEdad)
24:     {
25:         elAnimal.suPeso = elPeso;
26:         elAnimal.suEdad = laEdad;
27:     }
28:
29:     int main()
30:     {
31:         Animal peppy;
32:         asignarValor(peppy,5);
33:         asignarValor(peppy,7,9);
34:         return 0;
35:     }
```



## Día 16

### Cuestionario

1. ¿Qué es el operador de inserción, y qué hace?

El operador de inserción (`<<`) es un operador miembro del objeto `ostream` y se utiliza para escribir en el dispositivo de salida.

2. ¿Qué es el operador de extracción, y qué hace?

El operador de extracción (`>>`) es un operador miembro del objeto `istream` y se utiliza para escribir en las variables de su programa.

3. ¿Cuáles son las tres formas de utilizar `cin.get()`, y cuáles son sus diferencias?

La primera forma de `cin.get()` es sin parámetros. Ésta regresa el valor del carácter encontrado, y regresará `EOF` (fin de archivo) si se llega al fin del archivo.

La segunda forma de `cin.get()` toma una referencia a un carácter como su parámetro; ese carácter se llena con el siguiente carácter en el flujo de entrada. El valor de retorno es un objeto `iostream`.

La última forma de `cin.get()` toma tres parámetros. El primer parámetro es un apuntador a un arreglo de caracteres, el segundo parámetro es el número máximo de caracteres a leer más uno, y el tercer parámetro es el carácter de terminación. El valor de retorno es el objeto `iostream`.

4. ¿Cuál es la diferencia entre `cin.read()` y `cin.getline()`?  
`cin.read()` se utiliza para leer estructuras de datos binarios.  
`cin.getline()` se utiliza para leer del búfer de `iostream`.
5. ¿Cuál es el ancho predeterminado para enviar como salida un entero largo mediante el operador de inserción?  
Lo suficientemente ancho para mostrar el número completo.
6. ¿Cuál es el valor de retorno del operador de inserción?  
Una referencia a un objeto `iostream`.
7. ¿Qué parámetro lleva el constructor para un objeto `ofstream`?  
El nombre del archivo que se va a abrir.
8. ¿Qué hace el argumento `ios::ate`?  
`ios::ate` lo lleva al final del archivo, pero puede escribir datos en cualquier parte del archivo.

## Ejercicios

1. Escriba un programa que escriba en los cuatro objetos `iostream` estándar: `cin`, `cout`, `cerr` y `clog`.

```
1: #include <iostream.h>
2: int main()
3: {
4:     int x;
5:     cout << "Escriba un número: ";
6:     cin >> x;
7:     cout << "Usted escribió: " << x << endl;
8:     cerr << "Oh oh, iesto va a cerr!" << endl;
9:     clog << "Oh oh, iesto va a clog!" << endl;
10:    return 0;
11: }
```

2. Escriba un programa que pida al usuario que escriba su nombre completo y luego lo despliegue en pantalla.

```
1: #include <iostream.h>
2: int main()
3: {
4:     char nombre[80];
5:     cout << "Escriba su nombre completo: ";
6:     cin.getline(nombre,80);
7:     cout << "\nUsted escribió: " << nombre << endl;
8:     return 0;
9: }
```

3. Modifique el listado 16.9 para que haga lo mismo, pero sin utilizar `putback()` ni `ignore()`.

```

1:      // Listado
2:      #include <iostream.h>
3:
4:      int main()
5:      {
6:          char ch;
7:          cout << "escriba una frase: ";
8:          while (cin.get(ch))
9:          {
10:              switch (ch)
11:              {
12:                  case '!':
13:                      cout << '$';
14:                      break;
15:                  case '#':
16:                      break;
17:                  default:
18:                      cout << ch;
19:                      break;
20:              }
21:          }
22:          return 0;
23:      }

```

4. Escriba un programa que tome un nombre de archivo como parámetro y abra el archivo para lectura. Lea todos los caracteres del archivo y despliegue en la pantalla sólo las letras y los signos de puntuación. (Ignore todos los caracteres no imprimibles.) Luego el programa deberá cerrar el archivo y terminar.

```

1:      #include <fstream.h>
2:      enum BOOL { FALSE, TRUE };
3:
4:      int main(int argc, char**argv) // regresa 1 en caso de error
5:      {
6:
7:          if (argc != 2)
8:          {
9:              cout << "Uso: argv[0] <infile>\n";
10:             return(1);
11:         }
12:
13:         // abrir el flujo de entrada
14:         ifstream fin (argv[1],ios::binary);
15:         if (!fin)
16:         {
17:             cout << "No se pudo abrir " << argv[1] << " para lectura.\n";
18:             return(1);
19:         }

```

```

20:
21:     char ch;
22:     while (fin.get(ch))
23:         if ((ch > 32 && ch < 127) || ch == '\n' || ch == '\t')
24:             cout << ch;
25:         fin.close();
26:     }

```

5. Escriba un programa que despliegue sus argumentos de la línea de comandos en orden inverso, y que no despliegue el nombre del programa.

```

1: #include <iostream.h>
2:
3: int main(int argc, char**argv) // regresa 1 en caso de error
4: {
5:     for (int ctr = argc-1; ctr ; ctr--)
6:         cout << argv[ctr] << " ";
7:     return 0;
8: }

```

## Día 17

### Cuestionario

1. ¿Puedo utilizar nombres definidos en un espacio de nombres sin utilizar la palabra reservada `using`?

Sí, puede utilizar nombres definidos en un espacio de nombres si les antepone el identificador del espacio de nombres.

2. ¿Cuáles son las principales diferencias entre espacios de nombres normales y espacios de nombres sin nombre?

Los compiladores agregan una directiva `using` implícita a un espacio de nombres sin nombre. Por lo tanto, los nombres de un espacio de nombres sin nombre pueden utilizarse sin necesidad de un identificador del espacio de nombres. Los espacios de nombres normales no tienen directivas `using` implícitas. Para usar nombres en espacios de nombres normales, debe aplicar ya sea las directivas `using` o las declaraciones `using`, o identificar los nombres con el identificador del espacio de nombres.

Los nombres en un espacio de nombres normal pueden utilizarse fuera de la unidad de traducción en la que está declarado el espacio de nombres. Los nombres en un espacio de nombres sin nombre pueden utilizarse sólo dentro de la unidad de traducción en la que está declarado el espacio de nombres.

3. ¿Cuál es el espacio de nombres estándar?

El espacio de nombres `std` está definido por la Biblioteca estándar de C++. Incluye declaraciones de todos los nombres que se encuentran en la Biblioteca estándar.

## Ejercicios

1. CAZA ERRORES: ¿Qué está mal en este programa?

```
#include <iostream>

int main()
{
    cout << "¡Hola, mundo!" << endl;
    return 0;
}
```

El archivo de encabezado `iostream` estándar de C++ declara a `cout` y a `endl` en el espacio de nombres `std`. No pueden utilizarse fuera del espacio de nombres `std` sin un identificador para el espacio de nombres.

2. Mencione tres formas de solucionar el problema del ejercicio 1.

1. `using namespace std;`
2. `using std::cout;`  
`using std::endl;`
3. `std::cout << "¡Hola, mundo!" << std::endl;`

D

## Día 18

### Cuestionario

1. ¿Cuál es la diferencia entre programación orientada a objetos y programación procedural?

La programación procedural se enfoca en separar las funciones de los datos. La programación orientada a objetos une los datos y la funcionalidad dentro de los objetos y se enfoca en la interacción que debe existir entre distintos objetos.

2. ¿Cuáles son las fases del análisis y del diseño orientados a objetos?

La fase del análisis determina las necesidades de la organización y se enfoca en comprender el dominio del problema. Es aquí donde se modelan las clases. La etapa del diseño se enfoca en crear las soluciones. En términos generales, el diseño es la transformación del concepto del problema en un modelo que pueda implementarse en software.

3. ¿Cómo se relacionan los diagramas de secuencia y los diagramas de colaboración?

Los diagramas de secuencia establecen la secuencia de cada objeto a través del tiempo, mientras que los diagramas de colaboración establecen la interacción entre las distintas clases. Se puede generar un diagrama de colaboración directamente de un diagrama de secuencia.

## Ejercicios

1. Suponga que tiene que simular la intersección de la avenida Massachusetts con la calle Vassar (dos caminos típicos de dos carriles, con semáforos y cruce de peatones). El propósito de la simulación es determinar si la sincronización del semáforo permite un flujo continuo de tráfico.

¿Qué tipos de objetos debe modelar en la simulación? ¿Cuáles serían las clases para la simulación?

Autos, motocicletas, camionetas, bicicletas, vehículos de emergencias y peatones utilizan la intersección. Además, existen señales de tráfico para vehículos y peatones.

¿Se debería incluir el estado del asfalto en la simulación?

Efectivamente, la calidad del asfalto tiene efectos importantes en el tráfico, pero para un primer diseño, será más sencillo si dejamos a un lado este aspecto.

El primer objeto es la intersección misma. Quizá el objeto intersección mantenga una lista de autos que esperan las señales de tránsito en cada dirección, así como una lista de peatones que esperan cruzar las avenidas; se necesitarán métodos para elegir cuáles y cuántos autos y peatones cruzarán la intersección.

Existirá sólo una intersección, así que considere como asegurar que sólo un objeto sea instanciado (sugerencia: piense en métodos estáticos y acceso protegido).

Los vehículos y peatones son clientes de la intersección y comparten ciertas características: pueden aparecer en cualquier momento, en cualquier cantidad y se detienen en los semáforos a esperar el paso (aunque en diferente situación). Esto sugiere una clase base común para peatones y vehículos.

Las clases pueden ser:

```
class Entidad; // un cliente de la intersección

// La base para todos los autos, bicicletas y vehículos de emergencia.
class Vehiculo : Entidad ...;

// La base para los peatones
class Peaton : Entidad...;

class Auto : public Vehiculo...;
class Camioneta : public Vehiculo...;
class Motocicleta : public Vehiculo...;
class Bicicleta : public Vehiculo...;
class VehiculoDeEmergencia : public Vehiculo...;
```

```
// Contiene las listas de autos y peatones que esperan el paso
class Interseccion;
```

2. Suponga que la intersección del ejercicio 1 está en un suburbio de Boston, que sin duda tiene las calles menos amigables de todo Estados Unidos. A cualquier hora hay tres tipos de conductores en Boston:

Los locales, quienes siguen conduciendo por las intersecciones aunque el semáforo esté en rojo; los turistas, que manejan lenta y cautelosamente (por lo general, en un auto rentado); y los taxistas, que tienen una amplia variedad de patrones de manejo, dependiendo de los tipos de pasajeros que lleven.

Además, Boston tiene dos tipos de peatones: los locales, que cruzan la calle cuando les da la gana y muy raras veces utilizan las áreas para cruce de peatones; y los turistas, quienes siempre utilizan las áreas para cruce de peatones y cruzan sólo cuando el semáforo lo permite.

Finalmente, Boston tiene ciclistas que nunca ponen atención a las señales de alto. ¿Cómo cambian el modelo estas consideraciones?

Un inicio razonable para ello será crear objetos derivados que modelen el refinamiento deseado por el problema:

```
class AutoLocal : public Auto...
    class AutoTurista : public Auto...
    class Taxi : public Auto...
    class PeatonLocal : public
Peaton...
    class PeatonTurista : public
Peaton...
    class BicicletaLocal : public Bicicleta...;
```

Mediante el uso de métodos virtuales, cada clase puede modificar el comportamiento genérico para alcanzar sus propias especificaciones. Por ejemplo, los conductores de Boston reaccionan de manera diferente a los turistas cuando ven una luz roja, aún así, heredarán el comportamiento genérico que se pueda aplicar.

3. Diseñe un programador de grupos. Este software le permite programar juntas entre individuos o grupos y reservar un número limitado de salones para conferencias. Identifique los subsistemas principales.

Se necesitarán dos programas discretos para este proyecto: el cliente, que ejecutarán los usuarios, y el servidor, que se podría ejecutar en una máquina remota. Además, la máquina cliente podrá tener un componente administrativo que permita al administrador agregar nuevos usuarios y salones.

Si decide implementar esta aplicación como un modelo cliente/servidor, el cliente aceptará información de los usuarios y generará peticiones al servidor. El servidor deberá atender las peticiones y regresar los resultados hacia el cliente. Con este modelo, mucha gente podrá programar encuentros y conferencias al mismo tiempo.

En el lado del cliente existirán dos subsistemas principales además del módulo administrativo: la interfaz de usuario y el sistema de comunicaciones. El servidor consistirá en tres subsistemas principales: comunicaciones, programación de encuentros y conferencias y una interfaz para correo que puede anunciar a los usuarios cuando hayan ocurrido cambios (de conferencias).

4. Diseñe y muestre las interfaces para las clases del módulo de reservación de salones del programa que se describe en el ejercicio 3.

Un encuentro se define como un grupo de gente que ha reservado un salón durante un lapso de tiempo. La persona que realiza la reservación podría querer un salón o un lapso específicos; pero el programador de encuentros y conferencias siempre debe indicar cuánto durará el último encuentro y quiénes deben asistir.

Probablemente los objetos incluirán a los usuarios del sistema así como los salones de conferencias. No olvide incluir una clase para el calendario y quizá una clase *Conferencia* que encapsule todo cuanto sabe acerca de un evento en particular.

Los prototipos para las clases podrían incluir:

```
class Calendario;           // referencia a una clase posterior
class Conferencia;          // referencia a una clase posterior
class Configuracion
{
public:
    Configuracion();
    ~Configuracion();
    Conferencia Programa(ListaDePersonas &, Delta Time
duracion);
    Conferencia Programa(ListaDePersonas &, Delta Time
duracion, Tiempo);
    Conferencia Programa(ListaDePersonas &, Delta Time
duracion, Salon);
    ListaDePersonas & Personas();      // métodos de acceso
públicos
    ListaDePersonas & Salones();      // métodos de acceso públicos
protected:
    ListaDeSalones    salones;
    ListaDePersonas   personas;
};
typedef long ID_Salon;
class Salon
{
public:
```

```
    Salon(Cadena nombre, ID_Salon id, int capacidad,
Cadena direcciones = "", Cadena descripcion = "");
    -Salon();
    ClaseCalendario Calendario();

protected:
    ClaseCalendario    calendario;
    int                capacidad;
    ID_Salon          id;
    Cadena            nombre;
    Cadena            direcciones;           // ¿dónde está el salón?
    Cadena            descripcion;

};

typedef long ID_Persona;
class Persona
{
public:
    Persona(Cadena nombre, ID_Persona id);
    -Persona();
    Clase Calendario Calendario();           // punto de acceso para
  // agregar conferencias

protected:
    ClaseCalendario    calendario;
    ID_Persona         id;
    Cadena            nombre;
};

class ClaseCalendario
{
public:
    ClaseCalendario();
    -ClaseCalendario();

    void Agregar(const Conferencia&);        // agregar una
conferencia al calendario
    void Eliminar(const Conferencia&);
    Conferencia * Busqueda(Hora);              // revisa si existe una
  // conferencia
  // a la hora especificada

    Bloquear(Hora, Duracion, Cadena razon = "");
// reservar tiempo para usted...

protected:
    ListaOrdenadaDeConferencias conferencias;
};

class Conferencia
{
public:
    Conferencia(ListaDePersonas &, Salon salon,
```

```
    Hora cuando, Duracion duracion, Cadena proposito  
= "");  
-Conferencia();  
protected:  
    ListaDePersonas personas;  
    Salon salon;  
    Hora cuando;  
    Duracion     duracion;  
    Cadena      proposito;  
};
```

## Día 19

### Cuestionario

1. ¿Cuál es la diferencia entre una plantilla y una macro?

Las plantillas están incluidas en C++ y tienen seguridad de tipos. Las macros son implementadas por el preprocesador y no tienen seguridad de tipos.

2. ¿Cuál es la diferencia entre el parámetro de una plantilla y el parámetro de una función?

El parámetro de una plantilla crea una instancia de la plantilla para cada tipo. Si usted crea seis instancias de una plantilla, se crearán seis clases o funciones diferentes. Los parámetros de una función cambian el comportamiento o datos de la función, pero sólo una función es creada.

3. ¿Cuál es la diferencia entre una clase amiga de plantilla de tipo específico y una clase amiga de plantilla general?

La clase amiga de plantilla general crea una función por cada tipo de la clase parametrizada; la clase amiga de plantilla de tipo específico crea una instancia de tipo específico por cada instancia de la clase parametrizada.

4. ¿Es posible proporcionar un comportamiento especial para una instancia de una plantilla, pero no para otras instancias?

Sí, cree una función especializada para la instancia particular. Además de crear `Arreglo<t>::UnaFuncion()`, también cree `Arreglo<int>::UnaFuncion()` para cambiar el comportamiento de un arreglo de enteros.

5. ¿Cuántas variables estáticas se crean si se coloca un miembro estático en la definición de una clase de plantilla?

Una por cada instancia de la clase.

6. ¿Qué son los iteradores de la biblioteca estándar de C++?

Los iteradores son apuntadores generalizados. Un iterador puede incrementarse para apuntar al siguiente nodo de una secuencia. También puede ser desreferenciado para obtener el nodo al que apunta.

7. ¿Qué es un objeto de función?

Un objeto de función es una instancia de una clase donde se define la sobrecarga del operador (). Puede ser usado como una función normal.

## Ejercicios

1. Cree una plantilla basada en esta clase Lista:

```
class Lista
{
private:
public:
    Lista():cabeza(0),cola(0),laCuenta(0) {}
    virtual ~Lista();
    void insertar(int valor);
    void agregar(int valor);
    int esta_presente(int valor) const;
    int esta_vacia() const { return cabeza == 0; }
    int cuenta() const { return laCuenta; }
private:
    class CeldaLista
    {
    public:
        CeldaLista(int valor, CeldaLista *celda =):val(valor),
        &siguiente(cel){}
        int val;
        CeldaLista *siguiente;
    };
    CeldaLista *cabeza;
    CeldaLista *cola;
    int laCuenta;
};
```

Aquí hay una forma de implementar la plantilla:

```
template <class Type>
class Lista
{
public:
    Lista():cabeza(0),cola(0),laCuenta(0) {}
    virtual ~Lista();
    void insertar(Tipo valor);
    void agregar(Tipo valor);
```

D

```
        int esta_presente(Tipo valor) const;
        int esta_vacia() const { return cabeza == 0; }
        int cuenta() const { return laCuenta; }

private:
    class CeldaLista
    {
public:
    CeldaLista(Tipo valor, CeldaLista *Celda = 0):val(valor),
    ->siguiente(cel){}
        Tipo val;
        CeldaLista *siguiente;
    };

    CeldaLista *cabeza;
    CeldaLista *cola;
    int laCuenta;
};

2. Escriba la implementación para la versión (que no sea de plantilla) de la clase
Lista .
void List::insert(int value)
{
    ListCell *pt = new ListCell(value, head);
    assert (pt != 0);

    // esta linea se agrega para manejar la cola
    if (head == 0) tail = pt;

    head = pt;
    theCount++;
}

void List::append(int value)
{
    ListCell *pt = new ListCell(value);
    if (head == 0)
        head = pt;
    else
        tail->next = pt;

    tail = pt;
    theCount++;
}

int List::is_present(int value) const
{
    if (head == 0) return 0;
    if (head->val == value || tail->val == value)
        return 1;
```

```
    ListCell *pt = head->next;
    for (; pt != tail; pt = pt->next)
        if (pt->val == value)
            return 1;

    return 0;
}
```

3. Escriba la versión de plantilla de las implementaciones.

```
template <class Type>
List<Type>::List()
{
    ListCell *pt = head;

    while (pt)
    {
        ListCell *tmp = pt;
        pt = pt->next;
        delete tmp;
    }
    head = tail = 0;
}

template <class Type>
void List<Type>::insert(Type value)
{
    ListCell *pt = new ListCell(value, head);
    assert (pt != 0);

    // esta linea se agrega para manejar la cola
    if (head == 0) tail = pt;

    head = pt;
    theCount++;
}

template <class Type>
void List<Type>::append(Type value)
{
    ListCell *pt = new ListCell(value);
    if (head == 0)
        head = pt;
    else
        tail->next = pt;

    tail = pt;
    theCount++;
}

template <class Type>
```

D

```

int List<Type>::is_present(Type value) const
{
    if (head == 0) return 0;
    if (head->val == value || tail->val == value)
        return 1;

    ListCell *pt = head->next;
    for (; pt != tail; pt = pt->next)
        if (pt->val == value)
            return 1;

    return 0;
}

```

4. Declare tres objetos de tipo lista: una lista de Cadena, una lista de objetos Gato y una lista de valores de tipo int.

```

Lista<Cadena> listaDeCadenas;
Lista<Gato> listaDeGatos;
Lista<int> listaDeEnteros;

```

5. **CAZA ERRORES:** ¿Qué está mal en el siguiente código? (Suponga que la plantilla Lista está definida y que Gato es la clase que se definió anteriormente en el libro.)

```

Lista<Gato> Lista_Gato;
Gato Felix;
ListaGato.agregar(Felix);
cout << "Felix " <<
    (Lista_Gato.esta_presente(Felix)) ? "" : "no " << " está presente\n";

```

**PISTA** (esto está difícil): ¿Qué hace a Gato diferente de int?

Gato no tiene un operator== definido; todas las operaciones que comparan los valores de los elementos de Lista, como esta\_presente, resultarán en errores de compilación. Para reducir la probabilidad de esto, ponga comentarios antes de la definición de la plantilla que establezcan las operaciones a definir para todas las clases que se utilicen en la plantilla; esto le permitirá compilar de manera segura.

6. Declare el operator== amigo para Lista.

```
friend int operator==(const Tipo& lhs, const Tipo& rhs);
```

7. Implemente el operator== amigo para Lista.

```

template <class Type>
int List<Type>::operator==(const Type& lhs, const Type& rhs)
{
    // primero compara la longitud
    if (lhs.theCount != rhs.theCount)
        return 0;      // longitudes diferentes

    ListCell *lh = lhs.head;
    ListCell *rh = rhs.head;

    for(; lh != 0; lh = lh.next, rh = rh.next)

```

```
    if (lh.value != rh.value)
        return 0;

    return 1;           // si no son diferentes, entonces son iguales
}
```

8. ¿Tiene operator== el mismo problema que en el ejercicio 5?

Sí, porque la comparación de los arreglos implica la comparación de los elementos; el operator!= también debe estar definido para estos elementos.

9. Implemente una función de plantilla para intercambiar dos variables.

```
// plantilla swap:
// debe tener asignación y constructor de copia
// definidos para Type.
template <class Type>
void swap(Type& lhs, Type& rhs)
{
    Type temp(lhs);
    lhs = rhs;
    rhs = temp;
}
```

10. Implemente a ClaseEscuela del listado 19.8 como una lista. Utilice la función push\_back() para agregar cuatro estudiantes a la lista. Luego desplácese por la lista resultante e incremente en uno la edad de cada estudiante.

```
#include <list>

template<class T, class A>
void ShowList(const list<T, A>& aList); // muestra los elementos de la lista

typedef list<Student> SchoolClass;

int main()
{
    Student Harry("Harry", 18);
    Student Sally("Sally", 15);
    Student Bill("Bill", 17);
    Student Peter("Peter", 16);

    SchoolClass GrowingClass;
    GrowingClass.push_back(Harry);
    GrowingClass.push_back(Sally);
    GrowingClass.push_back(Bill);
    GrowingClass.push_back(Peter);
    ShowList(GrowingClass);

    cout << "One year later:\n";

    for (SchoolClass::iterator i = GrowingClass.begin();
```



```
        i != GrowingClass.end(); ++i)
        i->AsignarEdad(i->ObtenerEdad() + 1);

    ShowList(GrowingClass);

    return 0;
}

// Muestra los elementos de la lista
//
template<class T, class A>
void ShowList(const list<T, A>& aList)
{
    for (list<T, A>::const_iterator ci = aList.begin();
          ci != aList.end(); ++ci)
        cout << *ci << "\n";

    cout << endl;
}
```

11. Modifique el ejercicio 10 para utilizar un objeto de función para desplegar el registro de cada estudiante.

```
#include <algorithm>

template<class T>
class Print
{
public:
    void operator()(const T& t)
    {
        cout << t << "\n";
    }
};

template<class T, class A>
void ShowList(const list<T, A>& aList)
{
    Print<Student>      PrintStudent;

    for_each(aList.begin(), aList.end(), PrintStudent);

    cout << endl;
}
```

## Día 20

### Cuestionario

1. ¿Qué es una excepción?

Una excepción es un objeto que se crea al invocar la palabra reservada `throw`. Se utiliza para señalar una condición de excepción y se pasa al primer bloque `catch` de la pila que maneje esa excepción.

2. ¿Qué es un bloque `try`?

Es un conjunto de instrucciones que pueden generar una excepción.

3. ¿Qué es una instrucción `catch`?

Una instrucción `catch` tiene la firma del tipo de excepción que maneja. Se utiliza después de un bloque `try` y actúa como el receptor de excepciones generadas dentro del bloque `try`.

4. ¿Qué información puede contener una excepción?

Una excepción es un objeto y puede contener cualquier información establecida dentro de las clases creadas por el usuario.

5. ¿Cuándo se crean los objetos de excepción?

Los objetos de excepción se crean cuando se invoca la palabra reservada `throw`.

6. ¿Se deben pasar las excepciones por valor o por referencia?

En general, las excepciones deben ser pasadas por referencia. Si no pretende modificar el contenido de una excepción, deberá pasar la excepción como una referencia constante.

7. ¿Atrapará una instrucción `catch` una excepción derivada si está buscando la clase base?

Sí, siempre que pase la excepción por referencia.

8. Si se utilizan dos instrucciones `catch`, una para la clase base y una para la derivada, ¿cuál debe ir primero?

Las instrucciones `catch` se examinan en el orden en que aparecen en el código fuente. La primer instrucción `catch` que coincide con la firma de la excepción será utilizada.

9. ¿Qué significa `catch(...)`?

`catch(...)` atrapará cualquier excepción de cualquier tipo.

10. ¿Qué es un punto de interrupción?

Un punto de interrupción es el lugar del código donde el depurador detendrá la ejecución sin terminar el programa. En un punto de interrupción, puede verificar el estado de las variables y modificar sus valores si es necesario.

D

## Ejercicios

1. Cree un bloque try, una instrucción catch, y una excepción simple.

```
#include <iostream.h>
class OutOfMemory {};
int main()
{
    try
    {
        int *myInt = new int;
        if (myInt == 0)
            throw OutOfMemory();
    }
    catch (OutOfMemory)
    {
        cout << "Unable to allocate memory!\n";
    }
    return 0;
}
```

2. Modifique la respuesta del ejercicio 1, coloque datos en la excepción junto con una función de acceso, y utilícela en el bloque catch.

```
#include <iostream.h>
#include <stdio.h>
#include <string.h>
class OutOfMemory
{
public:
    OutOfMemory(char *);
    char* ObtenerCadena() { return suCadena; }
private:
    char* suCadena;
};

OutOfMemory::OutOfMemory(char * theType)
{
    suCadena = new char[80];
    char warning[] = "Out Of Memory! Can't allocate room for: ";
    strncpy(suCadena,warning,60);
    strncat(suCadena,theType,19);
}

int main()
{
    try
    {
        int *myInt = new int;
        if (myInt == 0)
```

```
        throw OutOfMemory("int");
    }
    catch (OutOfMemory& theException)
    {
        cout << theException.ObtenerCadena();
    }
    return 0;
}
```

3. Modifique la clase del ejercicio 2 para que sea una jerarquía de excepciones. Cambie el bloque `catch` para utilizar los objetos derivados y los objetos base.

```
1:      #include <iostream.h>
2:
3:      // el tipo de datos Exception es una clase abstracta
4:      class Exception
5:      {
6:          public:
7:              Exception(){}
8:              virtual ~Exception(){}
9:              virtual void PrintError() = 0;
10:         };
11:
12:         // Clase derivada para manejar problemas de memoria.
13:         // Vea que no se reserva memoria en esta clase.
14:         class OutOfMemory : public Exception
15:         {
16:             public:
17:                 OutOfMemory(){}
18:                 ~OutOfMemory(){}
19:                 virtual void PrintError();
20:         private:
21:         };
22:
23:         void OutOfMemory::PrintError()
24:         {
25:             cout << "Out of Memory!!\n";
26:         }
27:
28:         // Clase derivada para manejar números erróneos
29:         class RangeError : public Exception
30:         {
31:             public:
32:                 RangeError(unsigned long number){badNumber = number;}
33:                 ~RangeError(){}
34:                 virtual void PrintError();
35:                 virtual unsigned long GetNumber() { return badNumber; }
36:                 virtual void SetNumber(unsigned long number) {badNumber =
37:                                     ?number;}
37:         private:
38:             unsigned long badNumber;
```

```

39:     };
40:
41: void RangeError::PrintError()
42: {
43:     cout << "Number out of range. You used " << GetNumber() <<
44:         "?!!\n";
45: }
46: void MyFunction(); // prototipo de función
47:
48: int main()
49: {
50:     try
51:     {
52:         MyFunction();
53:     }
54:     // Sólo se requiere una instrucción catch, utilice funciones
55:     // virtuales para llamar a la función correcta.
56:     catch (Exception& theException)
57:     {
58:         theException.PrintError();
59:     }
60:     return 0;
61: }
62:
63: void MyFunction()
64: {
65:     unsigned int *myInt = new unsigned int;
66:     long testNumber;
67:     if (myInt == 0)
68:         throw OutOfMemory();
69:     cout << "Enter an int: ";
70:     cin >> testNumber;
71:     // esta prueba se debe reemplazar por una serie de pruebas
72:     // que indiquen una mala entrada
73:     if (testNumber > 3768 || testNumber < 0)
74:         throw RangeError(testNumber);
75:
76:     *myInt = testNumber;
77:     cout << "OK. myInt: " << *myInt;
78:     delete myInt;
79: }

```

4. Modifique el programa del ejercicio 3 para que tenga tres niveles de llamadas a funciones.

```

1: #include <iostream.h>
2:
3: // El tipo de datos Exception es una clase abstracta
4: class Exception
5: {
6: public:
7:     Exception(){}

```

```
8:         virtual ~Exception(){}
9:         virtual void PrintError() = 0;
10:    };
11:
12:    // Clase derivada para manejar problemas de memoria.
13:    // Vea que no se reserva memoria en esta clase.
14:    class OutOfMemory : public Exception
15:    {
16:        public:
17:            OutOfMemory(){}
18:            ~OutOfMemory(){}
19:            virtual void PrintError();
20:    private:
21:    };
22:
23:    void OutOfMemory::PrintError()
24:    {
25:        cout << "Out of Memory!!\n";
26:    }
27:
28:    // Clase derivada para manejar números erróneos
29:    class RangeError : public Exception
30:    {
31:        public:
32:            RangeError(unsigned long number){badNumber = number;}
33:            ~RangeError(){}
34:            virtual void PrintError();
35:            virtual unsigned long GetNumber() { return badNumber; }
36:            virtual void SetNumber(unsigned long number) {badNumber =
37:                number;}
37:    private:
38:        unsigned long badNumber;
39:    };
40:
41:    void RangeError::PrintError()
42:    {
43:        cout << "Number out of range. You used " << GetNumber() <<
44:            "?!!\n";
45:
46:    // prototipos de función
47:    void MyFunction();
48:    unsigned int * FunctionTwo();
49:    void FunctionThree(unsigned int *);
50:
51:    int main()
52:    {
53:        try
54:        {
55:            MyFunction();
56:        }
57:        // Sólo se requiere una instrucción catch, utilice funciones
58:        // virtuales para llamar a la función correcta.
```

D

```
59:         catch (Exception& theException)
60:     {
61:         theException.PrintError();
62:     }
63:     return 0;
64: }
65:
66:     unsigned int * FunctionTwo()
67: {
68:     unsigned int *myInt = new unsigned int;
69:     if (myInt == 0)
70:         throw OutOfMemory();
71:     return myInt;
72: }
73:
74: void MyFunction()
75: {
76:     unsigned int *myInt = FunctionTwo();
77:
78:     FunctionThree(myInt);
79:     cout << "Ok. myInt: " << *myInt;
80:     delete myInt;
81: }
82:
83: void FunctionThree(unsigned int *ptr)
84: {
85:     long testNumber;
86:     cout << "Enter an int: ";
87:     cin >> testNumber;
88:     // esta prueba se debe reemplazar por una serie de pruebas
89:     // que indiquen una mala entrada.
90:     if (testNumber > 3768 || testNumber < 0)
91:         throw RangeError(testNumber);
92:     *ptr = testNumber;
93: }
```

5. **CAZA ERRORES:** ¿Qué está mal en el siguiente código?

```
class xNoHayMemoria
{
public:
    xNoHayMemoria()
    {
        elMsg = new char[ 20 ];
        strcpy(elMsje, "error en memoria");
    }
    ~xNoHayMemoria()
    {
        delete [] elMsje;
        cout << "Memoria restablecida." << endl;
    }
    char * Mensaje()
    {
        return elMsje;
    }
```

```

private:
    char * elMsje;
};

main()
{
    try
    {
        char * var = new char;
        if (var == 0)
        {
            xNoHayMemoria * apx = throw apx;
        }
    }
    catch(xNoHayMemoria * laExcepcion)
    {
        cout << laExcepcion->Mensaje() << endl;
        delete laExcepcion;
    }
    return 0;
}

```

**D****Ejercicios**

Durante el manejo de la condición `xNoHayMemoria`, el constructor de `xNoHayMemoria` crea un objeto `string`. Esta excepción se generará sólo cuando el sistema se quede sin memoria; de esta forma la creación del objeto `string` fallará.

Es posible que al intentar crear la cadena se genere la misma excepción, lo que provocará un ciclo infinito hasta que el sistema se congele. Si realmente se requiere esta cadena, puede reservar el espacio en un búfer estático antes de iniciar la ejecución del programa, y utilizarlo cuando se genere la excepción.

## Día 21

### Cuestionario

1. ¿Qué es un guardia de inclusión?

Los guardias de inclusión se utilizan para evitar que un archivo de encabezado se incluya más de una vez dentro de un programa.

2. ¿Cómo le indica al compilador que imprima el contenido del archivo intermedio, para que muestre los efectos del preprocesador?

Para el compilador de GNU, se debe utilizar la opción `-E`. Si utiliza un compilador diferente, la respuesta a esta pregunta debe ser descubierta por usted mismo (revise la documentación de su compilador).

3. ¿Cuál es la diferencia entre `#define depurar 0` y `#undef depurar`?

`#define depurar 0` establece que el término `depurar` será igual a 0 (cero). En cualquier lugar donde aparezca la palabra `depurar` será sustituida por el carácter cero. `#undef depurar` elimina cualquier definición de `depurar`; cuando se encuentre esta palabra en el archivo, se dejará intacta.

4. ¿Qué hace el operador de complemento (~) a nivel de bits?

Invierte el valor de cada bit contenido en un número.

5. ¿Cuál es la diferencia entre OR y OR exclusivo (xor)?

OR regresa `true` si alguno o ambos bits están encendidos; OR exclusivo regresa `true` sólo si alguno de los bits está encendido, pero regresa `false` si ambos lo están.

6. ¿Cuál es la diferencia entre & y &&?

`&` es el operador AND a nivel de bits, mientras que `&&` es el operador AND lógico.

7. ¿Cuál es la diferencia entre | y ||?

`|` es el operador OR a nivel de bits, mientras que `||` es el operador OR lógico.

## Ejercicios

1. Escriba las instrucciones de guardias de inclusión para el archivo de encabezado `STRING.H`.

```
#ifndef STRING_H
#define STRING_H
...
#endif
```

2. Escriba una macro `ASSERT()` que imprima tanto un mensaje de error como el archivo y el número de línea si el nivel de depuración es 2, que imprima un mensaje (sin archivo ni número de línea) si el nivel es 1, y que no haga nada si el nivel es 0.

```
1:     #include <iostream.h>
2:
3:     #ifndef DEBUG
4:     #define ASSERT(x)
5:     #elif DEBUG == 1
6:     #define ASSERT(x) \
7:         if (! (x)) \
8:         { \
9:             cout << "ERROR!! Assert " << #x << " failed\n"; \
10:        }
11:    #elif DEBUG == 2
12:    #define ASSERT(x) \
13:        if (! (x)) \
14:        { \
15:            cout << "ERROR!! Assert " << #x << " failed\n"; \
16:            cout << " on line " << __LINE__ << "\n"; \
17:            cout << " in file " << __FILE__ << "\n"; \
18:        }
19:    #endif
```

3. Escriba una macro llamada `DImprimir` que evalúe si `DEPURAR` está definida y, de ser así, que imprima el valor que se pasa como parámetro.

```
#ifndef DEPURAR
```

```
#define DImprimir(CADENA)
#else
#define DImprimir(CADENA) cout << #CADENA ;
#endif
```

4. Escriba un programa que sume dos números sin utilizar el operador de suma (+).  
 Pista: ¡Use los operadores a nivel de bits!

Si echa una mirada a la suma de dos bits, notará que la respuesta contiene dos bits de longitud: el bit de resultado y el bit de acarreo. Así, sumar 1 más 1 en binario tendrá como resultado 0 con un acarreo de 1. Si sumamos 101 más 001, éstos serán los resultados:

|     |     |
|-----|-----|
| 101 | //5 |
| 001 | //1 |
| 110 | //6 |

Si se suman dos bits “encendidos” (cada uno con el valor 1), el resultado será 0 y tendrá un acarreo de 1. Si se suman dos bits apagados, tanto el resultado como el acarreo serán 0. Si se suman un bit encendido y uno apagado, el resultado será 1 con un acarreo de 0. Aquí se muestra una tabla que resume estas reglas:

| bit1 | bit2 | acarreo | resultado |
|------|------|---------|-----------|
| 0    | 0    | 0       | 0         |
| 0    | 1    | 0       | 1         |
| 1    | 0    | 0       | 1         |
| 1    | 1    | 1       | 0         |

Examine la lógica del bit de acarreo. Si ambos bits (bit1 y bit2) son 0 o cualquiera de ellos lo es, el bit de acarreo es 0. Sólo si ambos bits son 1 el bit de acarreo valdrá 1. Éste es exactamente el mismo comportamiento que el operador AND (&).

De la misma manera, el resultado de la suma es una operación XOR (^): si cualquiera de los bits es 1 (pero no ambos), el resultado es 1; de otra forma, el resultado es 0.

Cuando tenga un acarreo, éste se sumará al bit más significativo (al bit que está a la izquierda). Esto implica que se puede utilizar una recursión o una iteración.

```
#include <iostream.h>

unsigned int suma(unsigned int bit1, unsigned int bit2)
{
    unsigned int resultado, acarreo;
    while (1)
    {
        resultado = bit1 ^ bit2;
        acarreo = bit1 & bit2;
        if (acarreo == 0)
```

```
        break;

        bit1 = acarreo << 1;
        bit2 = resultado;
    }

    return resultado;
}

int main()
{
    unsigned long a, b;
    for (;;)
    {
        cout << "Escriba dos números. (0 0 para salir): ";
        cin >> a >> b;
        if (!a && !b)
            break;
        cout << a << " + " << b << " = " << suma(a,b) << endl;
    }
    return 0;
}
```

Como una alternativa, puede resolver el problema con recursión:

```
#include <iostream.h>

unsigned int suma(unsigned int bit1, unsigned int bit2)
{
    unsigned int acarreo = bit1 & bit2;
    unsigned int resultado = bit1 ^ bit2;

    if (acarreo)
        return suma(resultado, acarreo << 1);
    else
        return resultado;
}

int main()
{
    unsigned long a, b;
    for (;;)
    {
        cout << "Escriba dos números. (0 0 para salir): ";
        cin >> a >> b;
        if (!a && !b)
            break;
        cout << a << " + " << b << " = " << suma(a,b) << endl;
    }
    return 0;
}
```

## Día 22

### Cuestionario

1. ¿Qué es POSIX?

POSIX es el resultado de un intento por crear una interfaz de programación estandarizada para sistemas operativos, iniciado en los 80. La palabra POSIX se deriva de “Intefaz Portable de Sistema Operativo”.

D

2. ¿Qué es X Windows?

X Windows es un sistema de ventanas desarrollado en el MIT a mediados de los 80. Es un estándar complejo construido sobre el modelo cliente/servidor.

3. ¿Cuáles son los dos principales editores de texto disponibles en Linux?

Los principales editores de texto que se incluyen con Linux son vim y emacs de GNU. vim es una implementación de código abierto del editor vi. emacs de GNU es una implementación de código abierto del editor emacs.

4. Cite una de las principales distinciones entre vi y emacs de GNU

vi y vim son editores de “modo” porque tienen distintos modos de operación. En modo de inserción, todos los caracteres escritos se insertan en el archivo editado. En modo de comandos, los caracteres se interpretan como comandos del editor (no todos los caracteres tienen un comando asociado). emacs de GNU es un editor “sin modo” porque no tiene distintos modos de operación. En todo momento, los caracteres que escribe pueden ser comandos para el editor o pueden ser texto para insertar en el archivo, esto depende de la secuencia de caracteres que escriba.

5. Cite una de las ventajas de las bibliotecas compartidas en comparación con las bibliotecas estáticas, y cite una de las ventajas de las bibliotecas estáticas en comparación con las bibliotecas compartidas.

Las bibliotecas compartidas producen programas ejecutables pequeños. Además, permiten las actualizaciones y mejoras de un programa sin requerir la reconstrucción total de mismo. Las bibliotecas estáticas simplifican la instalación porque no se necesita conocer la localización donde serán instaladas las distintas bibliotecas, además de eliminar su búsqueda a la hora de cargar el programa para su ejecución.

6. ¿Qué utilería se usa para compilar y crear programas? ¿Cuál es su archivo de entrada predeterminado?

La utilería make se utiliza para compilar y construir los programas. El archivo de entrada predeterminado es **GNUmakefile**. Si no se encuentra, make buscará un archivo llamado **makefile**, y si tampoco se encuentra buscará uno llamado **Makefile**.

### Ejercicios

1. Cree una función adicional para el programa de los datos que se mostró en la lección de hoy. La función debe tomar como entrada un apuntador al arreglo Dado.

Para cada cara del dado, esta función debe imprimir el porcentaje de veces que salió esa cara. La función debe estar en un archivo separado de `main()` y de `tirarDado()`.

```
#include <stdio.h>

void
doAverage(int * Die)
{
    int     i;
    int     TotRolls = 0;
    double Pct;
    double TotPct = 0.0;

    printf("\n\n");
    for(i = 0; i < 6; i++) {
        TotRolls += Die[i];
    }

    printf("%d total rolls.\n", TotRolls);

    for(i = 0; i < 6; i++) {
        Pct = ((float)Die[i] / (float)TotRolls) * 100.0;
        printf("\t%2d\t%5.2f%\n", i, Pct);
        TotPct += Pct;
    }
    printf("\t\t=====\\n\t\t%5.2f%%\\n", TotPct);
}
```

2. Modifique el archivo `make` para enlazar la nueva función.

```
#
# Makefile para los ejercicios del capítulo 22
# sin bibliotecas compartidas.
#
#
# $Header$
# $Id$
#
# Makefile para construir el programa del dado
# sin bibliotecas compartidas.

CFLAGS = -O
OBJS   = dice_ex.o doRoll_ex.o doAverage_ex.o

all:    dice
```

```
dice: $(OBJS)
$(CC) $(CFLAGS) -o $@ $(OBJS)

clean:
- $(RM) dice *.o
```

3. Analice el programa paso a paso con gdb.

## Día 23

D

### Cuestionario

1. ¿Qué es un shell?

Un shell es un programa que actúa como la interfaz de usuario para el sistema operativo. Lee los comandos que el usuario escribe, los interpreta y finalmente se los envía al kernel para su ejecución.

2. ¿Cuál es la sintaxis general de una línea de comandos de shell?

comando opcion1 ... opcionN argumento1 ... argumentoN

3. ¿Cuáles son los tres archivos de E/S disponibles para los programas?

Los tres archivos de E/S disponibles para un programa son la “entrada estándar” (comúnmente el teclado), la “salida estándar” (comúnmente la pantalla) y el “error estándar” (comúnmente la pantalla).

4. ¿Cuáles son las 3 formas de redirección de E/S, y qué caracteres se utilizan para representarlos en la línea de comandos?

Las tres principales formas de redirección de E/S son: redireccionamiento de entrada (<), redireccionamiento de salida (>) y las tuberías (!).

5. ¿Qué son las variables de entorno, “locales” o “globales”? ¿Y las variables de shell?

Las variables de entorno son globales y se pasan a cualquier programa o intérprete de comandos llamado por el intérprete actual. Las variables de shell son locales en el intérprete actual.

6. ¿Cuál variable de entorno de bash establece la ruta de búsqueda de comandos?

¿Cuál establece el indicador de comandos?

En el intérprete bash, la variable PATH establece la ruta de búsqueda de comandos y la variable PS1 establece el indicador de comandos.

7. Nombre 2 caracteres de sustitución (comodines) de la línea de comandos.

El asterisco (\*) es un comodín que sustituirá cualquier secuencia de 0 o más caracteres. El carácter de interrogación (de cierre) sustituirá un solo carácter.

8. ¿Qué necesita haber en un archivo de secuencia de comandos de shell para que el shell sepa a cuál intérprete debe enviar la secuencia de comandos?

La primera línea de cualquier archivo de secuencia de comandos de shell debe incluir los caracteres `#!` seguidos de la trayectoria y el nombre del intérprete que procesará el archivo. Por ejemplo, la línea 1 de un archivo de secuencia de comandos para bash debe ser `#!/bin/sh`.

## Ejercicio

1. Escriba una secuencia de comandos de bash para imprimir todos los argumentos de línea de comandos, además del número total de argumentos que reciba.

```
#!/bin/bash
#
echo "$# argumentos"

for i
do
    echo "argumento <$i>"
done
```

## Día 24

### Cuestionario

1. Enliste y defina los estados de un proceso.
  - \* `TASK_RUNNING` El proceso está en espera de ser ejecutado.
  - \* `TASK_INTERRUPTIBLE` El proceso está en ejecución y puede ser interrumpido.
  - \* `TASK_UNINTERRUPTIBLE` El proceso está en ejecución y no puede ser interrumpido.
  - \* `TASK_ZOMBIE` El proceso está detenido pero el sistema considera que aún está ejecutándose.
  - \* `TASK_STOPPED` El proceso está detenido, comúnmente por recibir una señal.
  - \* `TASK_SWAPPING` El sistema está intercambiando este proceso con algún otro.
2. Describa la diferencia entre la directiva de programación FIFO y la directiva de programación RR.

FIFO (Primero en Entrar Primero en Salir) programa cada proceso ejecutable en el orden en que se colocó en la cola de ejecución y ese orden nunca será cambiado. Un proceso FIFO se ejecutará hasta que se bloquee por cuestiones relacionadas con la E/S o hasta que sea desplazado por un proceso de mayor prioridad.

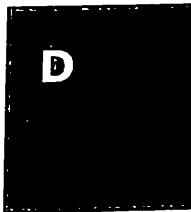
La programación de procesos RR ejecuta los procesos de tiempo real en turno (es decir, ejecuta el proceso que se encuentre al principio de la cola de procesos). La diferencia entre un proceso FIFO y un RR es que este último se ejecutará por un tiempo específico (cuanto) y entonces será expulsado y colocado al final de la cola de procesos.

3. ¿Cuál es la diferencia entre un semáforo binario y un semáforo de conteo?

Conceptualmente, existen dos tipos de semáforos: los binarios y los de conteo. Un semáforo binario sólo toma los valores cero y uno, y funciona como un mutex. Un semáforo de conteo puede tomar valores arbitrarios ya sea cero o cualquier número positivo.

4. Enliste y defina los cuatro requisitos para que se produzca un punto muerto.

1. Exclusión mutua. Por lo menos un bloqueo no es compatible.
2. Ocupar y esperar. Un subproceso está ocupando un recurso y esperando un recurso que está siendo ocupado por otro subproceso.
3. No preferencia. Un recurso ocupado sólo puede ser liberado por el subproceso que lo posee.
4. Espera circular. Debe existir un conjunto de subprocesos en espera, {t<sub>0</sub>, t<sub>1</sub>, t<sub>2</sub>, ... t(n)} en donde t<sub>0</sub> está esperando un bloqueo ocupado por t<sub>1</sub>, t<sub>1</sub> está esperando un bloqueo ocupado por t<sub>2</sub>, ... t(n-1) y t(n) está esperando un bloqueo ocupado por t<sub>0</sub>.



## Ejercicios

1. Usando un semáforo de conteo, ¿cómo podría resolver la “condición de carrera” al iniciar subprocesos?

\* Crear un semáforo y decrementar su valor al número de subprocesos que se están iniciando; esto es, si existen tres subprocesos, se debe decrementar el semáforo tres veces.

\* Obtener el semáforo al principio de cada subproceso.

\* El subproceso será bloqueado hasta que todos los subprocesos se hayan iniciado.

2. Implemente el ejemplo reentrant del listado 24.8 usando el objeto variable de condición CondVar.

```
#include <iostream.h>

#include "tcreate.h"
#include "mutex.h"

int data;

void read_thread(void* param)
{
    CondVar* apCond = static_cast<CondVar*>(param);

    while (1)
    {
```

```
    apCond->Wait();
    cout << "leer: " << data << endl;
    apCond->Signal();
}
}

void write_thread(void* param)
{
    CondVar* apCond = static_cast<CondVar*>(param);

    while(1)
    {
        apCond->Wait();
        cout << "escribir: " << data++ << endl;
        apCond->Signal();
    }
}

int main(int argc, char** argv)
{
    CondVar lock;
    Thread thread1((void*)&write_thread, &lock);
    Thread thread2((void*)&read_thread, &lock);

    lock.Create();

    thread1.Create();
    thread2.Create();

    for (int i = 0; i < 100000; i++)
        ; // La instrucción nula

    lock.Destroy();

    thread1.Destroy();
    thread2.Destroy();

    return 0;
}
```

## Día 25

### Cuestionario

1. Enliste las tres rutinas utilizadas para crear métodos de comunicación entre procesos de System V

msgget, semget y shmget

2. ¿Qué señal se produce si los extremos de lectura y de escritura de una tubería no están preparados?

SIGPIPE

3. ¿Por qué la memoria compartida es más rápida que los mensajes?

La copia de los datos no se realiza desde el núcleo; realmente se realiza el acceso desde/hacia la memoria compartida directamente.

4. ¿Qué es un semáforo binario?

Un semáforo que sólo puede tener los valores cero o uno.

D

## Ejercicios

1. Implemente un programa cliente/servidor en el que el cliente y el servidor comparten datos usando la clase SharedMemory, y sincronice el acceso a la memoria compartida usando la clase Semaphore.

El primer listado es el cliente; el segundo listado es el servidor; el tercero es el archivo make y el último es un archivo de encabezado.

```
#include <lst25-13.h>
#include <lst25-15.h>
#include "ex1.h"
int main(int argc, char** argv)
{
    char Buffer[BUFFER_SIZE]; // crea un semáforo
    Key* semkey = new Key();
    semkey->Create(SEM_KEY);
    Semaphore* sem = new Semaphore(*semkey);
    sem->Create(SEM_PERM);

    // crea memoria compartida
    Key *smkey = new Key();
    smkey->Create(SMEM_KEY);
    SharedMemory *smem = new SharedMemory(*smkey);
    smem->Create(SMEM_PERM, BUFFER_SIZE);
    smem->Attach();

    while (1)
    {
        sem->Acquire();
        memcpy(Buffer, smem, BUFFER_SIZE);
        sem->Release();
    }

    // limpieza
    sem->Destroy();
    smem->Destroy();
```

```
    delete sem;
    delete semkey;
    delete smem;
    delete smkey;
    return (0);
}

#include <lst25-13.h>
#include <lst25-15.h>
#include "ex1.h"

int main(int argc, char** argv)
{
    // crea un semáforo
    Key* semkey = new Key();
    semkey->Create(SEM_KEY);
    Semaphore* sem = new Semaphore(*semkey);
    sem->Create(SEM_PERM);

    // crea memoria compartida
    Key *smkey = new Key();
    smkey->Create(SMEM_KEY);
    SharedMemory *smem = new SharedMemory(*smkey);
    smem->Create(SMEM_PERM, BUFFER_SIZE);
    smem->Attach();

    while (1)
    {
        int i = 0;

        sem->Acquire();
        memset(smem, i, BUFFER_SIZE);
        sem->Release();
    }

    // limpia
    sem->Destroy();
    smem->Destroy();
    delete sem;
    delete semkey;
    delete smem;
    delete smkey;

    return (0);
}

INCLUDES= -I../inc -I../SharedMemory -I../Semaphores -I../Key
CFLAGS= -Wall
OBJS=../obj/key.o ../obj/semap.o ../obj/smem.o
all: client server
client: client.cxx
    gcc -o client client.cxx $(OBJS) -lstdc++ $(INCLUDES)
server: server.cxx
    gcc -o server server.cxx $(OBJS) -lstdc++ $(INCLUDES)
```

```
#define SEM_KEY      1122
#define SEM_PERM     0666
#define SMEM_KEY     5678
#define SMEM_PERM    0666
#define BUFFER_SIZE   4096
```

2. Usando tuberías, prepare una comunicación dúplex total entre un proceso padre y un proceso hijo.

El primer listado es el programa principal; el segundo listado es el archivo make y el último listado es la tubería (que es el mismo listado lst25-02.cxx de este día).

D

```
#include <iostream>
#include "lst25-02.h"

using namespace std;

int main(int argc, char** argv)
{
    Pipe* rp = new Pipe;
    Pipe* wp = new Pipe;

    char msg[] = "¡Hola, mundo!\n";
    char buf[128];

    rp->Create();
    wp->Create();

    if (fork() > 0)
    {
        rp->SetToRead();
        rp->ReadFromPipe(static_cast<char*>(buf));
        wp->SetToWrite();
        wp->WriteToPipe(static_cast<char*>(msg));

        cout << buf << endl;
    }
    else
    {
        rp->SetToWrite();
        rp->WriteToPipe(static_cast<char*>(msg));
        wp->SetToRead();
        wp->ReadFromPipe(static_cast<char*>(buf));
    }

    delete rp;
    delete wp;
    return (0);
}
```

```
INCLUDES=-I../inc -I../Key -I../Pipes
CFLAGS=-c
```

```
OBJS=pipe.o

all: pipetest

main.o : main.cxx
    gcc $(CFLAGS) -o main.o main.cxx $(INCLUDES)

pipetest: pipe.o main.o
    gcc -o pipetest main.o $(OBJS) -lstdc++

#include <iostream>
#include "lst25-02.h"

Pipe::Pipe():
    init_(false),
    read_(false)
{}

Pipe::~Pipe()
{
    if (init_)
        Destroy();
}

int Pipe::Create()
{
    if (pipe(pipe_) >= 0)
        init_ = true;
}

int Pipe::Destroy()
{
    if (read_)
        close(pipe_[0]);
    else
        close(pipe_[1]);
}

void Pipe::SetToRead()
{
    read_ = true;
    close(pipe_[1]);
}

void Pipe::SetToWrite()
{
    read_ = false;
    close(pipe_[0]);
}

int Pipe::ReadFromPipe(char* buf)
{
```

```
if (!read_)
    return -1;

    read(pipe_[0], buf, strlen(buf));
}

int Pipe::WriteToPipe(char* buf)
{
    if (read_)
        return -1;

    write(pipe_[1], buf, strlen(buf));
}
```

3. Extienda la clase `NamedPipe` para que pueda abrir tuberías que no se bloqueen y una función miembro `Read()` que no bloquee si no hay datos disponibles. Hacer esto permite que el objeto `NamedPipe` detecte si no existen datos y entonces realice otras acciones.

El primer listado es el cliente; el segundo es el servidor; el tercero es la implementación de la tubería con nombre; el cuarto es el archivo de encabezado y el último es el archivo make.

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

int main(int argc, char**argv)
{
    NamedPipe* pipe = new NamedPipe;

    pipe->Create();
    pipe->Open();

    int i = 0;
    do
    {
        const int len = 32;
        char buf[len];

        sprintf(buf, "%d", i++);

        pipe->Write(&buf[0], static_cast<int>(strlen(buf)));

        cout << "client" << buf << endl;
    } while (i < 10);

    pipe->Close();
```

D

```
    pipe->Destroy();

    delete pipe;
}

#include <iostream>
#include "np.h"

int main(int argc, char**argv)
{
    NamedPipe* pipe = new NamedPipe;

    pipe->Create();
    pipe->Open();

    int i = 0;
    do
    {
        const int len = 32;

        char buf[len];

        if (pipe->Read(buf, len) > 0)
            cout << "server" << buf << endl;

    } while (i < 10);

    pipe->Close();
    pipe->Destroy();

    delete pipe;
}

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "np.h"

NamedPipe::NamedPipe():
    init_(false),
    fp_(static_cast<int>(0)),
    nombre_(static_cast<char*>(0))
{
}

NamedPipe::~NamedPipe()
{
    if (init_)
```

```
        Destroy();
    }

    int NamedPipe::Create()
    {
        char nombre[] = "NP";
        umask(0);

        int stat = mknod(nombre, S_IFIFO | 0666, 0);

        nombre_ = new char[strlen(nombre)];
        strcpy(nombre_, nombre);
        init_ = true;
    }

    int NamedPipe::Open()
    {
        // La clave para no bloquear la E/S es el indicador O_NONBLOCK
        fp_ = open(nombre_, O_RDWR | O_NONBLOCK);
    }

    int NamedPipe::Close()
    {
        close(fp_);
        fp_ = static_cast<int>(0);
    }

    int NamedPipe::Create(char* nombre)
    {
        mknod(nombre, S_IFIFO|0666, 0);

        nombre_ = new char[strlen(nombre)];
        strcpy(nombre_, nombre);
        init_ = true;
    }

    int NamedPipe::Destroy()
    {
        if (fp_ != 0)
            close(fp_);

        delete []nombre_;
        init_ = false;
    }

    int NamedPipe::Read(char* buf, int len)
    {
        len = read(fp_, buf, len);

        return len;
    }
```

D

```
}

int NamedPipe::Write(char* buf, int len)
{
    write(fp_, buf, len);

    return len;
}

// Listado 25.3 La clase NamedPipe

#ifndef C_NP_H
#define C_NP_H

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include "lst25-01.h" // #include "object.h"

class NamedPipe : public Object
{
public:
    NamedPipe();
    ~NamedPipe();
    int Create();
    int Create(char * name);
    int Destroy();
    int Open();
    int Close();
    int Read(char * buf, int len);
    int Write(char * buf, int len);
private:
    // no permitir la copia
    NamedPipe & operator=(const NamedPipe &);

    bool init_;
    FILE * fp_;
    char * name_;
};

#endif

INCLUDES=-I../inc -I../Key
CFLAGS=-c
OBJS= ../obj/key.o ../obj/np.o

all: client server

np.o : np.cxx np.h
```

```
gcc $(CFLAGS) -o np.o np.cxx $(INCLUDES)
cp np.o ../obj

client.o : client.cxx
    gcc $(CFLAGS) -o client.o client.cxx $(INCLUDES)

server.o : server.cxx
    gcc $(CFLAGS) -o server.o server.cxx $(INCLUDES)

client: np.o client.o
    gcc -o client client.o $(OBJS) -lstdc++

server: np.o server.o
    gcc -o server server.o $(OBJS) -lstdc++
```

D

## Día 26

### Cuestionario

1. ¿Cuál es la diferencia entre programas controlados por eventos y programas controlados por procedimientos?

Un programa controlado por procedimientos se ejecuta de principio a fin, siguiendo las trayectorias de control que se establecen a partir de la entrada. Un programa controlado por eventos espera en un ciclo diversos eventos externos que lo afectarán y realizará funciones como respuesta a dichos eventos.

2. ¿Qué es un widget, en términos computacionales?

Un widget es un elemento de la GUI que muestra información u ofrece una manera específica para que el usuario interactúe con el sistema operativo y con las diversas aplicaciones.

3. ¿Qué es una función callback, y por qué son propensas a errores?

Una función callback es una llamada a función que se pasa como referencia a otra función. Estas funciones se definen con el tipo `(void) (* func)()`, por lo que el compilador no puede conocer si la función que se pasará como argumento es del tipo adecuado. Aun si la declaración del tipo es segura, programadores sin escrúpulos pueden utilizar la conversión de tipos para empeorar los efectos. Lo peor de todo es que el apuntador puede ser nulo o inválido, y esto llevará a un terrible "core dump".

4. ¿Qué es una ranura Qt, y cómo reacciona a la señales Qt?

Una ranura Qt es una función miembro normal, que se vincula a una señal utilizando la función `connect()`. Las ranuras que se conectan a señales de esta manera serán invocadas cuando un objeto específico emita la señal correspondiente.

5. Las señales y las ranuras ofrecen seguridad de tipos. ¿Qué significa esto y por qué es algo bueno?

Las ranuras y las señales siempre son del tipo correcto, lo que indica que una ranura nunca recibe el tipo de señal equivocado y nunca tendrá los terribles "core dumps". De hecho, las ranuras y las señales no están asociados estrechamente; cuando un objeto emite una señal no se sabe si la señal ha sido atrapada por una ranura o se ha ignorado completamente.

6. ¿Cómo enlazaría el siguiente manejador de eventos de wxWindows a un evento EVT\_MENU desde un elemento de menú con el identificador de ID\_MY\_HELP?

Asuma que OnHelp() es miembro de la clase MyFrame, y que MyFrame se deriva de wxFrame.

```
void OnHelp(wxCommandEvent & WXUNUSED(event));
```

Con la siguiente tabla de eventos:

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(ID_MY_HELP, MyFrame::OnHelp)
END_EVENT_TABLE()
```

7. ¿Qué hace la macro IMPLEMENT\_APP()?

Esta macro implementa la función main() en una aplicación de wxWindows y oculta los detalles del ciclo de eventos principal al programador.

8. ¿Qué hace la macro Q\_OBJECT?

Esta macro la utiliza el compilador de metaobjetos (MOC) para convertir las secciones signals: y slots: de Qt, que se encuentran en las declaraciones de clase de KDE, a la forma en que un compilador de C++ estándar pueda entender.

## Ejercicios

1. Usando el archivo fuente 1st26-01.cxx como base, extienda el programa para que la clase callback muestre un cuadro de diálogo al hacer clic en los botones, en lugar de escribir directamente en stdout.

Aquí verá que creamos una nueva función miembro estática llamada callback::do\_message() como nuestra función callback. En ella creamos un cuadro de diálogo con la llamada a gnome\_message\_box\_new() y la mostramos con la llamada a gnome\_dialog\_run(). Después de esto, llamamos a do\_message() desde la función callback que conectamos con gtk\_signal\_connect() en main().

```
// el archivo de encabezados de gnome principal
#include <gnome.h>
```

```
class callback {
public:
    static void button_clicked(GtkWidget *button, gpointer data);
    static gint delete_event(GtkWidget *widget,
    →GdkEvent *event, gpointer data);
private:
```

```
    static void do_message(const char*);  
};  
  
void  
callback::do_message(const char * msg)  
{  
    GtkWidget *dialogue;  
    int ret;  
    dialogue = gnome_message_box_new (  
        msg,  
        "",  
        "OK",  
        NULL);  
    ret = gnome_dialog_run (GNOME_DIALOG (dialogue));  
}  
  
void  
callback::button_clicked(GtkWidget *button, gpointer data)  
{  
    do_message((char*)data);  
}  
  
// Llamada cuando el usuario cierra la ventana.  
gint  
callback::delete_event(GtkWidget *widget, GdkEvent *event,  
                      gpointer data)  
{  
    // Indica el ciclo principal que va a salir.  
    g_print("using C++ callback to quit\n");  
    gtk_main_quit();  
    // Devuelve FALSE para seguir cerrando la ventana.  
    return FALSE;  
}  
  
int  
main(int argc, char *argv[])  
{  
    GtkWidget *app;  
    GtkWidget *button;  
    GtkWidget *hbox;  
  
    // Inicializa GNOME, es muy similar a gth_int.  
    gnome_init ("buttons-basic-example", "0.1", argc, argv);  
  
    // Crea un widget de aplicación de Gnome que configura  
    // una ventana básica para su aplicación.  
    app = gnome_app_new ("buttons-basic-example",  
                        "Buttons");  
  
    /* asocia "delete_event", el evento que obtenemos cuando
```

```
el usuario cierra una ventana desde el manejador de ventanas
a gtk_main_quit, la función que causa la salida del ciclo
gtk_main y por lo tanto de la aplicación*/
gtk_signal_connect (GTK_OBJECT (app), "delete_event",
                     GTK_SIGNAL_FUNC (callback::delete_event),
                     NULL);

// Crea un cuadro horizontal para los botones
// y lo agrega en el widget de aplicación.
hbox = gtk_hbox_new (FALSE,5);
gnome_app_set_contents (GNOME_APP (app), hbox);

// Crea un botón y lo agrega al cuadro horizontal,
// le asocia el clic del ratón al método button_clicked.
button = gtk_button_new_with_label("Button 1");
gtk_box_pack_start (GTK_BOX(hbox), button, FALSE, FALSE, 0);
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                     GTK_SIGNAL_FUNC (callback::button_clicked),
                     "Button 1\n");

// y otro botón.
button = gtk_button_new_with_label("Button 2");
gtk_box_pack_start (GTK_BOX(hbox), button, FALSE, FALSE, 0);
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                     GTK_SIGNAL_FUNC (callback::button_clicked),
                     "Button 2\n");

// Muestra todo lo que está dentro del widget
// de aplicación y al widget mismo.
gtk_widget_show_all(app);

// Entra al ciclo principal.
gtk_main ();

return 0;
}
```

2. Tomando como base el ejemplo final de wxWindows o el de KDE, extienda el código para que despliegue un botón de exploración de archivos que le permita seleccionar un archivo por medio de un cuadro de diálogo estándar de selección de archivos.

En el listado ex26-02KDE.h verá que hemos agregado una variable miembro button adicional, `m_btnBrowse`. La inicializamos al crear un nuevo botón en ex26-02-KDE.cxx y utilizamos el mecanismo de señal/ranura para conectarla al manejador `SlotBrowse()` de la aplicación. Dentro del manejador utilizamos la función miembro estática `QFileDialog::getOpenFileName()` para abrir un diálogo estándar para examinar archivos. Vea que no tenemos el archivo `#include "ex26-01KDE.moc"` en nuestros listados. Esto es normal porque MOC crea el archivo `.moc` por nosotros cuando procesa el archivo de encabezado.

```
/*
 * ex26-01KDE.h
 */

#include <kapp.h>
#include <ktmainwindow.h>
#include <qpushbutton.h>
#include <kmenubar.h>
#include <qpopupmenu.h>

class KDEHelloWorld : public KTMainWindow
{
    Q_OBJECT
public:
    KDEHelloWorld();
    void closeEvent(QCloseEvent * );
public slots:
    void SlotGreet();
    void SlotQuit();
    void SlotBrowse();
private:
    QPushButton *m_btnGreet;
    QPushButton *m_btnQuit;
    QPushButton *m_btnBrowse;
    KMenuBar *m_Menu;
    QPopupMenu *m_MenuApp;
};

/*
 * ex26-01KDE.cxx
 */

#include "ex26KDE-01.moc"
#include <kapp.h>
#include <ktmainwindow.h>
#include <kmsgbox.h>

KDEHelloWorld::KDEHelloWorld() : KTMainWindow()
{
    m_btnGreet = new QPushButton("Greet", this);
    m_btnGreet->setGeometry(45,30,50,20);
    m_btnGreet->show();
    connect(m_btnGreet, SIGNAL(clicked()), this, SLOT(SlotGreet()));

    m_btnQuit = new QPushButton("Exit", this);
    m_btnQuit->setGeometry(105,30,50,20);
    m_btnQuit->show();
    connect(m_btnQuit, SIGNAL(clicked()), this, SLOT(SlotQuit()));

    m_btnQuit = new QPushButton("Browse", this);
    m_btnQuit->setGeometry(155,30,50,20);
}
```

```
m_btnQuit->show();
connect(m_btnQuit, SIGNAL(clicked()), this, SLOT(SlotBrowse()));

m_MenuApp = new QPopupMenu();
m_MenuApp->insertItem("&Greet", this, SLOT(SlotGreet()));
m_MenuApp->insertItem("&Quit", this, SLOT(SlotQuit()));
m_MenuApp->insertItem("&Browse", this, SLOT(SlotBrowse()));
m_Menu = new KMenuBar(this);
m_Menu->insertItem("&Application", m_MenuApp);
}

void KDEHelloWorld::closeEvent(QCloseEvent *)
{
    kapp->quit();
}

void KDEHelloWorld::SlotGreet()
{
    KMessageBox::message(0,"KDEHelloWorld","Hello World!");
}

void KDEHelloWorld::SlotBrowse()
{
    QCadena fileName = QFileDialog::getOpenFileName(
    QCadena::null, QCadena::null, this);
}

void KDEHelloWorld::SlotQuit()
{
    close();
}

int main(int argc, char **argv)
{
    KApplication MyApp(argc, argv);
    KTMainWindow *MyWindow = new KTMainWindow();
    MyWindow->setGeometry(50,50,200,100);

    MyApp.setMainWidget(MyWindow);
    MyWindow->show();
    return MyApp.exec();
}

//Compile estos archivos con los comandos:
//moc ex26-01KDE.h -o ex26-01KDE.moc
//g++ -c -I$KDEDIR/include -I$QTDIR -fno-rtti ex26-01KDE.cxx
//g++ -L$KDEDIR/lib -lkdecore -lkdeui -lqt -o ex26-01KDE ex26-01KDE.o
```

En el archivo `ex26·02wxWin.cxx`, tenemos que agregar una variable miembro button adicional, `m_btnBrowse` en la declaración de la clase `MyFrame`. Inicializamos la variable y creamos nuestro botón en el constructor de `MyFrame()`, después conectamos el identificador del evento `ID_Browse` al manejador `MyFrame::OnBrowse()` en la declaración de `EVENT_TABLE()`. Cuando haga clic en el botón, el manejador invoca a `wxFileSelector()` para mostrar un diálogo estándar para examinar archivos.

```
/*
 * ex26·02wxWin.cxx
 */

#ifndef __GNUG__
// #pragma implementation
#endif

// Para compiladores que soporten precompilación, incluya "wx/wx.h".
#include "wx/wxprec.h"

#ifndef __BORLANDC__
#pragma hdrstop
#endif

// Para compiladores que soporten precompilación, incluya "wx/wx.h".
#ifndef WX_PRECOMP
#include "wx/wx.h"
#endif

class MyApp: public wxApp
{
    virtual bool OnInit();
};

class MyFrame: public wxFrame
{
public:

    MyFrame(const wxString& title, const wxPoint& pos,
    const wxSize& size);
    void OnQuit(wxCommandEvent& event);
    void OnGreet(wxCommandEvent& event);
    void OnBrowse(wxCommandEvent& event);
    DECLARE_EVENT_TABLE()
private:
    wxPanel *m_panel;
    wxButton *m_btnGreet;
    wxButton *m_btnQuit;
    wxButton *m_btnBrowse;
};
```

D

```
enum
{
    ID_Quit = 1,
    ID_Greet,
    ID_Browse,
};

BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(ID_Quit, MyFrame::OnQuit)
    EVT_MENU(ID_Greet, MyFrame::OnGreet)
    EVT_BUTTON(ID_Greet, MyFrame::OnGreet)
    EVT_BUTTON(ID_Quit, MyFrame::OnQuit)
    EVT_BUTTON(ID_Browse, MyFrame::OnBrowse)
END_EVENT_TABLE()

IMPLEMENT_APP(MyApp)

bool MyApp::OnInit()
{
    MyFrame *frame = new MyFrame("Hello World",
    wxDefaultPosition, wxDefaultSize);
    frame->Show(TRUE);
    SetTopWindow(frame);
    return TRUE;
}

MyFrame::MyFrame(const wxString& title,
    const wxPoint& pos, const wxSize& size)
: wxFrame((wxFrame *)NULL, -1, title, pos, size)
{

    wxSize panelSize = GetClientSize();
    const int buttonWidth=50;
    const int buttonHeight=25;
    const int buttonSpacing=10;
    const wxSize buttonSize(buttonWidth, buttonHeight);

    int height = panelSize.GetHeight();
    int width = panelSize.GetWidth();
    m_panel = new wxPanel(this, -1, wxDefaultPosition, panelSize);

    m_btnGreet = new wxButton(m_panel, ID_Greet,
        _T("Greet..."),
        wxDefaultPosition -
    (buttonWidth/2+buttonWidth+buttonSpacing), 10), buttonSize);
    m_btnQuit = new wxButton(m_panel, ID_Quit,
```

```
    _T("Quit"),
    wxPoint((width/2)-(buttonWidth/2),
    -10), buttonSize);

    m_btnBrowse = new wxButton(m_panel, ID_Browse,
        _T("Browse..."),
        wxPoint((width/2)-
    -(buttonWidth/2+buttonSpacing), 10), buttonSize);

    wxMenu *menuApp = new wxMenu;

    menuApp->Append(ID_Greet, "&Greet...");
    menuApp->Append(ID_Browse, "&Browse...");
    menuApp->AppendSeparator();
    menuApp->Append(ID_Quit, "&Quit");

    wxMenuBar *menuBar = new wxMenuBar;
    menuBar->Append(menuApp, "&Application");

    SetMenuBar(menuBar);

}

void MyFrame::OnQuit(wxCommandEvent& WXUNUSED(event))
{
    Close(TRUE);
}

void MyFrame::OnBrowse(wxCommandEvent& WXUNUSED(event))
{
    const wxCadena& file = ::wxFileSelector("Select a file");
}

void MyFrame::OnGreet(wxCommandEvent& WXUNUSED(event))
{
    wxMessageBox("This is a wxWindows Hello world sample",
        "Greet: Hello World", wxOK | wxICON_INFORMATION, this);
}

// compile estos archivos con el comando:
// gcc -g -Wall `gnome-config --cflags gnome gnomeui \
// `LDFLAGS=`gnome-config \
// --libs gnome gnomeui` ex26-02wxWin.cxx -o ex26-02wxWin
```

3. Extienda el código del ejercicio 2 de forma que, al hacer clic en OK y seleccionar el archivo, el programa lo despliegue en un control de texto en la ventana principal.

D

Extendamos la declaración de la clase `KDEHelloWorld` en el listado `ex26-03KDE.h` para incluir la variable miembro `QCadena` que contendrá el nombre del archivo que seleccionaremos. También agregaremos el control `QTextV1ew` que mostrará el texto en el área del cliente. Crearemos la vista del texto en el constructor y lo posicionaremos de manera arbitraria en la ventana principal.

Ahora extendamos la función de ranura `SlotBrowse()` para tomar el nombre de archivo del diálogo para examinar archivos, que después usaremos para inicializar un objeto `QFile`. Abrimos el archivo con la función miembro `open()` de `QFile` y hacemos un ciclo de lectura, una línea a la vez, hasta que hayamos leído el archivo completo dentro de un objeto string temporal. Después de esto, estableceremos el objeto string como el texto que será mostrado en el objeto `text-view`. Es un enfoque muy simplista trabajar únicamente con archivos de texto: si lee un archivo binario, podría ver algunos resultados extraños.

```
/*
 * ex26-03KDE.h
 */

#include <kapp.h>
#include <ktmainwindow.h>
#include <qpushbutton.h>
#include <kmenubar.h>
#include <qpopupmenu.h>

class KDEHelloWorld : public KTMainWindow
{
    Q_OBJECT
public:
    KDEHelloWorld();
    void closeEvent(QCloseEvent * );
public slots:
    void SlotGreet();
    void SlotQuit();
    void SlotBrowse();
private:
    QPushButton *m_btnGreet;
    QPushButton *m_btnQuit;
    QPushButton *m_btnBrowse;
    QTextView   *m_textView;
    QCadena *m_fileName;

    KMenuBar *m_Menu;
    QPopupMenu *m_MenuApp;
};

/*
 * ex26-03KDE.cxx
 */

#include "ex26KDE-03.moc"
#include <kapp.h>
#include <ktmainwindow.h>
```

```
#include <kmsgbox.h>

KDEHelloWorld::KDEHelloWorld() : KTMMainWindow()
{
    m_btnGreet = new QPushButton("Greet", this);
    m_btnGreet->setGeometry(45,30,50,20);
    m_btnGreet->show();
    connect(m_btnGreet, SIGNAL(clicked()), this, SLOT(SlotGreet()));

    m_btnQuit = new QPushButton("Exit", this);
    m_btnQuit->setGeometry(105,30,50,20);
    m_btnQuit->show();
    connect(m_btnQuit, SIGNAL(clicked()), this, SLOT(SlotQuit()));

    m_btnQuit = new QPushButton("Browse", this);
    m_btnQuit->setGeometry(155,30,50,20);
    m_btnQuit->show();
    connect(m_btnQuit, SIGNAL(clicked()), this, SLOT(SlotBrowse()));

    m_textView = new QTextView(this);
    m_textView->setMinimumSize(450, 250);
    m_textView->show();

    m_MenuApp = new QPopupMenu();
    m_MenuApp->insertItem("&Greet", this, SLOT(SlotGreet()));
    m_MenuApp->insertItem("&Quit", this, SLOT(SlotQuit()));
    m_MenuApp->insertItem("&Browse", this, SLOT(SlotBrowse()));
    m_Menu = new KMenuBar(this);
    m_Menu->insertItem("&Application", m_MenuApp);
}

void KDEHelloWorld::closeEvent(QCloseEvent *)
{
    kapp->quit();
}

void KDEHelloWorld::SlotGreet()
{
    KMMsgBox::message(0, "KDEHelloWorld", "Hello World!");
}

void KDEHelloWorld::SlotBrowse()
{
    m_fileName = QFileDialog::getOpenFileName(QCadena::null,
    QCadena::null, this);

    if(!fileName.IsEmpty()) {
        // Necesitamos leer el archivo.

        QFile theFile(m_fileName);
```

D

```

        if (theFile.open(IO_ReadOnly)) { // Archivo abierto exitosamente
            QTextStream elFlujo(&theFile); // use un "text Stream"
            QCadena text;
            int n = 1;
            while (!elFlujo.eof()) { // Hasta el fin de archivo...
                text += elFlujo.readLine();
            } // Linea de texto sin '\n'
        }
        theFile.close();
        m_textView.setText(text);
    }
}

void KDEHelloWorld::SlotQuit()
{
    close();
}

int main(int argc, char **argv)
{
    KApplication MyApp(argc, argv);
    KTMainWindow *MyWindow = new KTMainWindow();
    MyWindow->setGeometry(100,100,500,500);

    MyApp.setMainWidget(MyWindow);
    MyWindow->show();
    return MyApp.exec();
}

//Compile estos archivos con los comandos:
//moc ex26-03KDE.h -o ex26-03KDE.moc
//g++ -c -I$KDEDIR/include -I$QTDIR -fno-rtti ex26-03KDE.cxx
//g++ -L$KDEDIR/lib -lkdecore -lkdeui -lqt -o ex26-03KDE ex26-03KDE.o

```

La siguiente es una versión más simple que la de KDE. Agregamos un control `wxTextCtrl` a la clase `MyFrame` declarada en `ex26-03wxWin.cxx`, así como `wxCadena` para almacenar el nombre del archivo. Esto es muy parecido a lo que hicimos para el ejemplo de KDE. Inicializamos el control de texto en el constructor de `MyFrame`. En el manejador de eventos, `MyFrame::OnBrowse()`, abrir y mostrar el archivo es tan simple como obtener el nombre de archivo e indicarle al control de texto que cargue los datos del archivo con la llamada a `LoadFile()`, que pertenece al objeto de control de texto.

```

/*
 * ex26-03wxWin.cxx
 */

```

```
#ifdef __GNUG__
// #pragma implementation
#endif

// Para compiladores que soporten precompilación, incluya "wx/wx.h".
#include "wx/wxprec.h"

#ifndef __BORLANDC__
#pragma hdrstop
#endif

// Para compiladores que soporten precompilación, incluya "wx/wx.h".
#ifndef WX_PRECOMP
#include "wx/wx.h"
#endif

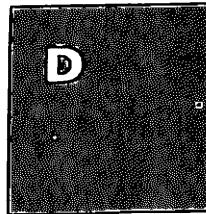
class MyApp: public wxApp
{
    virtual bool OnInit();
};

class MyFrame: public wxFrame
{
public:

    MyFrame(const wxString& title, const wxPoint& pos,
            const wxSize& size);
    void OnQuit(wxCommandEvent& event);
    void OnGreet(wxCommandEvent& event);
    void OnBrowse(wxCommandEvent& event);
    DECLARE_EVENT_TABLE()
private:
    wxPanel *m_panel;
    wxButton *m_btnGreet;
    wxButton *m_btnQuit;
    wxButton *m_btnBrowse;
    wxString m_fileName;
    wxTextCtrl *m_textViewer;
};

enum
{
    ID_Quit = 1,
    ID_Greet,
    ID_Browse,
};

BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(ID_Quit, MyFrame::OnQuit)
    EVT_MENU(ID_Greet, MyFrame::OnGreet)
```



```
EVT_BUTTON(ID_Greet, MyFrame::OnGreet)
EVT_BUTTON(ID_Quit, MyFrame::OnQuit)
EVT_BUTTON(ID_Browse, MyFrame::OnBrowse)
END_EVENT_TABLE()

IMPLEMENT_APP(MyApp)

bool MyApp::OnInit()
{
    MyFrame *frame = new MyFrame("Hello World",
➥wxPoint(50,50), wxSize(500,500));
    frame->Show(TRUE);
    SetTopWindow(frame);
    return TRUE;
}

MyFrame::MyFrame(const wxCadena& title, const wxPoint& pos,
➥const wxSize& size)
: wxFrame((wxFrame *)NULL, -1, title, pos, size)
{

    wxSize panelSize = GetClientSize();
    const int buttonWidth=50;
    const int buttonHeight=25;
    const int buttonSpacing=10;
    const wxSize buttonSize(buttonWidth, buttonHeight);

    int height = panelSize.GetHeight();
    int width = panelSize.GetWidth();
    m_panel = new wxPanel(this, -1, wxPoint(0, 0), panelSize);

    m_btnGreet = new wxButton(m_panel, ID_Greet,
                           _T("Greet..."),
                           wxDefaultPosition +
                           wxPoint((width/2)-
(buttonWidth/2+buttonWidth+buttonSpacing), 10), buttonSize);
    m_btnQuit = new wxButton(m_panel, ID_Quit,
                           _T("Quit"),
                           wxDefaultPosition +
                           wxPoint((width/2)-(buttonWidth/2),
➥10), buttonSize);

    m_btnBrowse = new wxButton(m_panel, ID_Browse,
                           _T("Browse..."),
                           wxDefaultPosition +
➥(buttonWidth/2+buttonSpacing), 10), buttonSize);

    m_textViewer = new wxTextCtrl(m_panel, ID_Browse,
                           _T(""),
                           wxDefaultPosition +
                           wxPoint(10,40),
```

```
    wxDefaultPosition, wxDefaultSize, wxBORDER_NONE);
    m_textViewer = new wxTextCtrl(this, ID_TextView,
        wxDefaultPosition, wxDefaultSize, wxTE_MULTILINE | wxTE_READONLY);

    wxMenu *menuApp = new wxMenu;

    menuApp->Append(ID_Greet, "&Greet...");
    menuApp->Append(ID_Browse, "&Browse...");
    menuApp->AppendSeparator();
    menuApp->Append(ID_Quit, "&Quit");

    wxMenuBar *menuBar = new wxMenuBar;
    menuBar->Append(menuApp, "&Application");

    SetMenuBar(menuBar);

}

void MyFrame::OnQuit(wxCommandEvent& WXUNUSED(event))
{
    Close(TRUE);
}

void MyFrame::OnBrowse(wxCommandEvent& WXUNUSED(event))
{
    m_fileName = ::wxFileSelector("Select a file");
    if(!m_fileName.IsEmpty()) {
        m_textViewer->LoadFile(m_fileName);
    }
}

void MyFrame::OnGreet(wxCommandEvent& WXUNUSED(event))
{
    wxMessageBox("This is a wxWindows Hello world sample",
        "Greet: Hello World", wxOK | wxICON_INFORMATION, this);
}

// compile este archivo con el comando:
// gcc -g -Wall `gnome-config --cflags gnome gnomeui\
// `LDFLAGS=`gnome-config \
// --libs gnome gnomeui` ex26-03wxWin.cxx -o ex26-03wxWin
```

4. Analice los resultados del ejercicio 1 y considere cómo podría envolver toda la aplicación en una sola clase. Considere por qué podría necesitar implementar las funciones callback como miembros estáticos privados de la clase, y exponer sólo “envolturas” que las llamen cuando el usuario active eventos. Vea por ejemplo la función `do_message()` en el archivo `lst26-01.cxx`. Piense cómo podría extender el concepto para crear una envoltura genérica para la aplicación, y utilizar funciones virtuales para configurar la ventana principal y conectar señales.

Puede ver que en el siguiente listado creamos una nueva clase, `TheAppClass`, y en ella declaramos los widgets de Gtk que representan la aplicación y el cuadro que contiene los botones como variables miembro.

Creamos los botones mismos en `CreateWidgets()` y los integramos dentro del cuadro. Vea que la nueva clase encapsula todos los widgets que utilizamos, y que las funciones miembro estáticas de la misma clase manejarán los eventos que se generen cuando haga clic en cada uno de los botones.

Esto significa que todo lo que necesitamos hacer en `main()` es instanciar el objeto aplicación, decirle que cree sus propios widgets y que se ejecute.

```
/*
 ex26-04.cxx - Envolvemos la aplicación como una clase. Puede expandirla
 a cualquier grado de abstracción que quiera.
 */

#include <gnome.h>

class TheAppClass {
public:
    TheAppClass(int argc, char* argv[]);
    virtual gint CreateWidgets();
    void Run();
private:
    static void button_clicked(GtkWidget *button, gpointer data);
    static gint delete_event(GtkWidget *widget, GdkEvent *event,
    →gpointer data);
    static void do_message(const char*);

    GtkWidget *app;
    GtkWidget *hbox;

};

gint
TheAppClass::CreateWidgets()
{
    GtkWidget *button=0;

    button = gtk_button_new_with_label("Button 1");
    gtk_box_pack_start (GTK_BOX(hbox), button, FALSE, FALSE, 0);

    gtk_signal_connect (GTK_OBJECT (button), "clicked",
                        GTK_SIGNAL_FUNC (TheAppClass::button_clicked),
                        "Button 1");

    button = gtk_button_new_with_label("Button 2");
    gtk_box_pack_start (GTK_BOX(hbox), button, FALSE, FALSE, 0);
    gtk_signal_connect (GTK_OBJECT (button), "clicked",
```

```
    GTK_SIGNAL_FUNC (TheAppClass::button_clicked),
    "Button 2");
return 0;
}

TheAppClass::TheAppClass(int argc, char* argv[])
{
    gnome_init ("buttons-basic-example", "0.1", argc, argv);

    app = gnome_app_new ("buttons-basic-example",
    ↪"Class-Wrapped Buttons");

    gtk_signal_connect (GTK_OBJECT (app), "delete_event",
                        GTK_SIGNAL_FUNC (TheAppClass::delete_event),
                        NULL);

    hbox = gtk_hbox_new (FALSE,5);

    gnome_app_set_contents (GNOME_APP (app), hbox);

}

void
TheAppClass::do_message(const char * msg)
{
    GtkWidget *dialogue;
    int ret;
    dialogue = gnome_message_box_new (
        msg,
        "",
        "OK",
        NULL);
    ret = gnome_dialog_run (GNOME_DIALOG (dialogue));
}

void
TheAppClass::button_clicked(GtkWidget *button, gpointer data)
{
    do_message((char*)data);
}

// Llamada cuando el usuario cierra la ventana.
 gint
TheAppClass::delete_event(GtkWidget *widget, GdkEvent *event,
    ↪gpointer data)
{
    gtk_main_quit();
    return FALSE;
}
```

```
void
TheAppClass::Run()
{
    gtk_widget_show_all(app);
    gtk_main ();
}

int
main(int argc, char *argv[])
{
    TheAppClass MyApp(argc, argv);
    MyApp.CreateWidgets();
    MyApp.Run();
    return 0;
}

// compile este archivo con el comando:
// gcc -g -Wall `gnome-config --cflags gnome gnomeui\
// `LDFLAGS=`gnome-config \
// --libs gnome gnomeui` ex26-04.cxx -o ex26-04
```

# ÍNDICE

## Símbolos

\b código de escape, 58  
\ (barra diagonal inversa), 754  
> (carácter de redirección de salida), 838  
\\" código de escape, 58  
\' código de escape, 58  
\\" código de escape, 58  
\> código de escape, 58  
// comentarios, 34, 41  
< comentarios, 34, 41  
“ ” (comillas), colocar alrededor de cadenas, 757  
[ ] (corchetes), 384, 721  
#define, instrucción  
    listado, 749-750  
    pruebas, 749  
sustituciones de cadenas, 748  
sustituciones de constantes, 749  
#else, comando del precompilador, 749-751  
#ifdef, comando del precompilador, 749  
#ifndef, comando del precompilador, 749

#include, directiva, 101-102, 748  
{ } (llaves), 31  
\n código de escape, 58  
\n (código de nueva línea), 33  
= 0, notación, 441  
# (numeral), 30, 748  
& (operador AND), 775  
>: (operador condicional), 94-95  
= (operador de asignación), 51, 68, 71, 321-322, 324  
    confusión con operador igual a (= =), 80  
    listado, 322-323  
    precedencia, 78  
- (operador de decremento), 75  
    postfijo, 75-77  
    prefijo, 77  
[ ] (operador de desplazamiento), 395  
++ (operador de incremento), 75-77  
[ ] (operador de índice), 375  
\* (operador de indirección), 229-230, 284-285  
<< (operador de inserción), 549, 553, 698  
++ (operador de prefijo), 308-310  
. (operador de punto), 155, 241  
<< (operador de redirecciónamiento), 21, 31  
& (operador de referencia), 260-262, 284-285  
% (operador de residuo), 72-73  
:: (operador de resolución de ámbito), 601  
- (operador de resta), 71-72  
[ ] (operador de subíndice), 698  
+ (operador de suma), 317-320  
+= (operador de suma autoasignado), 74  
!= (operador diferente de), 80  
== (operador igual a), 80  
&& (operador lógico AND), 91-92  
! (operador lógico NOT), 92

**<= (operador menor o igual que)**, 81  
**< (operador menor que)**, 32, 81  
**| (operador OR)**, 776  
**^ (operador OR exclusivo)**, 776  
**( ) (paréntesis)**, 78  
 agrupar, 93  
 anidación, 78-79  
 macro, sintaxis, 753-754  
**/proc, sistema de archivos**, 858  
**; (punto y coma)**, 68  
 alinear, 84, 781  
 funciones, 104  
 instrucciones if anidadas, 88-90  
**\t código de escape**, 58  
**\t (código de tabulador)**, 33  
**~ (tilde)**, 159  
**| (tubería), carácter**, 839

## A

**a nivel de bits, operadores**, 775  
 AND (&), 775  
 complemento (), 776  
 OR (|), 776  
 OR exclusivo (^), 776  
**abrazo mortal**, 870  
**abrir archivos**, 585-587  
**abstracción (en programación)**, 129  
 niveles de, 129  
 programación orientada a objetos, 620  
**abstractas, clases**, 449  
 declarar, 441  
 derivación de otros ADTs, 445-448  
 funciones virtuales puras, 440  
 Java, 450  
 listado de ejemplo, 440  
 ventajas, 452

**acceder**  
 arreglos, 375  
 atómicamente, 867  
 ayuda en línea en el editor vi, 811-812  
 clases contenidas, 508  
 datos miembro, 152  
 heap, 241-242  
 privados, 151-154  
 públicos, 152-153  
 datos miembro estáticos  
 métodos no estáticos, 459-461  
 sin objetos, 458-459  
 direcciones de memoria, 232-233  
 elementos, 694-697  
 de arreglos, 368  
 información del sistema, 858  
**miembros**  
 datos miembro en el heap (listado), 242  
 objetos, 149  
 estáticos, 691  
 objetos derivados, 338-339  
 páginas del manual de g++, 39  
**actores**, 626-628  
**ADD**, 9  
**Administrador de presentaciones de IBM**, 898  
**administrar búferes**, 560  
**ADTs (tipos de datos abstractos)**, 436-440  
 declarar, 441  
 derivar de otros, 445-448  
 funciones virtuales puras, 440  
 Java, 450  
 listados, 440-441, 445-448  
 producir errores en tiempo de compilación, 440  
 ventajas, 452  
**agregar**  
 botones en aplicaciones KDE, 928  
 dos variables Contador, 317

incremento, operadores de, 307  
 nuevos miembros a espacios de nombres, 606  
 operadores de despliegue, 673  
**agregar al final de un archivo**, 587-589  
**alcance**  
 ciclos for, 202  
 conflictos de nombres, 600  
 de archivo, 602  
 definir, 602  
 espacios de nombres, 609  
 global, 602  
 limitar, 455, 604  
 ocultar nombres fuera de, 612  
 referenciar objetos fuera de, 286  
 usar directiva, 610  
 variables, 108-111  
 externas, 604  
 visibilidad, 602  
**algoritmos de secuencia**, 708-709  
**alias**. Vea también referencias, 259  
 comandos/secuencias de comandos, 845-846  
 espacios de nombres, 613  
 para frases, 53  
**alinear llaves {}**, 84, 781  
**ambigüedad, resolución de (herencia múltiple)**, 426-427  
**ambulantes, apuntadores**, 247  
 comparados con apunadores nulos, 250  
 crear, 248-249  
 precauciones, 249  
**amigas (plantillas)**, 669  
 de tipo específico, 710  
 generales, 673-676, 710  
 que no son de plantillas, 669, 673  
**análisis**, 624  
 casos de uso, 626-633, 637  
 actores, 626-628  
 cajero automático, ejemplo, 629

- creación de paquetes, 637
- del dominio, 633
- diagrama, 636
- escenarios, 634
- lineamientos, 634
- mecanismos, 628
- modelos de dominio, 629
- resultados, 628
- UML, diagrama de interacción, 637
- de aplicación, 638
- de requerimientos, 626
- de sistemas, 638
- precauciones, 659
- analizar cadenas sintácticamente (listado), 253-255**
- ancho (salida), 578**
- AND (&), operador a nivel de bits, 775**
- AND (&&), operadores lógicos, 91-92**
- anidar**
  - ciclos (for), 200-201
  - espacios de nombres, 606
  - if, instrucciones, 86-90
  - paréntesis, 78-79
- ANSI (Instituto Estadounidense de Estándares Nacionales), 15-16**
  - C++, estándar, 15-16
  - espacios de nombres, 599
  - estándares, 15
  - excepciones, 717
- antropomorfismo (tarjetas CRC), 646-647**
- apagar bits, 776-777**
- apariencia de xxgdb, 825**
- APIs (GNOME), 905**
- aplicaciones. Vea también programas**
  - análisis, 638
  - ciclos principales de eventos, 925
  - conscientes de la sesión, 900
- crear**
  - KDE, 923-927
  - KDE, agregar botones, 928-931
  - KDE, agregar menús, 934-936
  - KDevelop, 937-938
  - wxStudio, 922
  - wxWindows, kit de herramientas, 910-918
  - wxWindows, kit de herramientas, agregar menús a la clase Window, 920-921
  - wxWindows, kit de herramientas, procesamiento de eventos, 919-920
  - enfoque, 918
  - GNOME, programación, 908
  - GTK++, 907
  - manejo de eventos (wxGTK), 918-919
  - marcos de trabajo, 903
  - por procedimientos, 897
  - X, clientes, 898
- aptrv (apuntador a función virtual), 356**
- apunta a, operador (->), 241**
- apuntadores, 225-227, 230, 235. Vea también referencias a cadenas, 391**
  - comparados con apunta-dores nulos, 250
  - crear, 248-249
  - precauciones, 249
  - a funciones, 463-466
  - a funciones miembro (listado), 478-479
  - acceso, 229
  - alcance, limitar, 455
  - almacenamiento, 232-233
  - aritmética, 253
  - arreglos de, 380-382
  - en comparación con apuntadores a arreglos, 382
  - nombres de, 382-384
- asignar, 237-238**
- como datos miembro**
  - heap, 242
  - listado, 243-244
- const, 250-253**
  - declarar, 250-251
  - listado, 251-252, 279-280
  - métodos, 251-252
  - pasar, 278-281
- const this, 253**
- constantes, 281**
- declarar, 228, 234, 285**
- descontrolados/ambulantes, 247, 250**
  - crear, 248-249
  - precauciones, 249
- desreferenciar, 229-230, 234**
- direcciones de memoria**
  - asignar, 228-229
  - examinar, 232-233
  - recuperar, 229-230
- eliminar, 237-238, 247**
  - arreglos, 470-472
  - asignar, 467
  - declarar, 464-466
  - desreferenciar, 470
  - listado, 464-468
  - pasar, 472-475
  - typedef, 475-477
  - ventajas, 467-470
- importancia de, 256**
- inicializar, 228, 234-236**
- instrucción, 130**
- listado, 389-390**
- manipulación de datos, 231-232**
- manipulación, ventajas, 234**
- memoria, propiedad del área de, 239**
- métodos, 477-480**
  - arreglos, 480-482
  - declarar, 478
  - invocar, 478, 480
- nombrar, 229**
- nulos, 228, 266**
- pasar por referencia, 268**
- perdidos, 228**
- pisotear, 249**

- propiedad, 290  
 reasignar, 239  
 referencias  
     combinar, 284  
     como alternativas, 281  
     comparación, 283-284,  
         291  
     regresar valores múltiples,  
         272-274  
     RTTI (Identificación de  
         Tipo en Tiempo de  
         Ejecución), 416  
 tablas v, 356  
 tamaños, 228  
 this, 246, 313-314  
     listado, 246-247, 313-314  
     métodos de acceso, 247  
 valores actuales, imprimir,  
     767-768  
 ventajas, 234  
**árboles, 398**  
**archivos**  
     abrir para entrada/salida,  
         585-587  
     agregar al final, 587-589  
     alcance, 602-604  
     bibliotecas, 133  
     binarios, 589-591  
     core, depurar, 741  
     de encabezado, 134  
         métodos, 271-272  
     de inicio, 849  
     de paquetes, manipular, 133  
     de proyecto, 134  
     de texto, 590-591  
         comparación con  
             archivos binarios,  
                 589-591  
         depurar, 741, 815  
         descriptores, 875  
         determinar formato, 818  
         ejecutables, 19  
         entrada, 585-589  
         espacios de nombres, 614  
         extensiones (compiladores  
             de GNU), 17  
         fuente, 17, 822  
         kernel, 741  
         make, 136, 820-821  
     modificar (RCS), 827-829  
     objeto, 19  
     printf(), 581  
     rastrear cambios, 828  
     salida, 585-589  
     tag, 815  
**arge (conteo de argumentos),**  
     592  
**argumentos, 37, 112, 837**  
     funciones, 101  
     línea de comandos, 592-595  
     pasar, 104  
         a constructores base,  
             342-346  
         apuntadores, 268  
         por referencia, 266-268,  
             275-278  
         por valor, 113-114, 142,  
             267-268  
         predeterminados, 116-118  
         señales y ranuras, 931  
**argv (vector de argumentos),**  
     592  
**aridad, 321**  
**aritmética de apuntadores, 253**  
**arreglos**  
     apuntadores, 380-382  
         a funciones, 470, 472  
         a funciones miembro  
             (listado), 480-48  
         a métodos, 480-482  
         en comparación con  
             arreglos de apunta-  
             dores, 382  
         nombres de arreglos,  
             382-384  
     bidimensionales, 377-379  
     char, 385-386  
     clases, 409-410  
     combinar, 411  
     consts y enums (listado),  
         374-375  
     contenedores vectoriales,  
         692  
     crear (listado), 383-384  
     de bolsa, 410  
     de diccionario, 409  
     de enteros (listado),  
         368-369  
     declarar, 367-368, eliminar  
         del heap, 384  
     dispersos, 410  
     elementos, 367-369  
         acceder, 375  
         cero, 373  
         no inicializados, 374, 411  
         numerar, 368  
         ordenar, 409-410  
     errores, 369  
     errores tipo poste de barda,  
         372  
     escribir más allá del final  
         de, 369-372  
     guardar  
         en el heap, 380-381  
         en la pila, 380  
     Inicializar, 373-374  
     integrados, 409  
     llenar, 386-387  
     memoria, 379  
     multidimensionales, 377  
         inicializar, 378-379  
         listado, 378-379  
     nombres (apuntadores),  
         382, 384  
     objetos, 375-377  
     pasar objetos a funciones,  
         668-669  
     plantillas  
         declarar, 662-664  
         implementar, 665-668  
     tamaños, 374-375  
     unidimensionales, 377  
**artefactos, 640**  
**ASCII, 56**  
     código de caracteres, 56  
     conjuntos de caracteres, 47  
     equivalentes numéricos  
         (tabla de), 958  
**asignación (=), operador, 51,**  
     68, 71  
     combinar con operadores  
         matemáticos, 74-75  
     confusión con operador  
         igual a (=), 80  
     contenedores vectoriales,  
         693  
     listado, 322-323  
     precedencia, 78

- asignadores, 692**  
**asignar.** *Vea también crear apuntadores, 237-238 apuntadores a funciones, 467 direcciones a referencias, 262-264 direcciones de memoria a apuntadores, 228-229 memoria, heap, 236 valores a variables, 51-52, 149-150 variables a clases definidas por el usuario, 324-325*
- AsignarEdad(), método, 242**
- AsignarPrimerNombre(), función, 507**
- asociación, 633**
- ASSERT(), macro, 758-761, 785**  
 código fuente, 759-760 depurar funciones, 760-762 excepciones, 760 limitaciones, 761 listado, 759
- asterisco (\*)**  
 operador de indirección, 229-230 sintaxis, 284-285
- AT& T UNIX System V, 879**
- atrapar excepciones, 722**
- autoasignados, operadores, 75**
- Ayuda, archivos (compilador g++), 39**
- AYUDA.CPP, demostración de los comentarios (listado), 35**
- B**
- barra diagonal inversa, carácter (\), 754**
- barra vertical (!), 91-92**
- barras diagonales (comentarios), 34, 41**
- barras diagonales inversas, dobles (\\"), 58**
- base (de números), establecer, 579**
- base 2, números, 951-952**  
 convertir  
     a decimales, 951  
     a números en base 10, 951  
     decimales a, 950  
     ventajas, 951
- base 7, números, 949**
- base 8, números, 948-949, 952-953**
- base 10, números, 948**  
 convertir a base 2, 950-951, 954  
 convertir a base 6, 949-950  
 convertir a base 7, 949  
 convertir a base 8, 953  
 subíndice, 948
- base 16, números, 953-956**
- base, clases, 334**  
 especializadas, 631  
 compartidas  
     funciones, 416  
     herencia, 427-431  
 comunes (listado), 428-430  
 métodos  
     llamar, 350-351  
     ocultar, 348-350  
     redefinir, 347-348
- base, constructores, 340**  
 pasar argumentos a, 342-346  
 sobrecargar, 342-346
- base, destructores, 340**
- bash, 836**  
 comandos  
     completación, 842  
     editar, 845  
     lista de historial, 844  
 establecer variables locales, 839  
 instrucciones de control, 846, 848  
 sustitución mediante comodines, 843
- BASIC, 10**
- Biblioteca de plantillas estándar. *Vea STL***
- sqrt(), función, 272**
- usar directivas, 614**
- bibliotecas, 558**  
 archivos de, 133  
 C++, de envoltura, 905  
 compartidas, 818-819  
 definición, 19  
 espacios de nombres, declarar, 604  
 estándar, uso de directivas, 22, 614  
 funciones, 41, 134, 137  
 GNOME, 905  
 iostream, 557  
 KDE, 938  
 libgnomeui, 908  
 múltiples, 599  
 POSIX, de subprocessos, 861  
 STL, 691-694, 698-701, 705  
 TrollTech, de gráficos de Qt, 899  
 wxGTK, 903, 909  
 wxWindows, kit de herramientas, 909-910
- binarios,**  
 archivos, en comparación con archivos de texto, 589-591  
 números. *Vea también números de base 2*  
 operadores, 320  
 semáforos, 868
- bits, 775, 951**  
 apagar, 776-777  
 cambiar, 777  
 campos, 777-780  
 encender, 776
- bloqueo de E/S, 875**
- bloques (instrucciones), 81**  
 alcance, 602, 610  
 catch, 716  
 múltiples especificaciones, 722-725  
 sintaxis, 721
- try, 716, 721**  
 excepciones sin errores, 739  
 sintaxis, 721
- variables, 110-111**

- Bonobo, 900**
- Booch, Grady, método, 623**
- Bool, tipo de datos, 79**
- booleanos, valores, 775**
- botones**
  - agregar a la clase de ventana de wxWindows, 913
  - agregar a la clase KDE Window, 928-931
- botones.cc, listado, 906-907**
- Bourne, shell (sh). Vea sh**
- Bourne, shell vuelto a nacer. Vea bash**
- break, instrucciones, 186-189**
  - do...while, ciclo, 193
  - listado, 187-188
  - precauciones, 189
  - switch, instrucciones, 207
- búferes**
  - administrar, 560
  - caracteres, 564
  - copiar cadenas, 387-390
  - fluxos, 558-560
  - implementar, 560-561
  - limpiar o vaciar, 560
  - manipular, 560
  - no inicializados, 385-386
- bugs o errores, 714. Vea también depuración**
  - apuntadores descontrolados, 247-249
  - arreglos, 369
  - compilar, 714
  - corrupción del código, 715
  - depurar, 739
    - archivos core, 741
    - costo de solucionar, 714
    - depuradores simbólicos, 739
    - ensambladores, 742
    - examinar memoria, 742
    - GNU, depurador, 740-741
    - puntos de interrupción, 742
    - puntos de observación, 742
  - distinguir entre, 714
  - excepciones, 716
- atrapar, 722
- funciones virtuales, 734
- múltiples, 722
- plantillas, 735-738
- sin errores, 738-739
- lógicos, 714
- minimizar, 167
- producir fragilidad, 714
- programación defensiva, 714
- sintáticos, 714
- solución de problemas
  - comentarios en el código, 716
  - excepciones, 720-721
- buscar direcciones de memoria, 226-227**
- bytes, 46, 951**
- C**
- C, APIs, 905**
- C-, comando, 813**
- C, compilador (gcc), 8**
- C, extensión de nombre de archivo, 167**
- C, lenguaje, 14**
- C, shell (csh). Vea csh**
- C/C++ Users Journal, 786**
- C++**
  - aprender, 15
  - apunta a (->), operador, 241
  - archivos de encabezado, 271
  - bibliotecas (paquete wxWindows), 909
  - bibliotecas de envoltura, 905
  - cadenas, 385
  - caja negra, método, 272
  - compilador (g++), 8
  - compilar programas, 817
  - constantes, 58, 283
  - contravarianza, 801
  - conversión descendente, 417
  - conversiones explícitas, 357
  - desarrollo, 75, 620
- elegir la herramienta adecuada, 190
- especificar el formato numérico de la salida, 962
- etiquetas, 182
- expresiones, 69
- funciones, 269
- GTK++, 904
- historia de, 9-10, 14
- implementar tipos de poder, 655
- limitar el alcance, 455
- manejo de excepciones, 715
  - catch, bloques, 716
  - ordenar, 728
  - try, bloques, 716
- notación húngara, 783
- operadores a nivel de bits, 775
- palabras reservadas, 50
- polimorfismo, 14, 352, 800
- programación orientada a objetos, 13, 619
  - encapsulación, 14
  - herencia, 14
- referencias, 262
- relaciones es un, 334, 352
- software comercial, 15
- switch, instrucciones, 205
- tipos de variables, 47
- tipos integrados, 306
- variables de conteo, 201
- ventajas, 10
- verdad/falsedad, 79
- C++ al descubierto, 379**
- C++ Report, 786**
- cabeza (nodos), 398**
- Cadena, clase, 391, 393-398, 505**
  - acceso, 508
  - constructores
    - de copia, 396
    - predeterminados, 396
  - declarar, 391, 393-398
  - main(), función, 397
- destructor, 396**
- listados, 392-395, 502-505**
- operadores, 397**
  - sobrecargados, 395

- usar instrucciones cout, 549
- cadena con terminador nulo, 391**
- cadenas**
- anализировать синтаксически (листинг), 253-255
  - caracteres нулов, 564
  - Char, массивы, 385
  - расположить между кавычками, 757
  - склеивать, 33, 757-758
  - контейнеры векторные, 698
  - копирование
    - strcpy(), 387-388
    - strncpy(), 388
  - создание, 507
  - дать формат, 581
  - из символов, анализ синтаксический (листинг), 253-255
  - текст, 31-33
  - стиль C, 544
  - функции общие (листинг), 387-388
  - функции член, 570
  - функции для, 139, 389
  - листинг, 389-390
  - манипуляция, 138, 756
  - вывод, символы специальные, 583
  - замены, 748, 843
  - токены, 748
  - использование, 757
  - значения текущие, вывести, 767-768
- каса черная, метод, 272**
- касирский автомат, пример, 629**
- вычислить размеры массивов, 373**
- callback, вопросы, 931**
- изменять**
- поведение предопределено потока, 587
  - константы, 59-60
  - биты, 777
- файлы (бит), 777-780**
- cannot find file, сообщения об ошибке, 22**
- caracteres, 45**
- &, 841
  - ASCII, набор, 47
  - бинарные, 564
  - строки текста, 31
  - комодины, 838
  - удаление, 58
  - заполнение, 578-579
  - EBCDIC, набор символов, 57
  - специальные
    - строки выхода, 583
    - печатать, 58
  - исследование, 570
  - функции для, 139
  - манипуляция, 138
  - имена переменных, 48
  - нулы, 385, 564
  - # (число), символ, 30
  - | (труба), 839
  - значения, 56
  - значение/буква, связь, 57
- загружать функции, 122**
- каскад, развитие, 623**
- case, значения (инструкции switch), 206**
- случаи использования, 626-633, 637**
- акторы, 626-628
  - автоматический кассир, пример, 629
  - клиенты, 627
  - создание
    - классы, 641
    - пакеты, 637
  - диаграмма, 636
  - домены
    - анализ, 633
    - модели, 629
  - сценарии, 634
    - пример, 635
    - направления, 634
  - механизмы, 628
  - результаты, 628
  - UML, диаграмма взаимодействия, 637
- catch, блоки, 716**
- многие спецификации, 722-725
  - синтаксис, 721
- категории, 146. Vea también tipos**
- CDE (Entorno Común de Escritorio), 899**
- CD-ROM (включено в этот книга), 9**
- центиградусы, конвертировать в Fahrenheit, 107-108**
- континенты (массивы), 372**
- ноль, элемент, 373**
- ноль вправо, развернуть, 579**
- cerr, объекты, 561, 596**
- закрыть дебаггер gdb, 824**
- C-h , команда (emacs), 813**
- char, массивы, 385-386**
- char, переменные, 45, 56**
- символы избегать, 58
  - кодификация символов, 57
  - размеры, 56
- цикла разработки (программирование), 19-20**
- основной цикл событий, 925**
- циклы, 181**
- do...while, 192-193
  - листинг, 192-193
  - синтаксис, 193
  - вечные. Vea циклы forever
    - листинг, 209-211
    - menu(), функция, 211
    - выход из, 209
    - switch, инструкция, 208-211
    - while, 189-190
  - externos, 190
- Fibonacci, применение в серии, 203-204**
- fib(), функция, 204
  - предупреждения, 205
- for, 195-196**
- доступ, 202
  - вложиться, 200-201
  - продвинутые, 196
  - инициализация, 195
  - инициализация множественная, 196-197
  - инструкции пустые, 197-200

- listado, 195-197  
 secuencia de ejecución, 196  
 vacíos, 198-200
- goto**, palabra reservada, 181-182  
 limitaciones, 183  
 recomendaciones, 183  
 sintaxis, 183
- listados**  
 condiciones de inicio, 194  
 saltar cuerpo de, 191  
 sintaxis, 184-185  
 while, 189-190
- listas enlazadas, 408  
 regresar al inicio de, 186  
 salir, 186  
 uso de, con la palabra reservada goto (listado), 182  
 while, 183-186  
 break, instrucciones, 186-189  
 complejos (listado), 185-186  
 continue, instrucción, 186-189  
 ejecutar, 191-192  
 expresiones complejas, 185-186  
 limitaciones, 191  
 listados, 184, 189-190, 194
- cin**  
 cadenas, terminar, 564  
 espacio en blanco, 564  
 tipos de datos, manejar, 563
- cin, objeto, 561-563**  
 entrada  
 cadenas, 564  
 múltiple, 564-567  
 operador de extracción, 567  
 métodos  
 get(), 567-571  
 getline(), 571-572  
 ignore(), 573-574  
 peek(), 574-575  
 putback(), 574-575
- clases, 19, 145-147, 155**  
 abstractas, 449  
 amigas, 534-535  
 declarar, 543  
 listado de programa de ejemplo, 535-542  
 sugerencias de uso, 543
- Arreglo, 409-410, 720**  
 arreglos, plantillas, 663-664  
 asignadores, 692  
 base, 334, 348  
 compartidas, 427-431  
 múltiples, 424
- C++, intenciones, 271**
- Cadena, 391-398, 501, 505, 553**  
 constructor de copia, 396  
 constructor predeterminado, 396  
 declarar, 391-398  
 destructor, 396  
 listado, 392-395  
 operadores, 397  
 sobrecargar, 395  
 capacidad, 436  
 compartir datos, 461  
 completas, declarar, 172-173  
 acceder, 508  
 constructores, 509-511  
 copiar por valor, 511, 515  
 costos, 508, 511  
 delegación, 516-531  
 implementar, 515-516  
 constantes, 762-767, 770-773  
 constructores, 159-160  
 predeterminados, 160, 298
- Contador**  
 declarar, 306-307  
 incremento, funciones, 307-308  
 contenedoras, 662, 702-704  
 contenedores vectoriales, 692-694, 698-699  
 contenidas, 501  
 crear, 641, 644  
 datos miembro, 146-147  
 acceder, 149, 152
- de otras clases, 171-175  
 privados, 151-152, 337-338  
 protegidos, 337-338  
 públicos, 151-153
- de algoritmos, 706-707  
 operaciones de secuencia mutantes, 708-709  
 operaciones de secuencia no mutantes, 707-708
- de arreglos, 720  
 parametrizadas, 662  
 plantillas, 663-664  
 usar en vez de arreglos integrados, 409
- de capacidad, 436  
 de figuras (listado), 437-439
- de flujos, 585
- de interfaz, 643-644  
 declarar, 146-147, 152, 155, 164-168, 543  
 definidas por el usuario, 324-325  
 definir, 149, 785
- deque, contenedores, 701  
 derivadas, 33-337, 436-439  
 constructores, 342-346  
 datos miembro, acceder, 338-339  
 declarar, 335, 337
- diseñar, 641-643
- Empleado, 505, 508
- enfoque del modelo estático, 644
- escribir  
 en archivos, 590-591  
 para guardar un objeto, 398
- especializadas, 631
- estructuras, comparación, 175
- excepciones, 720-721, 725
- fstream, 561**
- funcionalidad, 419
- funciones  
 miembro, sobrecargar, 293  
 modificar, 414

- Gato  
 declarar, 147, 169-170  
 implementar, 170-171
- gdb, comandos, 823
- herencia  
 conversión descendente, 416-419, 451  
 filtrar funciones comparadas, 416, 451  
 limitaciones, 413-416
- interfaz, 643
- ios, 561
- iostream, 561
- istream, 561
- jerarquía, 337, 436
- List, 600
- lista, 661  
 contenedores, 699-701
- listado de programa de ejemplo, 535-542
- Mamífero, 337
- métodos, 157-158, 168  
 constantes, 163-164  
 de acceso públicos, 153-154  
 definir, 150  
 en línea, 169-171  
 implementar, 156-159  
 modelo de diseño estático, 645  
 sobrecargar, 293-295  
 valores predeterminados, 296-298
- mezclas, 436
- nomenclatura, convenciones, 147-148
- objetos  
 comparar, 148  
 definir, 148, 155  
 inicializar, 300-301  
 referenciar, 264-265  
 tamaños, 177  
 valores, 149-150
- ostream, 561
- Perro, 335-337
- polimorfismo, 14
- privadas, 151-153
- Process, 855
- programación orientada a objetos, diseño, 646
- CRC, tarjetas, 647  
 relaciones, 648
- públicas, 151-153
- Punto, 174
- Qt, 932
- Rectángulo, 173-175
- resolución de nombres, 600-604
- responsabilidades, 645
- seguridad, 155
- streambuf, 560
- subclases, 171-175
- sugerencias de uso, 543
- superconjuntos, 334
- tipos de variables personalizados, 146
- UML, 622
- class, instrucciones, 155-156**
- class, palabra reservada, 147, 663**
- ClassGenerator, 937**
- clientes, 164, 627**  
 X, 898
- clog, objetos, 561, 596**
- clonación, código, 133, 137, 140-141**
- Clone(), método, 360, 363**
- cnt, variable estática, 608**
- COBOL, 10**
- código**  
 agrupar, 93  
 alias, 53  
 ambigüedad de nombres, 611  
 apuntadores, 253  
 apurar, 658  
 bugs o errores, 714  
 clonación, 133, 137, 140-141  
 comentarios, 34, 784  
 cuándo utilizar, 34  
 listado, 35  
 precauciones, 35  
 comodines, 838  
 compilación  
 compiladores, 9  
 errores, 24  
 símbolos, 739
- corrupción del, 715-716**
- crear fugas de memoria, 239
- depurar, 739  
 archivos core, 741  
 ensambladores, 742  
 examinar memoria, 742  
 GNU, depurador, 740-741  
 puntos de interrupción, 742  
 puntos de observación, 742
- editores, 809
- elegir la herramienta adecuada, 190
- errores, 714
- espacio en blanco, 68, 284
- espacios de nombres, 614
- expresiones, 69
- fuente, 11
- fuente abierto, 808  
 compilar, 18-19  
 programas ejecutables, 11  
 reutilización, 12-14
- funciones, 30
- guardias de inclusión, 752
- Hola, mundo, programa, 29
- intérpretes, 10
- legibilidad, 68
- lineamientos en el estilo, 780-781
- listados. *Vea* listados
- macros, 752
- manejo de excepciones, 716, 728
- notación húngara, 783
- # (numeral), símbolo, 30
- rellenar funciones, 335
- reutilizar, 133, 137, 140-141
- solución de problemas, 716
- Código Extendido de Caracteres Decimales**
- Codificados en Binario para el Intercambio de información, 57**
- CODIGO\_ERR, enumeración, 275**
- cola (nodos), 398**
- colas, 706**

- colecciones ordenadas (arreglos), 409**
- COLOR, variables de tipo, 61**
- comandos**
  - ifdef, 749
  - ifndef, 749
  - alias, 845-846
  - argumentos, 837
  - comodines, 838, 843
  - completación de, 842-843
  - Ctrl+Z, 841
  - editar, 845
  - ejecutar en segundo plano, 841
  - emacs, 813-814
  - gdb común, 740
  - gdb, depurador, 823-824
  - info, 137
  - ipcrm, 881
  - ipcs, 881
  - kill, 841
  - líneas de, 837
    - argumentos (listado), 592-595
    - concatenar, 837
    - desventajas, 896
    - GCC, opciones del compilador, 40
    - variables, 839
  - lista de historial, 844
  - make, 135
  - modo ex (editor vi), 811
  - RCS, 827
  - secuencias de comandos de shell, 846
  - shells, 836
- combinar**
  - arreglos, 411
  - operadores matemáticos con operadores de asignación, 74-75
  - referencias y apuntadores, 284
- comentarios, 34-35**
  - /< (estilo C), 34, 41
  - // (estilo C++), 34, 41
  - ayuda en la solución de problemas, 716
  - control de versiones, 827
- cuando utilizar, 34
- escribir, 41
- legibilidad, 784
- listado, 35
- precauciones, 35
- comillas**
  - caracteres de escape, 58
  - insertar, operador de cadena, 757
- comodines, 838, 843**
- comparar clases con objetos, 148**
- compiladores, 10-11, 18-19, 748**
  - almacenamiento, expandir, 694
  - arreglos, 368
  - compilar con símbolos, 739
  - conflictos de nombres, 600
  - constructores predeterminados, 298
  - contar, 372
  - descargar, 9
  - editores integrados, 25
  - enlazadores, 10
  - errores, 24, 167, 419, 600
  - errores en tiempo de compilación, 167
  - espacios de nombres, 599
  - evaluar instrucciones AND (&&), 92
  - formato intermedio, guardar, 748
  - gcc, 8, 816-817
  - GNU, 31, 808
  - GNU, g++, 8, 23, 817
    - opciones, 40
    - página del manual, 39
  - invocar, 18
  - macros, 758-761
  - objetos nulos, 266
  - operadores, 71
  - palabras reservadas, 50
  - preprocesadores, 30
  - RTTI, soporte, 419
  - soporte de excepciones, 717
  - soporte para plantillas, 791
- susceptibilidad al uso de mayúsculas y minúsculas, 49
- tablas v, 356
- this, apuntador, 247
- compilar**
  - bugs, 714
  - código fuente, 18-19
  - con pthreads, 861
  - de archivos fuente, 822
  - errores, 24, 167
  - g++, compilador, 817
  - gcc, compilador, 816-817
  - programas con referencias a objetos nulos, 289
  - símbolos, 739
- complemento, operador (-), 776**
- comportamientos predeterminados, 587**
- computadoras**
  - evolución, 11
  - interpretar números, 56
- con un solo enlace, listas, 398**
- concatenación, 837**
  - cadenas, 33, 757-758
  - operador de, 757-758
- conceptualización, 624-625**
- concordar (definiciones de funciones), 105**
- condición de carrera, 856**
- condicional, operador (>:), 94-95**
- condiciones de paro, 125**
- conflictos de nombres, 599-600**
- conflictos. Vea espacios de nombres**
- conjuntos (arreglos), 409**
- conscientes de la sesión, aplicaciones, 900**
- const, apuntadores, 250-253**
  - declarar, 250-251
  - listado, 279-280
  - métodos, 251-252
  - pasar, 278-281
- const, funciones miembro, 251-253**
- const, instrucción, 60, 163, 177, 786**

- const, métodos, 163-164**
- const, palabra reservada, 283**
- funciones miembro estáticas, 463
  - redefinir, 603
  - ventajas, 64
- constantes, 58, 762-767, 770-773**
- cambiar, 60
  - definir
    - const, 60
    - #define, 59
  - enumeradas, 60-62
  - demostración (listado), 61
  - listas enlazadas, 405
  - sintaxis, 60
  - valores, 61
- literales, 58-59
- r values, 71
- simbólicas, 59, 64
- sustituciones, 749
- Constantes(), método (listado), 762-766**
- constructores**
- base, pasar argumentos a, 342-346
  - clases contenidas, 509-511
  - de copia, 302-306
    - copias profundas, 302-303, 305-306
    - copias superficiales (de los datos miembro), 302
  - crear, 303
  - listado, 303-304
  - parámetros, 302
  - predeterminados, 302
  - virtuales, 360, 363
- declarar, 159, 329
- especializados, 686
- herencia, 340-342
- implementar, 163
- inicializar, 300
- invocar, 300
- llamar (listado), 340-342
- múltiples, 160, 424-426
- operadores de conversión, 325-328
- predeterminados, 160-163, 166
- sobrecargar, 299-300
  - clases derivadas, 342-345
  - listado, 299-300
- variables miembro, inicializar, 301
- Contador, clase**
- declarar, 306-307
  - incremento, funciones, 307-308
  - listado, 306-307
  - variables, 317
- contadores, crear, 326**
- contención, 501, 632**
- clases contenidas
    - acceder, 508
    - constructores, 509-511
    - copiar por valor, 511, 515
    - costos, 508, 511
    - delegación, 516-531
    - en comparación con la herencia múltiple, 651
    - herencia privada, 554
    - implementar, 515-516
- contenedores, 692**
- asociativos, 692, 701, 705
  - de secuencias, 692
    - deque, 701
    - envolturas, 705
    - lista, 699-701
    - map, 701, 704-705
    - vectoriales, 692-694, 698-699
- conteo**
- semáforos, 868
  - variables, 201
  - while, ciclos, 189
- continue, instrucciones, 186, 188-189**
- do...while, ciclo, 193
  - listado, 187-188
  - precauciones, 189
- contravarianza, 801-803**
- control de versiones, 827**
- convenciones**
- para nombrar clases, 147-148
- programación orientada a objetos, diseño, 622
- conversión descendente (herencia), 416-419, 451**
- conversiones**
- base 2 a base 10, 951
  - base 8 a base 10, 953
  - base 10 a base 2, 950-951
  - base 10 a base 6, 949
  - base 10 a base 7, 949
  - base 10 a base 8, 953
  - base, números, 950
  - decimales a binario, 954
  - explicitas, 357
  - Fahrenheit/centígrados, 107-108
- Convertir(), función, 107-108**
- copiar**
- cadenas
    - a búferes, 390
    - strcpy(), 387-388
    - strncpy(), 388
    - datos, 146
    - por valor, 511, 515
- copias profundas, 302-306, 322**
- copias superficiales (de los datos miembro), 302, 322**
- CORBA (Arquitectura de Intermediario de Solicituds de Objetos Comunes), 900**
- corchetes ([ ]), 384, 721**
- costo**
- contención, 508
  - memoria (métodos virtuales), 363
- cout, instrucciones, 32-34**
- en comparación con printf(), 581
  - listado, 32
  - pasar valores a, 33
  - quitar comentarios, 508
- cout, objeto, 31, 561, 575**
- métodos
    - fill(), 578-579
    - flush(), 575
    - put(), 575-576
    - setf(), 580-581
    - width(), 577-578
    - write(), 576-577

- cpp, archivos, 17, 167, 751**
- CPU, tiempo, 859**
- CRC, sesiones, 645**
  - limitación, 647
  - responsabilidades, 646
- CRC, tarjetas, 644-646**
  - antropomorfismo, 646-647
  - limitaciones, 647
  - traducir en UML, 648
- crear.** Vea también **asignar**
  - alias, 845-846
  - aplicaciones
    - KDE, 923-927
    - KDE, agregar botones, 928-931
    - KDE, agregar menús, 934-936
    - KDevelop, 937-938
    - wxStudio, 922
    - wxWindows, kit de herramientas, 910-917
    - wxWindows kit de herramientas, agregar menús a la clase Window, 920-921
    - wxWindows, kit de herramientas, comunicación de objetos, 918
    - wxWindows kit de herramientas, procesamiento de eventos, 919-920
  - apuntadores a funciones que son miembros de clases, 477
  - apuntadores descontrolados, 247-248
  - archivos de proyecto, 134
  - archivos make, 820
  - arreglos, 383-384
  - artefactos, 640
  - bibliotecas compartidas, 819
  - botones (aplicaciones wxWindows), 916
  - búferes de caracteres, 564
  - cadenas, 507
  - casos (análisis del diseño), 626
- clases
  - de arreglos parametrizadas, 662
  - excepciones, 725
  - manipulación de datos, 644
  - para guardar un objeto, 398
  - claves (System V IPC), 879-880
  - Clonet(), métodos, 360
  - colas de mensajes, 883-885
  - constructores de copia, 303
  - contadores, 326
  - ejecutables, 40
  - elementos, 693
  - Empleado, objetos, 511
  - espacios de nombres, 604
  - fugas de memoria, 239
  - GNOME, widgets, 907
  - KDE Window, clase, 926-927
  - listas enlazadas, 405, 487, 494-497
  - memoria compartida, 890
  - menús, 921
  - objetos, 873
  - objetos en el heap, 240-241
  - operadores de conversión, 325
  - paquetes, 637
  - procesos, 854-855
  - programas
    - buen diseño, 16
    - Hola, mundo, 23
  - puntos de observación, 741
  - Qt/KDE, aplicaciones GUI, 924
  - referencias, 260
  - representaciones, 146
  - semáforos, 887
  - subprocesos, 862-864
  - tipos, 146
  - tipos de variable, 60
  - tuberías
    - con nombre, 878
    - semidúplex, 877
  - variables, 48
  - variables múltiples, 51
- vectores, 694-697
- vi, etiquetas (tags), 815
- csh, 836**
- etags, 815**
- Ctrl+Z, comando, 841**
- cuadro de lista, widgets, 931**
- cuantos, 859**
- cubo, calcular volumen, 118**
- CXX, extensión de nombre de archivo, 17-18**
- Cygnus, sitio Web, 9**
- D**
- dados.c, programa (listado), 816**
- dar formato**
  - archivos, 818
  - archivos make, 136
  - cadenas, 581
  - caracteres especiales, 58
  - con printf(), 581
  - salida, 583-584
    - anchura, 577-578
    - indicadores, 580-581
- datos**
  - arreglos, 367
  - compartidos, 110
  - copiar, 146
  - estáticos, inicializar, 691
  - estructuras
    - colas, 706
    - pilas, 705
  - hacer referencia a, 146
  - heap, 235
  - manipular, 146, 231, 644
  - miembro. Vea datos miembro
    - de estado, 483
  - ocultar, 15
  - partición, 358-360
  - recuperar, 590
  - reserva, 456
  - tipos. Vea tipos de datos
  - validar, 62
    - declarar miembros, 691

- definir, 457  
funciones, plantillas, 687-690  
initializar, 691  
listado, 456-457
- datos miembro, 146**  
acceder, 149, 152  
de clases, 171-175  
estáticos, 455-458  
acceder, 458-461  
declarar, 687, 691  
definir, 457  
listado, 456-457  
ventajas, 483
- heap  
acceder, 241-242  
apuntadores, 242-244
- privados, 151-152, 337-338  
acceder, 151  
ventajas, 177
- protegidos, 337-338
- públicos, 151-152  
seguridad, 155  
initializar, 159
- de dirección, operador (&), 226-227, 261-262**
- de paro, condiciones, 125**
- decimal (ios::dec), 579**
- decimales, números, 948**  
convertir a base 2, 950-951, 954  
convertir a base 6, 949-950  
convertir a base 7, 949  
convertir a base 8, 953
- declarar, 149. Vea también initializar; definir**  
apuntadores, 228, 234  
a funciones, 464  
constantes, 250-251
- arreglos, 367-368, 374-375  
bidimensionales, 379  
en el heap, 381-382  
memoria, 379  
objetos, 375-376
- clases, 146-147, 152, 155, 168  
amigas, 543  
Cadena, 391-398  
completas, 172-173
- Contador, 306-307  
derivadas, 335-337  
errores, 164-167  
Gato, 147, 169-170  
Punto, 174  
Rectángulo, 173-175, 300  
subclases, 171, 173
- constructores, 159  
datos miembro estáticos, 687, 691  
elementos dentro de espacios de nombres, 600  
espacios de nombres, 604-606  
estructuras, 175-176
- Figura, 439  
funciones, 101-103  
de plantillas, 669, 673  
en línea, 122-124, 755-756  
Suma(), 317-318  
virtuales, 735
- herencia múltiple, 423  
herencia virtual, 435
- List, 600  
métodos  
amigos, 549  
apuntadores a, 478  
const, 163  
constructores, 329  
ubicaciones de archivos, 167-168  
valores predeterminados, 296-298
- operadores sobrecargados, 327  
plantillas, 662-664  
precauciones, 285  
referencias, 260-261, 265-266, 285
- tipos, 155  
tipos de datos abstractos, 441  
variables, 285  
variables miembro, 154
- decrementar, 75**
- decremento, operador (-), 75-77**
- default, instrucción, 207**
- define, instrucción, 59**  
listado, 749-750  
pruebas, 749  
sustituciones de cadenas, 748  
sustituciones de constantes, 749
- definir. Vea también inicializar**  
alcance (resolución de nombres), 602  
clases, 149, 785  
constantes  
#define, 59  
const, 60  
constantes simbólicas, 59  
datos miembro estáticos, 457  
espacios de nombres pobremente, 605  
funciones, 101, 104-105  
espacios de nombres, 607  
fuera de espacios de nombres, 606  
macros, 752-753  
métodos, ubicaciones de archivos, 167-168  
objetos, 148, 155  
plantillas, 662-665  
Semaphore, clase, 889  
SharedMemory, clase, 892  
sustituciones de cadenas, 748  
sustituciones de constantes, 749
- variables, 44, 49  
de tipo COLOR, 61  
locales, 106  
múltiples, 51  
nombres de, 48-49  
notación húngara, 50  
palabras reservadas, 50  
susceptibilidad al uso de mayúsculas y minúsculas, 49-50
- DEL, comando, 813**
- delegación, 516-531**
- delegar a una lista enlazada contenida (listado), 517-524**

- delete, instrucción.** 236-238  
**delete[]**, instrucción. 384  
**delete()**, operador. 693  
**delimitadores.** 387  
**depuradores simbólicos.** 739  
**depurar.** 739, 822-823.  
*Vea también* solución de problemas  
 archivos. 815  
 archivos core. 741  
 arreglos. 369-370  
**ASSERT()**, macro. 760-762  
 depuradores simbólicos. 739  
 ensambladores. 742  
 errores tipo poste de barda. 372  
 examinar memoria. 742  
 GNU, depurador. 740-741  
 GNU, depurador gdb. 822-823  
   ayuda. 823-824  
   cerrar. 824  
   comandos. 824  
   sesión de ejemplo. 826-827  
 goto, instrucciones. 183  
 guardias de inclusión. 751-752  
 herencia múltiple. 435  
 imprimir valores interinos. 767-768  
 macros predefinidas. 758  
 mezclas. 436  
 niveles. 768-774  
 puntos de interrupción. 742  
 puntos de observación. 742  
 reescrituras del sistema operativo. 370  
 ventajas. 744
- DEPURAR, modo**  
 niveles de depuración. 768-774  
 valores interinos. 767-768
- deque, contenedores.** 701
- derivación.** 335-337
- derivar ADTs de otros ADTs.** 445-448
- desarrollo**  
 CRC, sesiones. 645  
 entornos. 17
- desasignar memoria.** 236-238
- descargar**  
 compiladores. 9  
 KDevelop. 903
- describir**
- desplazamiento.** 369
- desplazamiento, operador [].** 395
- desplazarse a través de listas por medio de iteradores.** 700
- desplegar**  
 ceros a la derecha. 579  
 información gráfica. 898
- Desplegar(), método.** 494
- desreferencia, operador (\*)**, 229-230
- desreferenciar apuntadores.** 234
- destino, direcciones (referencias).** 261  
 asignar. 262-264  
 listado. 261  
 regresar. 261-262
- destino/dependencia, archivos make.** 136
- destructores.** 159, 161, 163  
 herencia. 340, 342  
 implementar. 163  
 limpiar memoria asignada. 244  
 llamar (listado). 340-342  
 predeterminados. 160-161, 163  
 virtuales. 360, 452
- determinar**  
 direcciones de memoria. 226-227  
 tamaños de variables. 45-46
- diagramas.** 653  
 de colaboración. 653  
 de secuencia. 657
- DibujarFigura(), función.** 293, 295
- diferente de, operador (!=).** 80
- dinámico, enlace.** 355
- direcciones**  
 de destino. 261-264  
 de memoria. 225, 230  
 determinar. 226-227  
 examinar. 232-233  
 guardar en apuntadores. 228-229  
 recuperar. 229-230  
 operadores. 226-227
- directivas**  
 usar. 609-611, 614  
 define. 748  
 include. 748
- directivas de sincronización**  
 mutex. 864-866  
 semáforos. 867  
 unir. 870  
 variables de condición. 868-869
- directivas (procesos).** 860
- discos, escribir en.** 558
- discriminadores.** 652-656
- diseño (orientado a objetos).** 624, 640  
 análisis de aplicación. 638  
 análisis de requerimientos. 626  
 análisis de sistemas. 638  
 artefactos. 640  
 asociación. 633  
 cascada. 623  
 casos de uso. 626  
   actores. 626-628  
   análisis de dominio. 633  
   cajero automático. ejemplo. 629  
   creación de paquetes. 637  
   diagrama. 636  
   escenarios. 634  
   lineamientos. 634  
   mecanismos. 628  
   modelos de dominio. 629  
   resultados. 628  
   UML, diagrama de interacción. 637

clases, 641  
 crear, 642-643  
 manipulación de datos, 644  
 contención, 632  
 CRC, sesiones, 645-647  
 discriminadores, 652-656  
 documentos, 641  
 documentos de planeación, 638-639  
 escenarios, 633  
 herencia múltiple en comparación con la contención, 651  
 iterativo, 623-624  
 controversia, 625  
 pasos, 624  
 lenguajes para modelar, 621-622  
 lineamientos, 634  
 lineamientos en el estilo del código, 780  
 lluvia de ideas, 627  
 metodologista, 623  
 modelo dinámico, 657  
 modelo estático, 644  
 precauciones, 658  
 proceso de, 622  
 prototipos, 639  
 tipos de poder, 652-656  
 transformaciones, 643-644  
 visión, 625  
 visualización, 639  
**DISPLAY, variable de entorno, 840**  
**dispositivos**  
 programación orientada a objetos, diseño, 644  
 redirigir, 561  
**división, 72-73**  
**do...while ciclos**  
 break, instrucciones, 193  
 ciclos while, comparación, 212  
 continue, instrucciones, 193  
 listado, 192-193  
 sintaxis, 193  
**Doble(), función, 124**

**documentación (+ Linux), 829-831**  
**documentos**  
 artefactos, 640  
 diseño, 641  
 modelos de dominio, 629  
 planear, 638 *Vea también diseñar*  
**documentos de planeación, 638-639**  
**dominio**  
 expertos, 626  
 modelos, 629, 649  
 objetos, 637, 643  
**DOS, comandos, 562**  
**dúplex total, tuberías, 876-877**  
**Duplicador(), función, 116**  
**dynamic\_cast, operador, 417**

## E

**EBCDIC, conjunto de caracteres, 57**  
**editar comandos, 845**  
**editores**  
 código, 809  
 de texto, 17-18  
 emacs, 812-814  
 comandos, 813  
 modificar archivos, 815  
 tutorial en línea, 813  
 seguros, 17  
 sin modo, 814  
**eficiencia (programas), 122**  
**ejecutables, 11**  
 crear, 19, 40  
 predeterminados, 818  
**ejecutar. Vea también llamar**  
 funciones, 105  
 programas, 36  
 switch, instrucciones, 207  
 while, ciclos, 184, 186  
**ejercicios. Vea ejercicios de repaso**  
**elegir la herramienta adecuada (C++), 190**

**elementos**  
 acceder, 694-697  
 arreglos, 367, 375  
 crear, 693  
 elemento cero, 373  
 liberar, 693  
**ELF (Formato ejecutable y de enlace), 818**  
**eliminar**  
 apuntadores, 237, 244, 250  
 creación de apuntadores descontrolados, 247  
 precauciones, 247  
 arreglos del heap, 384  
 IPC, objetos, 882  
 memoria compartida, 891  
 objetos, 873  
 objetos del heap, 240  
 restaurar memoria en el heap, 238  
**eliminar trabajos, 841**  
**else,**  
 comando del precompilador, 749-751  
 instrucciones, 89  
 palabra reservada, 85-86  
**emacs, 812**  
 comandos, 813-814  
 iniciar, 812  
 modificar archivos, 815  
 tutorial en línea, 813  
**Empleado, clase, 505-507**  
**en línea, funciones, 122-124, 755-756**  
 desventajas, 122  
 expandidas, 756  
 listado, 755-756  
**en línea, implementación, 169-171, 330**  
**encabezados (funciones), 104**  
**encapsulación, 13, 146, 558**  
**endl, instrucción, 33**  
**enfoque, 918**  
**enlace**

dinámico, 355  
 nombres, 603  
 en tiempo de ejecución, 355  
 externo, 603  
 interno, 603

- enlazadores, 10**  
 espacios de nombres, 600  
 falla, 600
- enlazar programas, 818**
- ensambladores, 742**  
 enteros long, 54  
 valores, límites de tamaño, 54-55
- enteros, 45**  
 arreglos, declarar, 368-369  
 especificación de tipo, 73  
 long, 45, 54, 63  
 operaciones de división, 72-73  
 short, 45, 54  
 signed, 46, 55-56  
 unsigned, 46, 54-55, 145
- Entorno Común de Escritorio. Vea CDE**
- Entorno de Desarrollo Integrado (IDE), 937**
- Entorno GNU de Modelo de Objetos de red. Vea GNOME**
- entornos o ambientes**  
 GNOME, 899  
 KDE, 901  
 variables de, 839-840
- entrada**  
 archivos, 585-589  
 ignorar, 573-574  
 manejar  
   cadenas, 564, 570-572  
   de un solo carácter, 567, 569  
   extracciones, 562-563, 567  
   múltiple, 565-567  
   objeto cin, 562  
   peek(), método, 574-575  
   putback(), método, 574-575
- enum, palabra reservada, 60**
- enviar mensajes (procesos), 882**
- envolturas (contenedores), 705**
- EOF (fin de archivo), 573**
- EOL (fin de línea), 573**
- errores**  
 apuntadores descontrolados, 249  
 cannot find file, 22  
**CODIGO\_ERR, enumeración, 275**  
 corrupción del código, 715  
 de compilación, 24, 167, 419, 440, 600, 606  
 declaraciones de clases, 164-167  
 referenciar objetos no existentes, 286-287
- errores tipo poste de barda, 372**
- E/S, objetos, 561**  
 cerr, 561  
 cin, 561-563  
   cadenas, 564  
   entrada múltiple, 564-567  
   get(), método, 567-571  
   getline(), método, 571-572  
   ignore(), método, 573-574  
   operador de extracción, 567  
   peek(), método, 574-575  
   putback(), método, 574-575  
 clog, 561  
 cout, 561, 575  
   fill(), método, 578-579  
   flush(), método, 575  
   put(), método, 575-576  
   setf(), método, 580-581  
   width(), método, 577-578  
   write(), método, 576-577
- ESC, comando, 813**
- escenarios, 633**  
 ejemplo, 635  
 lineamientos, 634
- escribir**  
 clases en archivos, 590-591  
 en discos, 558  
 en dispositivos de salida, 575-576
- en pantallas, 561  
 incremento, funciones, 307-308  
 más allá del final de arreglos, 369  
 palabras, múltiples, 564-565  
 prototipos, 101
- espacio de código, 130**
- espacio en blanco, 284**  
 cin, 564  
 ignorar, 570
- espacios de nombres, 599, 607-609**  
 agregar nuevos miembros, 606  
 alias, 613  
 ambigüedad de nombres, 611  
 anidar, 606  
 crear, 604  
 declarar  
   elementos dentro de, 600  
   tipos, 605  
 definir  
   funciones fuera de espacios de nombres, 606  
   funciones miembro, 607  
 extender múltiples archivos de encabezado, 604  
 globales, 613  
 identificar nombres dentro de, 611  
 listado, 607  
 pobemente definidos, 605  
 resolución de nombres, 600-604  
 sin nombre, 613-614  
 soporte del compilador, 599  
 std, 614-615  
 usar declaración, 611-613  
 usar directivas, 609-611, 614  
 variables locales, 610  
 Ventana, 607
- especificación de tipo para enteros, 73**
- especificadores de acceso, 338**
- especificadores de conversión, 581-582**

- establecer**  
 base de números, 579  
 bits, 776  
 indicadores, 579  
 variables (shells), 840
- estados, 657**  
 iostream, objetos, 579  
 superestados, 658
- estereotipos, 653**
- estructuras**  
 clases, comparación, 175  
 declarar, 175-176  
 sugerencias de uso, 178
- estructura de permisos (IPC), 881**
- estructuras dinámicas de datos, 494**
- etapa de inicialización, invocar constructores, 300-301**
- etiquetas, 182**
- etiquetas de acceso, 784-785**
- evaluar, 95. Vea también operadores**  
 AND (&&), instrucciones, 92  
 expresiones, 70  
 expresiones complejas, 70  
 if, instrucciones, 86
- ex, modo (editor vim), 811**
- examinar**  
 caracteres, 570  
 direcciones de memoria, 232-233  
 memoria, 742
- excepciones, 715-717**  
 ASSERT(), macro, 760-761  
 atrapar, 722  
 catch, bloques, 721  
 clases, 720-721  
 datos  
   leer, 728-732  
   pasar por referencia, 732-735  
 desventajas, 744  
 funciones virtuales, 732-735  
 jerarquías, 725-728  
 manejadores, ordenar, 728  
 múltiples, 722-725  
 NoHayMemoria, 732  
 plantillas, 735-738
- producir (listado), 717-720  
 recuperar datos de (listado), 728, 730-731  
 sin errores, 738-740  
 solución de problemas, 720-721  
   identificar problemas potenciales, 717  
   opciones, 718  
 soporte del compilador, 717  
 try, bloques, 716, 721  
 ventajas, 743  
 xl límite, 738
- exec(), llamada de sistema, 856**
- expandidas, funciones en línea, 756**
- expandir almacenamiento (compiladores), 694**
- expresiones, 70**  
 evaluar, 70  
 parentesis anidados, 78-79  
 ramificiar instrucciones  
   switch, 207  
   switch, instrucción, 205  
   verdad/falsedad, 79
- extracción, símbolo del operador de (>>), 562**
- F**
- facilitadores, 645**
- Factor(), función**  
 apuntadores, 273  
 referencias, 274-275
- Fahrenheit, convertir a centígrados, 107-108**
- falsedad, 79, 93, 951**
- FAQs (Preguntas frecuentes), 831**
- fib(), función, 204**
- Fibonacci, serie de, 124**  
 recursión, 126-127, 135  
 resolver por medio de la iteración, 203-204  
   fib(), función, 204  
   precauciones, 205
- solución del enésimo número de (listado), 203-204
- FIFO (Primero en Entrar, Primero en Salir), directiva de procesos, 860**  
 colas, 706  
 tuberías con nombre, 877
- fill(), método, 578-579**
- filtrar el acceso a las clases contenidas, 508**
- filtrar funciones compartidas, 416, 451**
- filtros de eventos, 928**
- fin de archivo (EOF), 573**
- fin de linea (EOL), 573**
- fluxos, 557**  
 buferes, 558-560  
 implementar, 560-561  
 limpiar o vaciar, 560  
 de clase stream, 585  
 abrir archivos para E/S, 585-587  
 comportamiento predefinido, 587-589  
 estados de condición, 585  
 encapsulación, 558  
 función printf(), comparación, 582-583  
 objetos de E/S estándar, 561  
   cerr, 561  
   cin, 561-575  
   clog, 561  
   cout, 561, 575-581  
 redirección, 561-562
- flush(), método, 575**
- for, ciclos, 195-196**  
 alcance, 202  
 anidar, 200-201  
 avanzados, 196  
 inicialización, 195-197  
 instrucciones  
   múltiples (listado), 196-197  
   nulas, 197-199  
   vacías (listado), 198-199
- listado, 195**
- secuencia de ejecución, 196**
- vacíos, 198-200**
- while, comparación, 212**

- extensiones de archivos, 17  
 formato intermedio, 748  
 soporte para plantillas, 791  
 susceptibilidad al uso de mayúsculas y minúsculas, 49  
 variables miembro  
   estáticas indefinidas, 457  
 declarar función main(), 31  
 depuradores, 740-741  
 documentación en línea, 39  
 editores de texto, 17  
 emacs, 812-814  
   comandos, 813  
   modificar archivos, 815  
   tutorial en línea, 81  
 enlazador, 601  
 gdb, depurador, 822-823  
   ayuda, 823-824  
   cerrar, 824  
   comandos, 824  
   sesión de ejemplo, 826-827  
 make, utilería, 820-822  
 sitio Web, 9
- goto, instrucciones, 182**  
 ciclos, 181-182  
 limitaciones, 183  
 recomendaciones, 183  
 sintaxis, 183
- grupos de noticias, 786**
- GTK++, 898**  
 aplicaciones, 907  
 funcionalidad, 918  
 GNOME, 904  
 GTK—, 901  
 LGPL, licencia, 904  
 recursos en línea, 901  
 wxGTK, biblioteca, 909
- GTK++ (Kit de herramientas Gimp), 900**
- guardar**  
 apuntadores, 232-233  
 arreglos  
   en el heap, 380-381  
   en la pila, 380
- compiladores, 694  
 datos (arreglos), 367  
 direcciones de memoria en apuntadores, 228-229  
 objetos (arreglos), 375
- guardias de inclusión, 751-752**
- GUIs (interfaces gráficas de usuario), 808**  
 elementos, 896  
 historia, 897  
 interactuar con programas, 918  
 Linux, 899  
 nativas, 910  
 programación  
   wxWindows, kit de herramientas, 909  
 señales, 918  
 widgets, 903-904
- H**
- h, extensión de nombre de archivo, 168**
- HacerTareaUno(), función, 211**
- hardware (CPUs), 130**
- heap, 235, 256**  
 arreglos  
   declarar, 381-382  
   eliminar, 384  
   guardar, 380-381  
 crear/eliminar objetos (listado), 240-241  
 datos, 235  
 datos miembro  
   acceder, 241-242  
   apuntadores, 242-244  
 memoria  
   asignar, 236  
   restaurar, 236-238  
 objetos  
   crear, 240  
   eliminar, 240-241  
   ventajas, 235-236
- HELLO.CPP, demostración de componentes de un programa (listado), 30**
- herencia, 14, 333-335  
 agregar a dos listas, 419-420  
 clases, 335  
 comparación con la generalización, 631  
 comparación con plantillas, 710  
 constructores, 340-342  
   argumentos, 342-346  
   sobrecargar, 342-346
- contención, 501  
 acceso, 508  
 constructores, 509-511  
 copiar por valor, 511, 515  
 costos, 508, 511  
 delegación, 516-531  
 filtrar el acceso, 508  
 implementar, 515-516
- conversión descendente, 416-419, 451  
 derivación, 334-335, 364  
 destructores, 340, 342  
 funciones, 416, 451  
 limitaciones, 413-416  
 listado, 336-337  
 métodos virtuales, 352-357, 431-435  
   apuntadores v, 356  
   constructores de copia, 360, 363  
   costos en memoria, 363  
   declarar, 435  
   destructores, 360  
   invocar múltiples, 353-355  
   listado, 432-434  
   partición, 358-360  
   tablas v, 356-357
- mezclas, 436  
 múltiple, 420-422, 650  
   clases base compartidas, 427-431  
   constructores, 424-426  
   declarar, 423  
   Java, 450  
   limitaciones, 435  
   listado, 420-422

- métodos virtuales.** 422  
**objetos.** 423  
**resolución de ambigüedad.** 426-427  
**privada.** 525-526  
 listado de programa de ejemplo, 526-533  
**métodos.** 525  
 sugerencias de uso, 534  
**simple (listado).** 414-415  
**UML.** 622  
**virtual.** 431, 435  
 declarar, 435  
 listado, 432-434  
**herramientas (GNU),** 9  
**historia de C++.** 9-10, 14-15  
**historial, lista de (comandos).** 844  
**Hola, mundo, programa,** 20  
 código fuente, 21, 29-30  
 compilar, 21-22  
 ejecutar, 22  
**GNU, g++,** 23  
 listado, 21  
**Hollerith, tarjetas de.** 57  
**HOME, variable de ambiente.** 840  
**HOWTOs.** 831  
**hp, extensión de nombre de archivo.** 168  
**hpp, extensión de nombre de archivo.** 168
- 
- IDE (Entorno de Desarrollo Integrado).** 937  
**identificadores**  
 nombrar, 783  
 ocultar, 601  
**IEEE (Instituto de Ingenieros Eléctricos y Electrónicos).** 808  
**If, instrucciones.** 81, 83, 105  
 anidar, 86, 88  
 listado, 87  
 llaves ({ }), 88-90
- errores para utilizar el comando  
 exit(), 86  
 errores para usar cout, 51-52  
 errores, 86  
 infinito, 86
- ifdef, comando del precompilador.** 749  
**ifndef, comando del precompilador.** 749  
**ifstream, objetos.** 585-587  
 ignorar espacio en blanco, 570  
**ignore(), función.** 573-596  
 igual, signo (=), 51, 71,  
 321-324  
**imitar, RTTI (Identificación de Tipo en Tiempo de Ejecución).** 416  
**implementar,** 624  
 búferes, 560-561  
 clases, 170-171  
 contención, 515-516  
 flujos, 560-561  
 funciones virtuales puras,  
 441-445  
 intercambiar(), función  
 apuntadores, 268-269  
 referencias, 269-271  
 métodos, 156-159  
 const, 164  
 constructores, 163  
 de clases, 157-158, 168  
 de plantilla, 673  
 destructores, 163  
 en linea, 169-171, 330  
 plantillas, 666-668, 682-686  
 plantillas, arreglos, 665-667  
 subprocesos, 861  
 tipos de poder, 655
- imprimir**  
 cadenas de texto, 33  
 caracteres con base en los números (listado), 57  
 caracteres de impresión especiales, 58  
 en pantalla, 31-34  
 printf(), función, 581-583  
 valores en modo DEPURAR (listado), 767-768  
 valores interinos, 767-768
- ImprimirError(), función.** 735  
**include, archivos.** 785  
**include, instrucciones.** 40, 41,  
 561  
**incrementar,** 75  
**incremento, funciones.** 307, 348  
**incremento, operador (+ +),** 75  
 incremento de memoria, 82  
 incremento de memoria, 82  
 incremento de memoria, 82
- independientes de la plata- forma.** 557
- indicadores.** 775, 836  
 de estado, 811-812  
 establecer, 817-881  
 manipulación de texto, 178  
 setios, 884  
 setios, manipulación, 884  
 setios, manipulación, 884
- indice, operador [ ].** 375
- indices.** 692
- indireccion, operador (\*).** 229-230, 284-285
- info, comando.** 137
- initialización, instrucción de.** 195
- initializar,** 159, 261. *Vea también declarar*  
 apuntadores, 228, 234-236  
 arreglos, 373-374, 378-379  
 clases base, 342  
 constructores, 300  
 datos estáticos, 691  
 datos miembro, 159  
 for, ciclos, 195-197  
 funciones virtuales  
 con cero, 440  
 objetos, 300-301  
 Gato, 161-163  
 métodos constructores, 159  
 referencias, 261  
 variables, 51, 109  
 locales, 111  
 miembro, 301
- iniciar**  
 emacs, 812  
 vi, editor, 809-810

- for\_each()**, algoritmo, 707-708  
**fork()**, función, 854  
**Formato Ejecutable y de Enlace (ELF)**, 818  
**formato intermedio (compilador)**, 748  
**FORTRAN**, 201  
**friend**, palabra reservada, 549  
**fstream**, clases, 561  
**fuerte tipificación**, 164  
**fugas (memoria)**, 236, 287-289  
  delete, instrucción, 236  
  reasignación de apuntadores, 239  
**FUNC.CPP, demostración de una función simple (listado)**, 38  
**funciones**, 19, 30, 37-39. *Vea también macros; métodos*  
  abstracción, 129  
  amigas, 544, 548  
  amigas que no son de plantillas, 669-672  
  apuntadores, 463-466,  
    arreglos, 470, 472  
    asignar, 467  
    declarar, 464-466  
    desreferenciar, 470  
    listado, 464-468  
    pasar, 472-475  
    typedef, 475-477  
    ventajas, 467-470  
  archivos de encabezado, 271-272  
  argumentos, 101, 112  
    pasar, 104  
    pasar por referencia, 266-268, 275-278  
    pasar por valor, 113-114, 142, 267-268  
  predeterminados, 116-118  
  arreglos, objetos, 668-669  
**AsignarPrimerNombre()**, 507  
**ASSERT()**, 785  
  bibliotecas, 41, 134, 137  
**C++**, 269-271  
  cadenas, 139  
  listado, 387-388  
  nombres de arreglos, 389  
  caracteres, 139  
  cargar, 122  
  comunes, 140  
  const, funciones miembro, 163  
    declarar, 163  
    implementar, 164  
    ventajas, 164  
  control de procesos, 856  
**Convertir()**, 107-108  
  cuerpo, 37  
  de acceso, 153-154, 247  
  de encabezados, 104  
  de plantilla  
    declarar, 669, 673  
    especializada, 681-682, 686-687  
    implementación, 673  
  de plantilla generales, 673-676, 710  
  declarar, 101  
    listado, 103  
    ubicaciones de archivos, 167-168  
  definidas por el usuario, 100  
  definir, 104, 111, 150, 168  
    concordar, 105  
    en orden, 101  
    fuera de espacios de nombres, 606  
    prototipos, 102  
**DibujarFigura()**, 293-295  
**Doble()**, 124  
**Duplicador()**, 116  
  ejecutar, 105  
  en línea, 122-124, 755-756  
    desventajas, 122  
    expandidas, 756  
    listado, 755-756  
  espacios de nombres, 600, 606  
    especializadas, 681-682, 686-687  
  especificadores de acceso, 338  
**Factor()**  
  apuntadores, 273  
  referencias, 274-275  
**fib()**, 204  
  declarar, 549  
  sobrecargar operadores, 544-548  
**FuncionLlenarInt()**, 681  
**FuncionUno()**, 278  
  generales, 139  
**GetString()**, 397  
**HacerTareaUno()**, 211  
**herencia**  
  conversión descendente, 416-419, 451  
  filtrar funciones comparadas, 416, 451  
**ImprimirError()**, 735  
  incremento, 307-308  
**Insertar()**, 534  
  instrucciones, 111-112  
  integradas, 100, 137  
  intercambiar(), 113, 267  
    apuntadores, 268-269  
    referencias, 269-271  
**Intrusos()**, 669  
  invocar, 36, 129, 133  
**IPC**, 879-880  
  listado, 38  
**llamar**, 36, 100, 111  
  listado, 36  
  pila, llamadas, 722  
  macros, 752-755  
  main(), 30, 99  
  matemáticas, 137-138  
**Maullar()**, 154, 158  
**menu()**, 211  
**miembro**  
  const, palabras reservada, 251  
  getline(), 570  
**miembro estáticas**, 461-462, 687, 690-691  
  acceder, 463  
  listado, 461-462  
  llamar, 461-463  
  modificar, 414  
**MostrarTodo()**, 533  
  multiclas, 414

nivel de abstracción, 449  
**ObtenerArea()**, 174  
**ObtenerEdad()**, 158  
**ObtenerEstado()**, 779  
**ObtenerPeso()**, 169  
**ObtenerSupIzq()**, 174  
 parámetros de, 104, 112  
 polimorfismo, 14,  
   119-122  
**printf()**, 582-583  
 prototipos, 101-104,  
   271-272  
 nombres de parámetros,  
   103  
 tipos de valor de retorno,  
   102, 105  
 ramificar, 105  
 ranuras, 932  
 recursión, 124-128  
 redefinir, 346-348  
 regresar múltiples valores  
   apuntadores, 272-274  
   referencias, 274-275  
 llenar, 335  
 resolución de nombres,  
   600-604  
 retorno, valores de, 100  
 sintaxis, 31, 37  
 sistemas operativos, 562  
**sizeof()**, 46  
 sobrecargar, 119-120,  
   293-295, 348  
**sqr()**, 272  
**streat()**, 388  
**strcpy()**, 387-388, 391  
**strncpy()**, 388  
**Suma()**, 38, 317-318  
 tamaños, 112  
 tubería, 874  
 valores  
   regresar, 37, 114  
   void, 37  
 virtuales, 365, 452, 801  
   destructores, 452  
   llamar múltiples (listado),  
     354-355  
   puras, 440-445  
 void, valores, 114  
**VolumenCubo()**, 118

funciones miembro. *Vea también métodos:*  
   apuntadores a (listado),  
     478-479  
   const palabra reservada,  
     251  
   getline(), 570  
   sobrecargar, 293-295  
**FuncionI.llenarInt()**, función,  
   681  
**FuncionUno()**, función, 278

**G**

**g++, compiladores**, 8, 23,  
   817  
   opciones, 40  
   página del manual, 39

**Gato, clase**  
   datos miembro, acceder a,  
     152  
   declarar, 147, 169-170  
   funciones de acceso, 163  
   implementar, 170-171  
   listados, 169-171  
   métodos  
     AsignarEdad(), 158  
     de acceso, 154  
     implementar, 156-158  
     Maullar(), 154, 159  
     ObtenerEdad(), 158  
     ObtenerPeso(), 169  
     objeto, inicializar, 161-163

**GATO, constructor**, 302

**gcc, compilador**, 8, 816-817

**gdb, comandos**, 740

**gdb, depurador**, 822-823  
   ayuda, 823-824  
   cerrar, 824  
   comandos, 824  
   sesión de ejemplo, 826-827  
   xxgdb, 825

**GDI (Interfaz de Dispositivo Gráfico)**, 898

**GDK++, 905**

**generales, funciones**, 139

**generalización**, 631, 649

**get(), método**, 386, 567, 569  
   arreglos de caracteres,  
     570-571  
   con/sin parámetros,  
     568-569  
   parámetros de referencia de  
     caracteres, 569  
   sobrecargar, 572

**getline(), función**, 570-572

**GetString(), función**, 397

**Gimp, Kit de herramientas (GTK++)**, 900

**globales, variables**, 108-109,  
   130  
   limitaciones, 109-110, 141  
   listado, 108-109  
   ocultar, 108

**GNOME (Entorno GNU de Modelo de Objetos de Red)**, 899  
   API, 903, 905  
   biblioteca, 905  
   Bonobo, 900  
   CORBA, 900  
   GDK++, 904-905, 907  
   GTK++, 904  
   instalar, 900  
   listado básico de programa  
     (botones.cc), 906-907  
   programación, 904, 908  
   recursos en línea, 900  
   sitio Web, 900  
   widgets, crear, 907  
   wxWindows, kit de herramientas, 909

**GnomeApp, marco de trabajo de aplicación**, 908

**GNOMEHelloWorld, programa**  
   agregar  
     botones, 913-917  
     menús (listado), 921  
   comunicación de objetos,  
     918  
   listado (básico), 910

**GNU (GNU No es UNIX)**, 7  
   compiladores, 808  
     compilador g++, 39-40  
   errores, 419

- extensiones de archivos, 17  
 formato intermedio, 748  
 soporte para plantillas, 791  
 susceptibilidad al uso de mayúsculas y minúsculas, 49  
 variables miembro estáticas indefinidas, 457  
 declarar función main(), 31  
 depuradores, 740-741  
 documentación en línea, 39  
 editores de texto, 17  
 emacs, 812-814  
   comandos, 813  
   modificar archivos, 815  
   tutorial en línea, 81  
 enlazador, 601  
 gdb, depurador, 822-823  
   ayuda, 823-824  
   cerrar, 824  
   comandos, 824  
   sesión de ejemplo, 826-827  
 make, utilería, 820-822  
 sitio Web, 9
- goto, instrucciones, 182**  
 ciclos, 181-182  
 limitaciones, 183  
 recomendaciones, 183  
 sintaxis, 183
- grupos de noticias, 786**
- GTK++, 898**  
 aplicaciones, 907  
 funcionalidad, 918  
 GNOME, 904  
 GTK—, 901  
 LGPL, licencia, 904  
 recursos en línea, 901  
 wxGTK, biblioteca, 909
- GTK++ (Kit de herramientas Gimp), 900**
- guardar**  
 apuntadores, 232-233  
 arreglos  
   en el heap, 380-381  
   en la pila, 380
- compiladores, 694  
 datos (arreglos), 367  
 direcciones de memoria en apuntadores, 228-229  
 objetos (arreglos), 375
- guardias de inclusión, 751-752**
- GUIs (interfaces gráficas de usuario), 808**  
 elementos, 896  
 historia, 897  
 interactuar con programas, 918  
 Linux, 899  
 nativas, 910  
 programación  
   wxWindows, kit de herramientas, 909  
 señales, 918  
 widgets, 903-904
- H**
- h, extensión de nombre de archivo, 168**
- HacerTareaUno(), función, 211**
- hardware (CPUs), 130**
- heap, 235, 256**  
 arreglos  
   declarar, 381-382  
   eliminar, 384  
   guardar, 380-381  
 crear/eliminar objetos (listado), 240-241  
 datos, 235  
 datos miembro  
   acceder, 241-242  
   apuntadores, 242-244  
 memoria  
   asignar, 236  
   restaurar, 236-238  
 objetos  
   crear, 240  
   eliminar, 240-241  
   ventajas, 235-236
- HELLO.CPP, demostración de componentes de un programa (listado), 30**
- herencia, 14, 333-335**  
 agregar a dos listas, 419-420  
 clases, 335  
 comparación con la generalización, 631  
 comparación con plantillas, 710  
 constructores, 340-342  
   argumentos, 342-346  
   sobrecargar, 342-346  
 contención, 501  
   acceso, 508  
   constructores, 509-511  
   copiar por valor, 511, 515  
   costos, 508, 511  
   delegación, 516-531  
   filtrar el acceso, 508  
   implementar, 515-516  
 conversión descendente, 416-419, 451  
 derivación, 334-335, 364  
 destructores, 340, 342  
 funciones, 416, 451  
 limitaciones, 413-416  
 listado, 336-337  
 métodos virtuales, 352-357, 431-435  
   apuntadores v, 356  
   constructores de copia, 360, 363  
   costos en memoria, 363  
   declarar, 435  
   destructores, 360  
   invocar múltiples, 353-355  
   listado, 432-434  
   partición, 358-360  
   tablas v, 356-357
- mezclas, 436  
 múltiple, 420-422, 650  
   clases base compartidas, 427-431  
   constructores, 424-426  
   declarar, 423  
 Java, 450  
 limitaciones, 435  
 listado, 420-422

- métodos virtuales, 422  
 objetos, 423  
 resolución de  
     ambigüedad, 426-427  
 privada, 525-526  
     listado de programa de  
         ejemplo, 526-533  
     métodos, 525  
     sugerencias de uso, 534  
 simple (listado), 414-415  
 UML, 622  
 virtual, 431, 435  
     declarar, 435  
     listado, 432-434
- herramientas (GNU), 9**  
**historia de C++, 9-10, 14-15**  
**historial, lista de (comandos), 844**
- Hola, mundo, programa, 20**  
     código fuente, 21, 29-30  
     compilar, 21-22  
     ejecutar, 22  
     GNU, g++, 23  
     listado, 21
- Hollerith, tarjetas de, 57**
- HOME , variable de ambiente, 840**
- HOWTOs, 831**
- hp, extensión de nombre de archivo, 168**
- hpp, extensión de nombre de archivo, 168**
- I**
- IDE (Entorno de Desarrollo Integrado), 937**
- identificadores**  
     nombrar, 783  
     ocultar, 601
- IEEE (Instituto de Ingenieros Eléctricos y Electrónicos), 808**
- If, instrucciones, 81, 83, 105**  
     anidar, 86, 88  
     listado, 87  
     llaves ({ }), 88-90
- else, palabra reservada, 85-86  
 estilos de sangría, 84-85  
 evaluar, 86  
 sintaxis, 86
- ifdef, comando del precompilador, 749**
- ifndef, comando del precompilador, 749**
- ifstream, objetos, 585-587**
- ignorar espacio en blanco, 570**
- ignore(), función, 573-596**
- igual, signo (=), 51, 71, 321-324**
- imitar, RTTI (Identificación de Tipo en Tiempo de Ejecución), 416**
- implementar, 624**  
     búferes, 560-561  
     clases, 170-171  
     contención, 515-516  
     flujos, 560-561  
     funciones virtuales puras, 441-445  
     intercambiar(), función  
         apuntadores, 268-269  
         referencias, 269-271  
     métodos, 156-159  
         const, 164  
         constructores, 163  
         de clases, 157-158, 168  
         de plantilla, 673  
         destructores, 163  
         en línea, 169-171, 330  
     plantillas, 666-668, 682-686  
     plantillas, arreglos, 665-667  
     subprocesos, 861  
     tipos de poder, 655
- imprimir**  
     cadenas de texto, 33  
     caracteres con base en los números (listado), 57  
     caracteres de impresión especiales, 58  
     en pantalla, 31-34  
     printf(), función, 581-583  
     valores en modo DEPURAR (listado), 767-768  
     valores interinos, 767-768
- ImprimirError(), función, 735**
- include, archivos, 785**
- include, instrucciones, 30, 41, 561**
- incrementar, 75**
- incremento, funciones, 307-308**
- incremento, operador (++), 75**  
     agregar, 307-308  
     postfijo, 75-77  
     prefijo, 75-77
- independientes de la plataforma, 557**
- indicadores, 775, 836**  
     de estado, 577-579  
     establecer, 579-581  
     manipulación de bits, 775  
     resetiosflags, manipuladores, 584  
     setiosflags, manipuladores, 584
- índice, operador ([ ]), 375**
- índices, 692**
- indirección, operador (<), 229-230, 284-285**
- info, comando, 137**
- inicialización, instrucción de, 195**
- inicializar, 159, 261. Vea también declarar**  
     apuntadores, 228, 234-236  
     arreglos, 373-374, 378-379  
     clases base, 342  
     constructores, 300  
     datos estáticos, 691  
     datos miembro, 159  
     for, ciclos, 195-197  
     funciones virtuales  
         con cero, 440  
     objetos, 300-301  
         Gato, 161-163  
         métodos constructores, 159  
     referencias, 261  
     variables, 51, 109  
         locales, 111  
         miembro, 301
- iniciar**  
     emacs, 812  
     vi, editor, 809-810

- inline, instrucción, 122, 142, 169**
- inserción, operador (<<), 32**
- contenedores vectoriales, 698
  - sobrecargar, 549, 553
- Insertar(), función, 496, 534**
- insertar, modo (editor vi), 810**
- instalar**
- GNOME, 900
  - KDE, 901
  - Linux, 8
- instancias (plantillas), 662**
- instrucciones, 67**
- apuntadores, 130
  - bloques, 81
    - catch, 716, 721
    - try, 716, 721
  - break, 186-189
    - do...while, ciclo, 193
    - listado, 187-188
    - precauciones, 189
  - catch, 716, 721-725
  - class, 147, 155-156, 663
  - const, 60, 163, 177, 786
  - continue, 186-189
    - do...while, ciclo, 193
    - listado, 187-188
    - precauciones, 189
  - cout, 32-34
  - de control (bash), 846-848
  - de retorno múltiples (listado), 115-116
  - default, 207
  - define, 59
    - listado, 749-750
    - pruebas, 749
    - sustituciones de cadenas, 748
  - sustituciones de constantes, 749
  - delete, 236-238
  - delete[] , 384
  - do...while, 193
  - else, 749-751
  - endl, 33
  - enunciados de visión, 626
  - expresiones, 70
- for, 195-196
  - ciclos avanzados, 196
  - sintaxis, 195
- friend, 549
- funciones, 111
- funciones, prototipos de, 102
- goto, 182
  - ciclos, 181-182
  - limitaciones, 183
  - recomendaciones, 183
  - sintaxis, 183
- if, 81-83, 105
  - anidar, 86-90
  - else, palabra reservada, 85-86
  - estilos de sangría, 84-85
  - sintaxis, 86
- if complejas, 86
- include, 30, 561
- initialización, 195
- inline, 122, 142, 169
- new, 236
- null, 67
- operator, 317
- paréntesis, 78-79
- precedencia, 77
- protected, 337
- return, 37, 114-116
- struct, 175-176
- switch, 205-207, 451
  - ciclos eternos, 208-211
  - listado, 206-207
  - sintaxis, 205-208
  - sugerencias de uso, 211
  - valores de case, 206
- template, 663
- try, 716, 721
- typedef, 475-477
- verdad/falsedad, 93
- watch, 788
- while
  - ciclos, 183-184
  - limitaciones, 191
  - sintaxis, 184-185
- int main(), 116**
- interacción, diagrama (UML), 637**
- interactivos, shells, 849**
- intercambiar(), función, 113, 267**
- apuntadores, 268-269
  - encabezados y prototipos, 271
  - reescrita con referencias (listado), 270
  - referencias, 269-271
- interfaces**
- Java, 450
  - programación orientada a objetos, diseño, 643
- Interfaz de Dispositivo Gráfico (GDI), 898**
- interfaz gráfica de usuario.**
- Vea GUIs*
- intérpretes, 10**
- intérpretes en tiempo de ejecución, 10**
- interprocesos, comunicación entre.**
- Vea IPC*
- Intruso(), función, 669**
- invocar**
- apuntadores a métodos, 478-480
  - compiladores, 18
  - funciones, 36, 129, 133
  - métodos
    - estáticos, 461-463
    - de la base, 350-351
- ios, 587**
- ios, clase, 561**
- ios::dec (decimal, base 10), 579**
- ios::hex (hexadecimal, base diecisésis), 579**
- ios::oct (octal, base ocho), 579**
- iostream, biblioteca, 557**
- iostream, clase, 561**
- iostream, objetos, 579**
- IPC (comunicación entre procesos), 873**
- eliminar objetos, 882
  - estructura de permisos, 881
  - funciones, 879-880
  - limpiar recursos, 881
  - memoria compartida, 890
    - crear, 890
    - definir clase, 892
    - eliminar, 891

**semáforos**, 886  
  crear, 887  
  definir clase semáforo, 889  
**System V**, crear claves, 879-880  
  ver estado de objetos, 881  
**ipc()**, llamada de sistema, 892  
**ipc\_perm**, estructura (listado), 881  
**iperm**, comando, 881  
**ipes**, comando, 881  
**ISO (Organización Internacional de Estándares)**, estándar, 15  
**Istream**, clase, 561  
**iteración (ciclos)**, 181  
  alcance, 202  
  anidar, 200-201  
  do...while, 192-193  
  Fibonacci, aplicación de la serie de, 203-205  
  for, 194-200  
  goto, palabra reservada, 181-183  
  recursión, comparación, 212  
  while, 183-192  
**iteradores**, 699-700  
**iteradores no constantes**, 700  
**iterativo, diseño**, 623-624  
  controversia, 625  
  pasos, 624

**J**

**Jacobson, Ivar**, 623  
**Java**, 10, 450  
**jerarquías**  
  clases, 337, 436  
  excepciones, 725-728  
  herencia, 333

**K**

**Kapplication**, constructor, 927  
**KAppWizard**, 937

**KB (kilobytes)**, 952  
**KDE (Entorno de Escritorio K)**, 899, 923  
  API, 903  
  bibliotecas, 938  
  habilidades, 902  
  instalar, 901  
  KDEHelloWorld, programa, 924-927  
  KDevelop, 937-938  
  Qt, biblioteca, 903  
  recursos en línea, 903  
  Window, clase, 926-927  
    agregar botones a, 928-931  
    agregar menús a, 934-936  
  wxWindows, kit de herramientas, 909  
**KDEHelloWorld**, programa  
  agregar botones, 928-931  
  agregar menús, 934-936  
  con clase Window derivada (listado), 926  
  listado, 924  
**KDevelop**, 937  
  características, 937  
  descargar, 903  
  KDE, bibliotecas, 938  
**Key**, clase (listado), 880  
**key, parámetro**, 884  
**KFM, biblioteca**, 938  
**KHTMLW, biblioteca**, 938  
**kill, comando**, 841  
**kilobytes (KB)**, 952  
**Korn, shell (ksh)**, 836  
**kprocess, contexto**, 854  
**ksh**, 836  
**Kspell, biblioteca**, 938  
**KTMainWindow, clase**, 926

**L**

**l-values**. *Vea valores leer datos en excepciones*, 728-732  
**legibilidad del código**, 782

**lenguajes**  
  de programación, 62  
  independientes de la plataforma, 887  
  para modelar, 621  
  uso de secuencias de comandos, 818  
**Less Fif**, 898  
**LGPL, licencia**, 904  
**liberar**  
  elementos, 693  
  memoria, 236-238  
**LIFO, estructuras**, 705  
**limitaciones (variables globales)**, 109-110  
**limitar alcance de variables y de apuntadores**, 455  
**limpiar o vaciar**  
  bufferes, 560  
  salida, 575  
**lineamientos**, 634  
**lineamientos de estilo (código)**, 780  
  alineación de llaves, 781  
  comentarios, 784  
  definiciones de clases, 785  
  etiquetas de acceso, 784-785  
  include, archivos, 785  
  legibilidad del código, 782  
  líneas largas, 781  
  ortografía, 783-784  
  sangría, 781-782  
  uso de mayúsculas, 783-784

**Linux**, 8  
  /proc, sistema de archivos, 858  
  acceder información del sistema, 858  
  apariencia del escritorio, 896  
  ayuda en línea, 811-812  
  comandos, 8  
  control de procesos, 842, 856, 858  
  documentación en línea en línea, 39, 831  
    HOWTOs y FAQs, 831  
  editores de código  
    emacs, 812-815  
    vi, 809-810  
    vim, 809

- GNOME.** 899  
 API, 905  
 GDK++, 904-907  
 GTK++, 900  
 instalar, 900  
 programación, 904, 908  
 recursos en línea, 900  
 widgets, crear, 907  
**GNU.** compiladores, 808  
**GUIs.** 809, 896-899  
 interactuar con programas, 918  
 widgets, 903-904  
**historia.** 808  
**instalar.** 8  
**KDE.** 923  
 crear aplicaciones, 928-931, 934-936  
 crear aplicaciones con KDevelop, 937-938  
 habilidades, 902  
 instalar, 901  
**KDEHelloWorld.** programa, 924-927  
**Qt.** biblioteca, 903  
 recursos en línea, 903  
**lenguajes.** 816  
 lenguajes de secuencias de comandos, 818  
**llamadas de sistema.** 856-858  
 ipc(), 892  
 msgrcv(), 884  
 msgsnd(), 884  
 pipe(), 874  
 popen(), 877  
**msgbuf.** estructura, 884  
**msqid\_ds.** estructura, 882  
**multitareas.** 841  
 páginas del manual, 830  
 probar valores, 31  
 programación, 903  
 programación de sistemas, 853-856, 859-861  
 programador de tareas, 860  
 puertos de E/S, 837  
**shells**  
 archivos de inicio, 849  
 establecer variables, 840  
 funciones, 836  
 interactivos, 849  
 programar, 835-836  
 redirección de E/S, 837-839  
 variables de entorno, 839-840  
 variables locales, 839  
**shm\_id\_ds.** estructura, 890  
**wxWindows kit de herramientas**  
 agregar menús a la clase Window, 920-921  
 comunicación de objetos, 918  
 crear aplicaciones, 91-917  
 funcionalidad, 909-910  
 procesamiento de eventos, 919-920  
 wxStudio, 922  
**LinuxThreads, 861**  
**List, clase, 600**  
**lista, clases base, 661**  
**lista, contenedores, 699-701**  
**listados**  
 define, comando del pre-compilador, 749-750  
 +, operador, 319-320  
 abrir archivos para lectura y escritura, 586  
 acceder  
   datos miembro en el heap, 242  
   miembros públicos, 152  
**ADTs, derivar de otros ADTs, 445-448**  
 agregar al final de un archivo, 587-589  
 agregar un operador de incremento, 307-308  
 ajustar ancho de salida, 578  
 algoritmos de secuencia mutantes, 708-709  
 apuntadores descontrolados, crear, 248  
 apuntadores, 389-390  
   a funciones, 464-468  
   a funciones miembro, 478-479  
 a objetos const, 251-252  
 almacenamiento, 232-233  
 analizar cadenas sintácticamente, 253-255  
 como datos miembro, 243-244  
 archivos de encabezado, 134  
 archivos make, 136  
 argumentos de la línea de comandos, 592-593, 595  
**arreglos**  
 apuntadores a funciones, 470-472, 480-481  
 clases de arreglos, plantillas, 663-664  
 consts y enums, 374-375  
 creación de objetos, 376  
 crear por medio de new, 383-384  
 escribir más allá del final de, 370-371  
 guardar en el heap, 380-381  
 llenar, 386  
 multidimensionales, 378-379  
 arreglo de enteros, 368-369  
 asignación, operador (=), 322-323  
 asignar, usar y eliminar apuntadores, 237-238  
**ASSERT(), macro, 759**  
**bash, instrucciones de control, 847**  
**bloques (variables), 110-111**  
**botones.cc, programa (GNOME).** 906-907  
**break y continue, 187-188**  
**Cadena, clase, 392-395, 502-505, 553**  
**cadenas, 389-390**  
**campos de bits, 778-779**  
**ciclos eternos, 209-211**  
**ciclos while complejos, 185-186**  
**cin, manejar tipos de datos, 563**

- clase amiga, ejemplo.** 535-542  
**clase de bloqueo sincrónico.** 865  
**clases base.** 428-430  
**clases con métodos de acceso,** 154  
**clases de figuras.** 437-439  
**Condición, objetos variables de clase.** 869  
**condicional, operador.** 94-95  
**constantes enumeradas.** 61  
**Constantes(), método.** 762-768  
**constructor de copia virtual.** 360-363  
**constructores.** 161-162  
  clase contenida, 509-511  
  llamar, 340-342  
  sobrecargar en clases derivadas, 342-345  
**constructores de copia.** 303-304  
**Contador, clase.** 306-307  
**contravarianza.** 802-803  
**convertir**  
  de Contador a unsigned short(), 327-328  
  de int a Contador, 325-326  
  de USHORT a Contador, 326  
**Cout, instrucción.** 32-33  
**creación de vectores y acceso de elementos.** 694-697  
**crear referencias.** 260  
**crear/eliminar objetos del heap.** 240-241  
**dados.c, programa.** 816  
**datos miembro estáticos.** 456-457  
**declaración de clase gato en gato.hpp.** 169-170  
**declarar una clase completa.** 172-173  
**delegar a una lista enlazada contenida.** 517-524  
**demostración de una llamada a una función.** 36  
**demostrar el uso de variables.** 52  
**desplazarse a través de las listas por medio de iteradores.** 700  
**destino, direcciones (referencia).** 261  
**destructores.** 161-163, 340-342  
**determinar tamaños de tipos de variables.** 45  
**dirección de operadores.** 226-227  
**do - while, ciclo.** 192-193  
**else, palabra reservada.** 85  
**Empleado, clase y programa controlador.** 505-507  
**en línea, funciones.** 123, 755-756  
**enteros constantes.** 62  
**entrada múltiple.** 565-566  
**error de compilación.**  
  demonstración, 24  
**escribir**  
  más allá del final de un arreglo, 372  
  palabras múltiples, 564-565  
  una clase en un archivo, 590-591  
**espacios de nombres.** 607  
**especificación de tipo (conversión)**  
  descendente, 417-418  
  para un punto flotante, 73  
**Event, definición del objeto.** 868  
**excepciones**  
  con plantillas, 735  
  funciones virtuales en excepciones, 732-735  
  múltiples, 723-725  
  pasar por referencia en, 732-735  
  plantillas, 736-738  
  recuperar datos de, 728-731  
**fill(), método.** 579  
**for, ciclos.** 198  
  anidados, 200-201  
  instrucciones nulas vacías, 197-199  
**for-each, algoritmo.** 707-708  
**FUNC.CPP, demostración de una función simple.** 38  
**función amiga que no es de plantilla.** 669-672  
**funciones**  
  bibliotecas, 134  
  declaraciones, 103  
  objetos, 707  
  polimorfismo, 120-121  
**funciones de cadenas.** 387-388  
**funciones miembro estáticas.** 461-462  
**funciones virtuales.** 352-355  
**funciones y datos miembro estáticos.** 687-690  
**get(), función**  
  con arreglos de caracteres, 570-571  
  con parámetros, 569  
  sin parámetros, 568  
**getline(), función.** 571-572  
**globales y locales, variables.** 108-109  
**GNOMEHelloWorld, programa**  
  con botones, 913  
  con menús agregados, 921  
**guardados en el CD-ROM.** 21  
**HELLO.CPP, demostración de los componentes de un programa.** 30  
**HELP.CPP uso de comentarios.** 35  
**herencia**  
  múltiple, 420-422  
  privada, 526-533  
  sencilla, 414-415  
  simple, 336-337  
  virtual, 432-434

- hola.cxx (el programa Hola mundo), 21  
**ignore()**, 573  
 implementación de gato en gato.hpp, 170-171  
 implementaciones de plantilla, 682-686  
 implementar  
   funciones virtuales puras, 442-444  
   métodos de clases, 157-158  
   plantillas, arreglos, 665-668  
 imprimir  
   caracteres con base en los números, 57  
   con printf(), 582-583  
   valores en modo DEPURAR, 767-768  
 inicialización de variables miembro, 301  
 instrucciones de retorno múltiples, 115-116  
 instrucciones if anidadas, 87  
 instrucciones múltiples en ciclos for, 196-197  
 intentar asignar un int a un Contador, 324-325  
 intercambiar(), función reescrita con referencias, 270  
 ipc\_perm, estructura, 881  
 jerarquías de clases y excepciones, 725-728  
 KDEHelloWorld, programa, 924  
   con botones, 928  
   con clase Window derivada, 926  
   con menús, 934  
 Key, definición de clase, 880  
 límites de valores de los enteros con signo, 55  
 límites de valores de los enteros sin signo, 54-55  
 listas enlazadas, 400-408  
 llamar  
   a constructores múltiples, 424-426  
   a métodos de la base desde método redefinido, 350-351  
 llaves que aclaran las instrucciones else, 89  
 llenar un arreglo, 387  
 manipular datos mediante apuntadores, 231  
 map, clases de contenedores, 702-704  
 memoria, fugas, 287-289  
 Message, objetos, 885  
 miembros estáticos  
   acceder con métodos, 460  
   acceder sin un objeto, 458-459  
 msgbuf, estructura, 884  
 msqid\_ds, estructura, 883  
 múltiples excepciones, 722  
 Mutex, clase, 865  
 NamedPipe, definición de clase, 878  
 niveles de depuración, 769-774  
 Object, interfaz, 874  
 objetos de subprocessos, 862-863  
 objetos derivados, 338-339  
 objetos temporales, 311-312  
 ocultar métodos, 348-349  
 operator+ amigable, 544-548  
 ostream, operador, 673-676  
 paréntesis en macros, 753-754  
 partición de datos al pasar por valor, 358-359  
 pasar  
   apuntadores a funciones, 473-475  
   apuntadores const, 279-280  
   objetos de plantilla, 677-681  
   objetos por referencia, 276-277  
   por referencia usando apuntadores, 268-269  
   por valor, 113, 267-268, 512-515  
 peek() y putback(), 574  
 Pipe, definición de clase, 875  
 prefijo y posfijo, operadores, 76-77, 315-316  
 proceso con funciones agregadas para control de procesos, 856  
 Process, clase, 855  
 producir excepciones, 717-720  
 put(), 576  
 ramificación con base en los operadores relacionales, 82-83  
 rect.cpp, 173  
 rect.cxx, 174  
 recursión por medio de la serie de Fibonacci, 126-127, 135  
 redefinir un método de clase base, 347-348  
 referencias  
   a objetos no existentes, 286-287  
   asignar a, 263  
   pasar a objetos, 264-265, 281-283  
 regresar un objeto temporal, 310-311  
 repaso de la semana 1, 215-219  
 repaso de la semana 2, 487-494  
 repaso de la semana 3, 791-803  
 resta y desbordamiento de enteros, 72  
 Semaphore, definir, 889  
 sembuf, estructura, 888  
 semid\_ds, definición de estructura, 887  
 semop, llamada de sistema, 888  
 setf, 580  
 SharedMemory, clase, 891  
 shmid\_ds, estructura, 890

**sobrecargar**  
el constructor, 299-300  
funciones miembro, 294-295  
operator+(), 308-309  
operator<<(), 549-553  
**solución del enesimo**  
número de Fibonacci, 203-204  
**Suma()**, función, 317-318  
**switch**, instrucción, 206-207  
**this**, apuntador, 246-247, 313-314  
**tipos de datos abstractos**, 440-441  
tomar la dirección de una referencia, 262  
**typedef**, palabra reservada, 53  
**typedef** usado con apunadores a funciones, 475-477  
uso apropiado de llaves con instrucciones if, 90  
uso de ciclos con la palabra reservada goto, 182  
**valores**  
regresar con apunadores, 272-273  
regresar con referencias, 274-275  
valores de parámetros predeterminados, 117-118  
valores predeterminados, 296-297  
variables y parámetros locales, 106  
violaciones de la interfaz, 165-166  
while, ciclos, 184, 189-191  
**write()**, 576-577  
**wxWindows**, programa  
GNOMEHelloWorld, 910  
**listas**  
de parámetros, 100, 102  
múltiples, 119  
plantillas, 662  
desplazarse por medio de iteradores, 700

de tipo, 100  
de tipo dinámico, 100  
de tipo estático, 100  
de tipo constante, 100  
de tipo constante dinámico, 100  
de tipo constante estático, 100  
de tipo constante dinámico constante, 100  
de tipo constante estático constante, 100  
de tipo constante dinámico constante constante, 100  
de tipo constante estático constante constante, 100  
**llaves**, 31-104  
**llamar**  
**locales, variables**, 106-108, 839  
**logicos, errores**, 714  
solución de problemas, 714  
tips para depuración, 712  
**logicos, operadores**, 91  
AND, 91-92  
NOT, 91-92  
OR, 91-92  
precedencia, 91-92  
**long, enteros**, 45, 63  
**long, tipo de variable**, 54, 63  
**l-values**, 71

## M

**M-, comando**, 813  
**macros**, 752-753  
ASSEERT(), 758-761  
código fuente, 759-760  
depurar funciones, 760  
excepciones, 760  
limitaciones, 761-762  
listado, 759  
definir, 752-753  
desventajas, 754-755  
funciones, comparación, 754-755  
manipular cadenas, 756  
nombrar, 774, 783

- parámetros, 752  
 paréntesis (), 753-754  
 plantillas, comparación, 754-755  
 predefinidas, 758  
 sintaxis, 752-753  
 ventajas, 788  
**main()**, función, 30-31, 99  
**make**, comando, 135  
**make**, utilería, 820-821  
**makefile**, 136, 820-821  
 destino/dependencia, 136  
 formatos, 136  
 RCS, 828  
**Mamífero**, clase, 337  
**manejar tipos de datos**, 563  
**manejo de eventos**, 918-919  
**manipuladores**, 584  
**manipular**  
 apuntadores, 234  
 archivos de paquete, 133  
 búferes, 560  
 cadenas, 138  
 caracteres, 138  
 datos, 146, 231  
**mantenimiento (programas)**, 101, 137  
**map**, clases de contendores, 701-704  
**máquina virtual (VM)**, 10  
**matemáticas**, funciones, 137-138  
**Maullar()**, función, 154, 158  
**mayor o igual que (>=)**, operador, 81  
**mayor que (>)**, operador, 81  
**mecanismo de señales y ranuras**, 931  
**mecanismos**, 628  
**memoria**, 130  
 apuntadores, 225-227  
 a funciones, 463-477  
 a métodos, 477-482  
 arreglos, 380  
 const, 250-253  
 const this, 253  
 declarar, 228, 234  
 descontrolados/ambulantes, 247-250  
 desreferenciar, 229-230, 234  
 inicializar, 228, 234  
 manipulación de datos, 231-232  
 nombrar, 229  
 nulos, 228  
 perdidos, 228  
 pisotear, 249  
 reasignar, 239  
 RTTI, 416  
 this, 246-247, 313-314  
 ventajas, 234  
 arreglos, 379  
 asignadores 692  
 asignar, 149. *Vea también* crear  
 compartidos, 890  
 direcciones, 225, 230  
 determinar, 226-227  
 examinar, 232-233  
 guardar en apuntadores, 228-229  
 recuperar, 229-230  
 eliminar, 250  
 espacio de código, 130  
 examinar, 742  
 fugas, 287-289  
 delete, instrucción, 236  
 listado, 287-289  
 reasignación de apuntadores, 239  
**heap**, 235, 380  
 acceder, 241-242  
 apuntadores, 242-244  
 asignación de memoria, 236  
 datos, 235  
 objetos, 240-241  
 objetos, eliminar, 240  
 restaurar, 236-238  
 ventajas, 235-236  
**limpiar la memoria asignada**, 244  
**métodos virtuales**, 363  
**pila**, 130-131, 235, 705  
 limpiar, 235  
 meter datos en, 131-132  
 sacar datos de, 131-132  
 RAM, 44, 129-131  
 registros, 130  
 uso del compilador, 372  
 variables, 43, 54  
**memoria compartida**, 890  
 crear, 890  
 definir clase, 892  
 eliminar, 891  
**menor que, símbolo (<)**, 21, 32, 81  
**mensajes**  
 catch, instrucciones, 732  
 crear colas de, 882-885  
 enlazadores, 600  
 imprimir  
 en pantalla, 31  
**mensajes de advertencia**, 25  
**menu()**, función, 211  
**menús**  
 agregar a clase KDE Window, 934-936  
 agregar a la clase Window de wxWindows, 920-921  
**Message**, objetos, 885-886  
**Metaobjetos**, compilador (MOC), 927  
**Meta**, teclas, 812  
**metodologista**, 623  
**métodos de acceso**, 153-154  
**métodos de acceso públicos**, 153-154  
**métodos**. *Vea también* funciones  
 amigos, 548-549  
 apuntadores, 477-480  
 arreglos, 480-482  
 declarar, 478  
 invocar, 478, 480  
 archivos de encabezado, 271-272  
 AsignarEdad(), 242  
 clases, 157-158, 168  
 Clone(), 360, 363  
 Constantes(), 762-767, 770-771  
 constructores, 159  
 de copia, 302-306  
 declarar, 329  
 implementar, 163

- initializar, 300  
 llamar a múltiples, 424-426  
 predeterminados, 160-163, 298  
 sobrecargar, 299-300  
 de acceso, 154  
 declarar  
   ubicaciones de archivos, 167-168  
   valores predeterminados, 296-298  
 destructores, 159  
   implementar, 163  
   predeterminados, 160-163  
 en línea, 169-171, 330  
 estáticos, 461-462  
   acceder, 463  
   listado, 461-462  
   llamar, 461-463  
   modelo de diseño estático, 645  
   ventajas, 483  
**fill()**, 578-579  
**flush()**, 575  
**get()**, 386, 567-569  
   arreglos de caracteres, 570-571  
   parámetros de referencia de caracteres, 569  
   sobrecargar, 572  
**getline()**, 571-572  
 herencia privada, 525  
**ignore()**, 573-574, 596  
 implementar, 156-159  
 métodos de acceso públicos, 153-154  
 métodos de la base, llamar, 350-351  
**ObtenerArea()**, 174  
**ObtenerEdad()**, 158, 242  
**ObtenerSupIzq()**, 174  
 ocultar, 348-350  
**peek()**, 574-575  
**printf()**, 582-583  
**put()**, 575-576  
**putback()**, 574-575, 596  
 redefinir, 346-348  
**set()**, 580-581  
 sobrecargar, 293-295, 348  
 valores predeterminados, 329  
   declarar, 296-298  
   sugerencias de uso, 298  
 virtuales, 352-357  
   apuntadores-v, 356  
   constructores de copia, 360, 363  
   costos en memoria, 363  
   destructores, 360  
   herencia múltiple, 422  
   listado, 352-353  
   llamar múltiples, 353-355  
   partición, 358-360  
   tablas v, 356-357  
**width()**, 577-578  
**write()**, 575-577  
**métodos virtuales**, 352-357  
   aptrv, 356  
   constructores de copia, 360, 363  
   costos en memoria, 363  
   destructores, 360  
   herencia múltiple, 422  
   listado, 352-353  
   múltiples, llamar, 353-355  
   partición, 358-360  
   tablas v, 356-357  
**mezclas (clases de capacidades)**, 436  
**miembros**  
   clases, 151  
   estáticos, acceder, 691  
   objetos, acceder, 149  
**minimizar errores**, 167  
**mknod, comando de shell**, 878  
**MOC (Compilador de metaobjetos)**, 927, 932  
**modelos**  
   clases, 641  
   de dominio, 629  
   dinámicos, 657  
   estáticos, 644  
   programación orientada a objetos, 620  
   subtipos, 653  
**modelos dinámicos**, 657  
**modelos estáticos**, 644  
**modificar**  
   archivos, 827-829  
   funciones, 414  
**modo, editores de**, 810  
**mordiscos**, 951  
**MostrarTodo(), función**, 533  
**Motif**, 898  
**MOV**, 9  
**msgbuf, estructura (listado)**, 884  
**msgrecv(), llamada de sistema**, 884  
**msgsnd(), llamada de sistema**, 884  
**msqid\_ds, estructura**, 882  
**múltiple entrada**, 564-566  
**múltiple inicialización (ciclos for)**, 196-197  
**múltiples clases base**  
   constructores, 424-426  
   objetos, 423  
   resolución de ambigüedad, 426-427  
**múltiples excepciones**, 722-725  
**múltiples valores**, 272-275  
**multiplicación, operador**, 230  
**multiprocesamiento**, 859, 890  
**multitareas**, 841  
**mutex**, 864-866

**N**

- n (código de nueva línea)**, 31  
**NamedPipe, clase (listado)**, 878  
**navegar**  
   emacs, 812  
   vi, editor, 811  
**NCITS (Comité Estadounidense para Estándares de Tecnología de la Información), estándar**, 15  
**new, operador**, 283  
**new(), operador**, 693  
**new, instrucciones**, 236

- niveles de depuración, 769-774  
**no hacer nada, operador**, 84  
**no inicializar**  
 arreglos de caracteres, 385  
 búferes, 385-386  
 elementos de arreglos, 374, 411  
**NodoInterno, objeto**, 406-409  
**nodos**, 398-400  
**NoHayMemoria, excepciones**, 732  
**nombres**  
 ambigüedad, 611  
 apuntadores, 229  
 conflictos, 599-600  
 convenciones, 783  
 de arreglos como apuntadores, 382-384  
 de clases, 147-148  
 enlace, 603  
 espacios de, sin nombre, 613  
 identificadores, 783  
 macros, 774  
 ortografía, 783-784  
 plantillas, 664-665  
 referencias, 260  
 uso de mayúsculas, 783-784  
 variables, 48, 783  
 claridad, 49  
 notación de camelio, 50  
 notación húngara, 50  
 palabras reservadas, 50  
 precauciones, 48  
 susceptibilidad al uso de mayúsculas, 49-50  
 variables de conteo, 201  
**nombres de archivo, extensiones**  
 c, 167  
 CPP, 17, 167  
 CXX, 17-18  
 h, 168  
 hp, 168  
 hpp, 168  
 OBJ, 19  
**NOT, operadores**, 92  
**notación de camelio**, 50  
**notación húngara**, 50, 783  
**notación numérica**, 947-948  
**nueva línea, caracteres (\n)**, 31, 33  
**nueva línea, caracteres de escape**, 58  
**nueva línea, delimitador**, 386-387  
**nulas, instrucciones**, 67, 197-199  
**nulas, referencias**, 266  
**NULL, apuntadores**, 228, 266  
 eliminar, 250  
 en comparación con apuntadores perdidos, 250  
 probar por si hay, 238  
**nulo, carácter**, 385, 564  
**nulos, terminadores**, 388  
**numeral, símbolo (#)**, 30, 748  
**números**  
 base 10, 948  
 convertir a base 6, 949-950  
 convertir a base 7, 949  
 base 2 (binarios), 951-952  
 convertir a, 950  
 ventajas, 951  
 base 6 (hexadecimal), 953-956  
 base 7, 949  
 base 8 (octal), 948-949, 952-953  
 base, establecer, 579  
 contar (ciclos while), 189  
 Fibonacci, serie de, 203-204  
 fib(), función, 204  
 precauciones, 205  
 interpretación por computadora, 56  
 valor/letra, relación, 57  
**números hexadecimales**, 579, 953-956  
**números octales (base 8)**, 579, 952-953  
 base 10, 953  
 convertir números a ventajas, 953  
**O**  
**obj, archivos**, 19, 751  
**Object, interfaz (listado)**, 874  
**Objectory**, 623  
**objetos**, 423. *Vea también herencia; excepciones*  
 alias, 264  
 arreglos, 375-377  
 declarar, 375-376  
 listado, 376  
 comparados con las clases, 148  
 contenedores, 632, 692  
 cout, 31  
 crear, 873  
 de E/S estándar, 561  
 cerr, 561  
 cin, 561-575  
 clog, 561  
 cout, 561, 575-581  
 definir, 148, 155  
 derivados, 338-339  
 dominio, 643  
 eliminar, 873  
 estados, 579, 657  
 fuera de alcance, 286  
 funciones, 707  
 Gato, inicializar, 161, 163  
 guardar miembros por referencia, 245  
 heap, objetos del, 239  
 crear, 240  
 eliminar, 240-241  
**herencia**  
 conversión descendente, 416-419, 451  
 múltiple, 423  
**indicadores**, 775  
**inicializar**, 159, 300-301  
**iostream (estados)**, 579  
**Message**, 885  
**miembros, acceder**, 149  
**nulos**, 266  
**operador de asignación (=)**, 321  
**pasar**  
 partición de datos, 358-360

- por valor, 302  
referencias a, 282-283  
plantillas, pasar, 677-681  
programación orientada a, 620  
Qt, 903  
referenciar, 264-265  
en el heap, 287-289  
listado, 264-265  
no existentes, 286-287  
relaciones, 631  
representaciones, 146  
señales, 931  
subproceso, 863  
sustitutos, 643  
tamaños, 177  
temporales  
crear, 313  
regresar, 310-311  
sin nombre, 311-313  
valores, asignar, 149-150  
visibilidad, 602  
wxWindows, aplicaciones, 918
- objetos de E/S estándar, 561**  
cerr, 561  
cin, 561-563  
cadenas, 564  
entrada múltiple, 564-567  
get(), método, 567-571  
getline(), método, 571-572  
ignore(), método, 573-574  
operador de extracción, 567  
peek(), método, 574-575  
putback(), método, 574-575  
clog, 561  
cout, 561, 575-581  
flush(), método, 575  
put(), método, 575-576  
write(), método, 576-577
- objetos de función, 707**  
**objetos derivados (listado), 338-339**  
**objetos no existentes, 286-287**
- objetos nulos, referencias a, 289**
- objetos temporales**  
crear, 313  
listado, 310-311  
nombrar, 312  
regresar  
funciones sobrecargadas, 310-313  
sin nombre, 311-312
- ObtenerArea(), función, 174**  
**ObtenerEdad(), función, 158, 242**  
**ObtenerEstado(), función, 779**  
**ObtenerPeso(), función, 169**  
**ObtenerSupIzq(), función, 174**
- ocultar**  
identificadores, 601  
métodos, 348-350  
redefinir, comparación, 350  
variables (globales), 108
- ofstream, objetos, 585**  
abrir archivos para E/S, 585-587  
argumentos, 587  
comportamiento predeterminado, 587-589  
estados de condición, 585
- OMT (Tecnología de Modelado de Objetos), 623**
- opciones (línea de comandos), 592**
- operaciones de secuencia no mutantes, 707-708**
- operador de adición autoasignado (+=), 74**
- operador de resolución de ámbito (::), 601**
- operador igual a(==), 80**
- operadores aritméticos, 71-72**  
combinar con operadores de asignación, 74-75  
precedencia, 78, 944  
residuo (%), 72-73  
resta (-), 71-72
- operadores de conversión, 324-328**
- operadores de despliegue, agregar, 673**
- operadores integrados, 306**
- operadores matemáticos, 71-72**  
combinar con operadores de asignación, 74-75  
precedencia, 78, 944  
residuo (%), 72-73  
resta (-), 71-72
- operadores unarios, sobrecargar, 317**
- operadores, 71. Vea también funciones**  
<, 81  
<=, 81  
>, 81  
>=, 81  
a nivel de bits, 775  
AND (&), 775  
complemento (~), 776  
OR (|), 776  
OR exclusivo (^), 776  
apunta a (->), 241  
aridad, 321  
asignación (=), 51, 68, 71, 321-324  
binarios, parámetros, 320  
concatenación, 757-758  
condicional (>:), 94-95  
conversión, 324-328  
de despliegue, 673  
de dirección (&), 226-227, 261-262  
declarar sobrecargados, 327  
decremento (-), 75  
postfijo, 75-77  
prefijo, 75-77  
delete(), 693  
direcciones, 226-227  
dynamic\_cast, 417  
extracción (>>), 562  
incremento (++), 75  
agregar, 307-308  
postfijo, 75, 77  
prefijo, 75, 77  
indirección (<), 229-230, 284-285  
inserción (<<), 32, 549, 553

- integrados, 306  
 lógicos, 91  
     AND (&&), 91  
     NOT (!), 92  
     OR (||), 91-92  
 matemáticos, 71-72  
     autoasignados, 75  
     multiplicación, 230  
     residuo (%), 72-73  
     resta (-), 71-72  
 new, 283  
 new(), 693  
 no hacer nada, 84  
 ostream, 673-676  
 posfijo, 314-316  
 precedencia, 77-78, 92-93  
 prefijo, 314-316  
 punto (), 155, 241  
 redirección, 561  
 referencia (&), 260,  
     284-285  
 relaciones, 80-83  
 sobrecargar, 306-307  
     cuestiones, 320  
     funciones amigas,  
         544-548  
     limitaciones, 321  
     objetivos, 321  
     objetos temporales,  
         310-313  
     prefijo, operadores,  
         308-310  
     tipos de valor de retorno,  
         310-313  
 static\_cast, 73  
 suma (+), 317-320  
 unarios, 317
- operandos, 71**
- operator, instrucciones, 317**
- operator+ amigable (listado), 544, 546, 548**
- OR exclusivo (^), operador a nivel de bits, 776**
- OR, operadores**  
     a nivel de bits (!), 776  
     lógico (||), 91-92
- ordenar**  
     elementos de arreglos,  
         409-410
- manejadores de excepciones, 728
- ostream, clase, 561**
- ostream, operador, 673-676**
- P**
- páginas del manual, 830, 837
- palabras múltiples, 564-565**
- palabras reservadas, 50, 176, 609**  
     catch, 721  
     class, 147, 155-156, 663  
     const, 163, 177, 250-251,  
         283, 603  
     delete, 236-238  
     else, 85-86  
     enum, 60  
     for, 195-196  
     friend, 549  
     goto  
         ciclos, 181-182  
         limitaciones, 183  
         recomendaciones, 183  
         sintaxis, 183  
     inline, 122, 142, 169  
     new, 236  
     operator, 317  
     private, 153  
     protected, 337  
     public, 153, 158  
     RCS, 828  
     return, 114-116  
     static, 614  
     struct, 175-176  
     template, 663  
     try, 721  
     typedef, 53
- palabras reservadas de control de acceso, 155**
- pantalla, imprimir en, 32-34, 561**
- papeles, 627**
- paquetes, 637**
- paquetes de bibliotecas, conflictos de nombres, 600**
- paquetes, manipular, 133**
- parametrizar**  
     clases de arreglos, 662  
     plantillas, 662, 710
- parámetros, 37**  
     funciones, 104, 112  
     get(), método, 569  
     key, 884  
     listas, 100-102, 119  
     ocultos, 246  
     pasar, 592  
     pasar por valor, 113-114  
     predeterminados, 116-118  
     procesar en la línea de comandos, 593, 595  
     rhs, 305  
     UsarValActual, 298  
     variables locales, 106
- parámetros de referencia de caracteres (método get()), 569**
- parámetros predeterminados (funciones), 116-118**
- parches, 192**
- paréntesis (), 78**  
     agrupar, 93  
     anidación, 78-79  
     macro, sintaxis, 753-754
- partición (métodos virtuales), 358-360**
- particionar la RAM, 129-131**
- paso**  
     apuntadores a funciones, 472-475  
     apuntadores const, 278-281  
     argumentos, 104  
         para constructores base, 342-346  
         por referencia, 266-268, 276-278  
         por valor, 267-268  
     memoria entre funciones, 290
- objetos**  
     partición de datos, 358-360  
     por valor, 302
- objetos de arreglos a funciones, 668-669**
- objetos de plantilla, 677-681**

- parámetros, 113-114, 592  
**por referencia**  
 apuntadores (listado), 268  
 excepciones, 732-735  
 objetos (listado),  
 276-277  
 por valor, 113, 267-268,  
 512-515  
**referencias**  
 a objetos, 282-283  
 a objetos (listado),  
 281-283  
 excepciones, 732-735  
**PATH, variable de entorno,**  
 840  
**patrones de diseño, 449-450**  
**patrones de observación,**  
 449-450  
**peek(), método, 574-575**  
**perdidos, apuntadores, 228.**  
*Vea también apuntadores*  
**descontrolados**  
**Perro, clase**  
 constructores, 340-342  
 constructores sobrecargados,  
 345  
 declarar, 335-337  
 destructores, 340-342  
**persistencia (variables**  
**locales), 235**  
**personas, 627**  
**pila, llamadas, 722**  
**pila (memoria), 235, 380**  
 datos, 131-132  
 excepciones, 722  
 limpiar, 235  
 memoria, 130-131  
**pilas, 705**  
**Pipe, clase, definir (listado),**  
 875  
**pipe(), llamada de sistema, 874**  
**pisotear apuntadores, 249**  
**plantillas, 661-662**  
 agregar al ejemplo de la  
 semana 2, 800  
**amigas, 669**  
 de tipo específico, 710  
 generales, 673-676, 710  
 que no son de, 669, 673  
**arreglos**  
 clases, 663-664  
 implementar, 665-667  
**clases, 705**  
 comparación con herencia,  
 710  
 datos miembro estáticos,  
 687, 691  
 definir, 662-664  
 excepciones, 735-738  
**funciones**  
 contenedores vectoriales,  
 698  
 datos miembro estáticos,  
 687-690  
 declarar, 669, 673  
 especializadas, 681-682,  
 686-687  
 implementación, 673  
 map contenedores, 704  
 implementar, 665-668  
 instanciamiento, 662  
 instancias, 662  
 macros, comparación,  
 754-755  
 nombrar, 664-665  
 objetos, pasar, 677-681  
 parametrizadas, 662, 710  
 soporte del compilador, 665,  
 791  
 STL, 691-694, 698-701,  
 705  
**polimorfismo (funciones), 14,**  
**120-122, 352, 413**  
 funciones, 119-121  
 lograr en C++, 800  
 minar, 419  
**popen(), llamada de sistema,**  
 877  
**postijo, operadores, 75-77**  
**POSIX, biblioteca de subpro-**  
**cesos, 861**  
**POSIX, estándar, 808**  
 listado, 76, 315-316  
 operador de prefijo, compa-  
 ración, 314-316  
 sobrecargar, 314  
**precedencia (operadores),**  
**77-78, 92-93**
- precompiladores**  
 comandos  
 #define, 748-750  
 #else, 749-751  
 #ifdef, 749  
 #ifndef, 749  
 guardias de inclusión,  
 751-752  
 pruebas, 749  
**sustituciones**  
 cadenas, 748  
 constantes, 749  
**predefinidas, macros, 758**  
**prefijo, operadores, 75, 77**  
 listado, 76, 315-316  
 operador de postijo, com-  
 paración, 314-316  
 sobrecargar, 308-310  
**prefijos (variables), 147**  
**preprocesadores, 30, 747-748**  
 constantes de clases,  
 762-767, 770-771  
 funciones en línea, 755-756  
 macros, 752-753  
 ASSERT(), 758-761  
 definir, 752-753  
 desventajas, 754-755  
 parámetros, 752  
 paréntesis (), 753-754  
 predefinidas, 758  
 sintaxis, 752-753  
 ventajas, 788  
 manipulación de cadenas,  
 756  
 concatenación, 757-758  
 uso de cadenas, 757  
 niveles de depuración,  
 768-769, 773-774  
 valores interinos, imprimir,  
 767-768  
**printf(), función**  
 archivos de encabezado,  
 581  
 dar formato con, 581  
 en comparación con cout,  
 581  
 flujos, comparación, 582  
 limitaciones, 581, 596  
**prioridad (procesos), 859**

- private, palabra reservada, 153**
- private slots, sección, 932**
- probar**  
cadenas, 749  
por si hay apuntadores nulos, 238  
valores de retorno, 31
- procedimientos, 11, 620**
- procesadores de palabras, 25**
- procesar (línea de comandos), 592-593**
- procesos, 853**  
acceder a recursos, 886  
agregar funciones para control de, 857  
colas de mensajes, 882-885  
comunicación, 873  
condición de carrera, 856  
control de, 856-858  
crear, 854-855  
cuantos, 859  
directivas, 860  
estado del proceso, 859  
fork(), función, 854  
hijo, 856  
iterativo, 624  
listados  
funciones agregadas para control de procesos, 856  
Process, clase, 855  
subprocesos, 862-863
- memoria compartida, 890**  
crear, 890  
definir clase, 892  
eliminar, 891
- multiprocesamiento, 859**
- prioridad, 859**
- programación orientada a objetos, diseño, 622**
- programador de tareas, 860**
- semáforos, 886**  
crear, 887  
definir clase, 889
- sincronización, 886-889**
- subprocesos, 860**  
crear, 862-864  
directivas de sincronización, 864-870
- implementar, 861  
listado, 862-863  
programar, 864  
punto muerto, 870  
simples, 862  
subprocesamiento multiple, 862  
terminar, 856  
tuberías, 874-875  
con nombre, 877-879  
semidúplex, 876
- procesos de tiempo real, 859**
- procesos hijos**  
controlar, 858  
crear, 856  
descriptores de archivos, 875
- procesos normales, 859**
- Process, clase (listado), 855**
- producir excepciones (listado), 717-720**
- programación**  
a la defensiva, 62, 714  
alias, 53  
alias de comandos, 845-846  
apurarse al código, 658  
archivos ejecutables, 19  
bash  
completación de comandos, 842  
sustitución de cadenas, 843  
sustitución de salida de comandos, 844  
sustitución de variables, 844  
sustitución mediante comodines, 843
- C++ en el escritorio de Linux, 903**
- ciclo de desarrollo, 19-20
- código fuente, compilar, 18-19**
- comentarios, 784, 827
- controlada por eventos, 13, 897
- de sistemas, 853-856**  
crear, 854  
procesos, 853
- de tareas  
algoritmo, 860  
subprocesos, 864
- depurar archivos, 815
- diseño, 16
- entornos de desarrollo, 17
- entornos de subprocesamiento múltiple, 870
- estructurada, 11-12
- GNOME, 899, 904**  
API, 905  
aplicaciones, 908  
crear widgets, 907  
GDK++, 904-907  
GTK++, 900  
instalar, 900  
recursos en línea, 900
- GNU, utilería make, 820-821**  
opciones de la línea de comandos, 822
- guardias de inclusión, 752
- KDE, 923**  
agregar botones a programas, 928-931  
agregar menús a programas, 934-936  
habilidades, 902  
instalar, 901  
KDEHelloWorld, programa, 924-927  
KDevelop, 937-938  
Qt, biblioteca, 903  
recursos en línea, 903
- lenguajes de procedimientos, 620**
- lineamientos en el estilo del código, 780**
- niveles de abstracción, 129**
- orientada a objetos, 13, 660**  
encapsulación, 13  
herencia, 14  
lenguajes, 660  
ocultación de datos, 13  
polimorfismo, 14  
ventajas, 660
- palabras reservadas, 50**
- preparación, 16**

- ramificación de programas, 132-133  
 recursos, 786  
 reentrancia, 862  
 shells, 835-836  
   instrucciones de control, 846-848  
   secuencias de comandos, 846  
 solución de problemas, 11  
 subprocessos  
   crear, 862-864  
   directivas de sincronización, 864-870  
   listado, 862-863  
   programar, 864  
   punto muerto, 870  
 validación de datos, 62  
 variables, 44  
 widgets, 903-904  
 wxWindows, aplicaciones, 911-912  
   botones, 913-917  
   comunicaciones de objetos, 918  
   kit de herramientas, 910  
   menús (clase Window), 920-921  
   procesamiento de eventos, 919-920  
   wxStudio, 922  
 X, 897-898  
**programador de tareas, 860**  
**programas a prueba de todo, 714**  
**programas infractores, 370**  
**programas. Vea también aplicaciones**  
   a prueba de todo, 62, 714  
   ambigüedad de nombres, 611  
   apuntadores, 253  
   apuntadores descontrolados, 250  
   bibliotecas compartidas, 818  
   bugs o errores, 714  
   caja negra, método, 272  
   ciclos principales de eventos, 925  
   código fuente, 18  
   comentarios, 34-35, 716  
   // (estilo C++), 34  
   /\* (estilo C), 34  
   cuándo utilizar, 34  
   listado, 35  
   precauciones, 35  
   compilar, 24, 817  
   conflictos de nombres, 599  
   conscientes de la sesión, 900  
   control de versiones, 827  
   controlados por eventos, 13, 897  
   corrupción del código, 715  
   crear fugas de memoria, 239  
   definición, 10-11  
   depurar, 739  
     archivos core, 741  
     ASSERT(), macro, 760  
     ensambladores, 742  
     examinar memoria, 742  
     gdb, depurador, 823-827  
     GNU, depurador, 740-741  
     imprimir valores interiores, 767-768  
     niveles, 768-774  
     puntos de interrupción, 742  
     puntos de observación, 742  
   desarrollar, 11, 19-20  
   diseñar, 16  
   eficiencia, 122  
   ejecutables, 11, 19  
   ejecutar, 36  
   elegir la herramienta adecuada para programar, 190  
   enfoque, 918  
   enlazar, 818  
   espacio en blanco, 284  
   estructura de, 12, 29-31  
   excepciones  
     atrapar, 722  
     funciones virtuales, 734  
     múltiples, 722  
     ordenar manejadores, 728  
   plantillas, 735-736, 738  
   sin errores, 738-739  
   fragilidad, 714  
   fugas de memoria, 239  
   GNOME, 908  
   GTK++, 907  
   GUI, elementos, 896  
   Hola, mundo, 20  
     código fuente, 21, 29-30  
     compilar, 21-22  
     ejecutar, 22  
     listado, 21  
   imprimir mensajes en pantalla, 31  
   infractor, 370  
   interactuar con programas de GUI, 918  
   llamar a función fork(), 854  
   lógica, 714  
   macros, 752  
   manejo de eventos (wxGTK), 918-919  
   mantenimiento, 101, 137  
   marcos de trabajo de aplicaciones, 903  
   nulas, referencias, 266  
   nulos, apuntadores, 250  
   parches, 192  
   por procedimientos, 897  
   precedencia de operadores, 943-944  
   preparar para situaciones excepcionales, 715  
   probar valores de retorno, 31  
   procedimientos, 11  
   procesos, 853  
   programación orientada a objetos (poo), 13, 620  
   ramificación, 132-133  
   referencias a objetos nulos, 289  
   reinventar la rueda, 12  
   secuencias de comandos de shell, 846  
   solución de problemas, 16  
   apuntadores descontrolados, 247-249  
   excepciones, 720-721

- variables de conteo, 201  
**while**, ciclo, 184  
**X**, clientes, 898
- propiedad (apuntadores)**, 290
- protected**, palabra reservada, 337
- protected slots**, sección, 932
- prototipos**, 102-104, 624  
 escribir, 101  
 nombres de parámetros, 103  
 parámetros, 271-272  
 tipos de valor de retorno, 102, 105
- proyectos**  
 artefactos, 640  
 crear archivos, 134  
 documentos de planeación, 639
- PS1**, variable de entorno, 840
- pthreads**, compilar con, 861
- ptrace()**, llamada de sistema, 858
- public**, palabra reservada, 153, 158
- public slots**, sección, 932
- puerto de entrada estándar**, 837
- puerto de error estándar**, 837
- puerto de salida estándar**, 837
- punto flotante**, especificación de tipo para (listado), 73
- punto flotante**, variables de, 47
- punto muerto**, 870
- punto muerto mutuo**, 870
- punto muerto recursivo**, 870
- punto y coma (;)**, 68
- Punto**, clase, 174
- punto**, operador de (.), 155, 241
- puntos (operador de punto)**, 155, 241
- puntos de interrupción**, 741-742
- puntos de observación**, 741-742
- push y pop**, 705  
**putt()**, función, 575-576  
**putback()**, función, 574-575, 596  
**PWD**, variable, 840
- Q**
- Qevent**, objetos, 928
- Qt kit de herramientas**  
 clases, 932  
 mecanismo de señales y ranuras, 931
- quitar marcas de comentario de las instrucciones cout**, 508
- R**
- RAM (memoria de acceso aleatorio)**, 44  
 particionar, 129-131  
 variables, 44
- ramificación**  
 con base en los operadores relacionales (listado), 82-83  
 funciones, 105  
 programas, 132-133
- ranuras**, 931
- ranuras virtuales**, 932
- rastrear cambios en archivos**, 828
- Rational Rose**, 630
- Rational Software, Inc**, 623
- RCS (Sistema de control de revisiones)**, 827-829  
 archivos make, 828  
 comandos, 827  
 palabras reservada, 828
- reasignar**  
 apuntadores, 239  
 referencias, 262
- rect.cpp (código fuente)**, 173
- rect.cxx (código fuente)**, 174
- Rectangulo**, clase  
 declarar, 173-175, 300  
 funciones  
 constructores, 174  
 DibujarFigura(), 293-295  
 ObtenerArea(), 174  
 ObtenerSuplizq(), 174  
**recuperar datos**, 590  
 en excepciones, 728-732
- recursión**  
 de paro, condiciones, 125  
 directa, 124  
 Fibonacci, serie de, 126-127, 135  
 funciones, 124-128  
 indirecta, 124  
 iteración, comparación, 212
- recursos**, 786
- redefinir**  
 const, palabra reservada, 603  
 métodos, 346-348  
 métodos de una clase base (listado), 347-348  
 métodos sobrecargados, 350  
 ocultar, comparación, 350  
 sobrecargar, comparación, 348
- redirección**  
 de entrada, comando (<), 562  
 de entrada, símbolo (<), 561  
 de E/S, 838-839  
 de salida (>), carácter, 838  
 de salida, comando (>), 562  
 de salida, operador (<<), 31  
 de salida, símbolo (>), 561
- redireccionamiento, operadores (<<)**, 21, 31, 561
- redirigir**  
 dispositivos, 561  
 flujos, 561-562  
 tuberías, 839
- redirigir salida a, símbolo (!)**, 562
- reentrancia**, 862
- referencia, operador de (&)**, 260, 284-285

- referencias, 259-261, 265**
- apuntadores
    - combinar, 284
    - comparación, 283-284, 291
    - asignar a (listado), 263
    - como alternativas para los apuntadores, 281
    - crear, 260-261, 265
    - declarar, 260, 285
    - destino, direcciones
      - asignar, 262-264
      - listado, 261
      - regresar, 261-262
    - errores
      - objetos no existentes, 286-287
      - referenciar objetos en el heap, 287-289
    - inicializar, 261
    - intercambiar(), función, 269-271
    - nombrar, 260
    - nulas, 266
    - objetos, 264
      - en el heap, 287
      - listado, 264-265
      - no existentes, 286-287
      - nulos, 289
    - pasar
      - a objetos (listado), 281-283
      - argumentos de funciones, 266-267
      - por referencia, 268, 275-278
    - reasignar, 262
    - regresar valores múltiples, 274-275
    - variables locales, 287
  - regresar**
    - objetos temporales (listado), 310-311
    - valores, 114
      - con apuntadores (listado), 272-273
      - con referencias (listado), 274-275
  - valores múltiples**
    - apuntadores, 272-274
    - referencias, 274-275
  - relacionales, operadores, 80-81**
    - precedencia, 92-93
    - ramificación, 82-83
  - relaciones**
    - clases (diseño, programación orientada a objetos), 648
    - es un, 334, 352
  - relaciones es un, 334, 352**
    - derivación, 334
    - herencia pública, 516
  - relaciones tiene un. Vea contención**
  - llenar funciones, 335**
  - repasos**
    - semana 1, 215-219
    - semana 2, 487-497
    - semana 3, 791-803
  - reportes (sistema), 644**
  - representaciones (objetos), 146**
  - requerimientos**
    - análisis, 626
    - documentos, 639
  - reservadas, palabras (lista de), 945-946**
  - resetiosflags, manipuladores, 584**
  - residuo, operador de (%), 72-73**
  - resolver ambigüedad (clases base múltiples), 426**
  - resolver el enésimo número de Fibonacci (listado), 203-204**
  - resta, operador (-), 71-72**
  - resta y desbordamiento de enteros (listado), 72**
  - restaurar memoria en el heap, 236-237**
  - resultados, 628**
  - RET, comando, 813**
  - retorno, valores de**
    - funciones, 100, 114-116
    - múltiples, 114-116
  - return, instrucciones, 37, 114-116**
  - reutilización, 12**
    - de código, 14, 133, 137, 140-141
  - rhs, parámetro, 305**
  - RR (por petición), directiva de procesos, 860**
  - RTTI (Identificación de Tipo en Tiempo de Ejecución), 416**
    - imitar, 416
    - precauciones, 417
    - soporte del compilador, 419
  - Rumbaugh, James, 623**

## S

- salida, 575**
  - ancho, ajustar, 578
  - archivos, 585-589
  - dar formato, 583-584
    - ancho, 577-578
    - caracteres de llenado, 578-579
    - indicadores, 580-581
  - dispositivos de salida
    - escribir a, 575-576
    - limpiar o vaciar, 575- salir de ciclos**
  - break, instrucción, 186
  - ciclos eternos, 209
- saltos, 182**
- sangría en el código, 781-782**
- sangría, estilos, 84-85**
- secuencias de comandos**
  - instrucciones de control, 846-848
  - shell, 846
- secuencias mutantes, algoritmos, 708-709**
- seguridad (clases), 155**
- semáforos, 867, 886**
  - crear, 887
  - definir clase, 889
- Semaphore, clase, 889**
- sembuf, estructura, 888**

- semid\_ds, estructura, 887**
- semop, llamada de sistema (listado), 888**
- señales, 858, 918**
- sesiones CRC, 645**
- setf(), método, 579-581**
- Setiosflags, manipuladores, 584**
- setw, manipulador, 580**
- sh, 836**
- SharedMemory, clase, 891**
- SHELL, variable, 840**
- shells, 836**
  - alias de comandos, 845-846
  - archivos de inicio, 849
  - bash
    - completación de comandos, 842
    - editar comandos, 845
    - instrucciones de control, 846, 848
    - lista del historial de comandos, 844
    - sustitución de cadenas, 843
    - sustitución de salida de comandos, 844
    - sustitución de variables, 844
    - sustitución mediante comodines, 843
  - comandos, 836
  - control de procesos, 842
  - disponibles en Linux, 836
  - establecer variables, 840
  - función, 836
  - interactivos, 849
  - páginas del manual, 837
  - programar, 835-836
  - redirección de E/S, 837-839
  - secuencias de comandos, 846
  - variables, 846
    - de entorno, 839-840
    - locales, 839

**shmget(), llamada de sistema, 890**

**shmid\_ds, estructura (listado), 890**

**short, enteros, 45**

**short, tipo de variable, 54, 63**

**signo**
  - de admiración ('), 92
  - de intercalación (^), operador OR exclusivo, 776
  - más (+), carácter
  - operador de suma, 317-320
  - prefijo, operador (++), 308-310

**signos de interrogación (caracteres de escape), 58**

**símbolo &**
  - operador dirección de, 226-227, 775
  - referencias, 261-262
  - operador lógico AND (&&), 91
  - referencia, operador, 260, 284-285

**símbolos**
  - &, 841
  - comodines, 838, 843
  - compilar, 739
  - >> (operador de extracción), 562
  - < (redirigir entrada), 561
  - > (redirigir salida), 561
  - | (redirigir salida a entrada), 562
  - | (tubería), 839

**sincronizar procesos, 886-889**

**síncronos, bloqueos (listado), 865**

**sintácticos, errores, 714**

**sintaxis**
  - catch, bloques, 721
  - definiciones, 169
  - errores, 714
  - for, instrucciones, 195
  - switch, instrucciones, 205
  - try, bloques, 721

**Sistema de Control de Revisiones (RCS). Vea RCS sistemas**
  - actores, 626
  - análisis, 638
  - dispositivos, 644
  - heredados, 643
  - vistas, 644

**sistemas operativos**
  - diseño y análisis de programas, 638
  - función de redirección, 562

**sitio de TrollTech, 923**

**sitios**
  - Cygnus, 9
  - GNOME, 900
  - GNU, 9
  - GTK++, 901
  - KDE, 903
  - KDevelop, 903
  - WxWindows, kit de herramientas, 923

**sizeof(), función, 46**

**sobrecargar**
  - Cadena, clase, 395
  - constructores, 299-300
  - clases derivadas, 342-345
  - listado, 299-300
  - extracción, operadores de, 562
  - funciones, 119-120
  - miembro, 294-295
  - métodos, 293-295
  - operador de inserción (<<), 549, 553
  - operadores, 306-307
    - binarios, 320
    - cuestiones, 320
    - limitaciones, 321
    - objetivos, 321
    - objetos temporales, 310-313
    - postfijo, 314
    - prefijo, operadores, 308-310

- suma (+), 317-320  
 tipos de valor de retorno, 310-313  
 unarios, 317  
 redefinir, comparación, 348
- sobregirar**  
 el valor de un entero con signo, 55  
 el valor de un entero sin signo, 54
- software**  
 casos de uso, 626  
 código fuente abierto, 808  
 comercial, 15  
 CORBA, 900  
 diseño, 622-623  
 GNU, 7  
 Rational Rose, 630
- solución de problemas, 16,**  
**714. Vea también depuración**  
 ambigüedad en propiedad de apuntador, 290  
 apuntadores descontrolados, 247, 249  
 ASSERT(), macro, 761  
 bugs o errores  
     arreglos, 369  
     depuradores simbólicos, 739  
     distinguir entre, 714  
     lógicos, 715  
 comentarios en el código, 716  
 costo, 714  
 depurar, 739, 822-823  
     archivos core, 741  
     ayuda, 823-824  
     cerrar, 824  
     comandos, 824  
     ensambladores, 742  
     examinar memoria, 742  
     GNU, depurador, 740-741  
     niveles, 774
- puntos de interrupción, 742  
 puntos de observación, 742  
 sesión de ejemplo con gdb, 826-827
- descomposición del código, 715-716
- errores  
     en tiempo de compilación, 167  
     lógicos, 164-167
- excepciones, 715-716, 720-721  
     atrapar, 722  
     catch, bloques, 721-725  
     desventajas, 744  
     funciones virtuales, 734  
     jerarquías, 725-728  
     opciones, 715  
     ordenar manejadores, 728  
     plantillas, 735-738  
     producir, 717-720  
     sin errores, 738-739  
     soporte del compilador, 717  
     try, bloques, 716, 721  
     ventajas, 743
- guardias de inclusión, 751  
 imprimir valores interinos, 767  
 macros, 760  
 predefinidas, macros, 758  
 recursos para el programador, 786  
 referencias a objetos nulos, 289
- SOs, 836**  
 CPU, tiempo, 859  
 GUIs, 896  
 subprocessos, 861  
 widgets, 903-904
- SPC, comando, 813**
- sqrt(), función, 272**
- Stallman, Richard M., 7**
- static, palabra reservada, 604, 614**
- static cast, operador, 73**
- std, espacio de nombres, 614-615**
- stdio.h, archivo de encabezado, 581**
- STL (Biblioteca Estándar de Plantillas), 691-694, 698-701, 705**  
     clases de algoritmos, 706  
     contenedor clases, 692  
     map, contenedores, 701
- streat(), función, 388**
- strepy(), función, 387-388, 391**
- streambuf, clase, 560**
- strlen(), función, 388**
- strncpy(), función, 388**
- Stroustrup, Bjarne, 14, 176**
- Struct, palabra reservada, 175-176**
- su, prefijo, 147**
- subclases, 171-175**
- subíndice, 367, 948**  
     operador, 698
- subprocesamiento**  
     múltiple, 862  
     simple, 862
- subprocesos, 860**  
     crear, 862-864  
     directivas de sincronización, 864-870  
     implementar, 861  
     listado, 862-863  
     POSIX, biblioteca de, 861  
     programar, 864  
     punto muerto, 870  
     simples, 862  
     múltiples, 862
- suma**  
     operador (+), 317-320  
     operador autoasignado (+=), 74
- Suma(), función, 38**  
     declarar, 317-318  
     listado, 317-318

- superclases, 646**
- superconjuntos, 334**
- superestados, 658**
- susceptibilidad al uso de mayúsculas y minúsculas, 148**
- nombres de variables, 49-50
  - notación húngara, 50
- suspender trabajos, 841**
- sustitución**
- cadenas, 843
  - comodines, 843
  - salida de comandos, 844
  - variables, 844
- sustitución de salida de comandos, 844**
- sustitutos, 643**
- switch, instrucciones, 205-207, 451**
- break, instrucciones, 207
  - ciclos eternos, 208-211
  - listado, 206-207
  - ramificar mediante valores múltiples de una expresión, 207
  - sangría, 781-782
  - sintaxis, 205-208
  - sugerencias de uso, 211
  - valores de case, 206
- SyncLock, clase base virtual, 865**
- System V IPC, 879-880**
- T**
- t, shell (tcsh), 836**
- TAB, comando, 813**
- tablas**
- de eventos, 920
  - v, 356-357
- tabs**
- caracteres de escape, 58
  - código de escape, 33
- tag, archivos, 815**
- tamaños**
- arreglos, 373-375
  - enteros, 45
  - variables, 44, 47-48
  - char, 56
  - determinar, 45-46
  - uso de memoria, 54
- TASK INTERRUPTIBLE, estado del proceso, 859**
- TASK RUNNING, estado del proceso, 859**
- TASK STOPPED, estado del proceso, 859**
- TASK SWAPPING, estado del proceso, 859**
- TASK UNINTERRUPTIBLE, estado del proceso, 859**
- TASK ZOMBIE, estado del proceso, 859**
- tcsh, 836**
- teclas o claves**
- Meta, 812
  - System V IPC, 879-880
- template, palabra reservada, 663**
- TERM, variable de ambiente, 840**
- terminar**
- línea, 33
  - procesos, 856
- ternario, operador (>:), 94-95**
- this, apunadores, 246**
- const this, 253
  - funciones
  - de acceso, 247
  - miembro estáticas, 463
  - listado, 246-247, 313-314
- tiempo de compilación, 26**
- tilde (~), 159, 776**
- tipo específico, funciones amigas, 710**
- tipo int, convertir en objeto Contador, 325**
- tipos, 146. Vea también categorías**
- crear, 145-146
- de poder, 652-656**
- de valor de retorno, 102, 105**
- declarar, 155**
- definición de, 53**
- valores, 101**
- variables, 145**
- tipos de datos**
- abstractos, 436-440
  - declarar, 441
  - derivar de otros ADTs, 445-448
  - funciones virtuales puras, 440
  - Java, 450
  - listado, 440-441
  - ventajas, 452
  - cin, 563
  - espacios de nombres, 605
  - macros, 755
- tokens, 748**
- tomar la dirección de una referencia (listado), 262**
- trabajos, estado, 841**
- transformaciones (diseño), 643-644**
- Tres Amigos, 623**
- TrollTech, biblioteca de gráficos de Qt, 899**
- TrollTech, sitio Web, 923**
- try, bloques, 716, 721**
- excepciones
  - atrapar, 722
  - sin errores, 739
  - localizar, 722
  - sintaxis, 721
- tubería (l), carácter, 91-92, 839**
- tuberías, 561, 874**
- con nombre, 877-879
  - redirección, 839
  - semidúplex, 876-877
- typedef, instrucción, 475-477**
- typedef, palabra reservada, 53**

**U****UML (Lenguaje de Modelado****Unificado), 622**

clases, 622

clases derivadas, 334

clases especializadas, 631

CRC, tarjetas, 648

diagrama de interacción, 637

estereotipo discriminador, 653

herencia, 622

programación orientada a objetos, diseño, 622

discriminadores, 653  
relaciones de clases, 648

transformar tarjetas CRC en, 648

**unidades**autocontenido, 13  
de traducción, 614**unir, 870****UNIX**

AT&amp;T UNIX System V, 879

C, compiladores, 816

ELF, 818

KDE, 901

shells, 836

vim, editor, 809

XINU, 8

**usar**directiva, 609-611, 614  
palabra reservada, 609**UsarValActual, parámetro, 298****Usenet, grupos de noticias, 786****uso apropiado de llaves con instrucciones if (listado), 90****uso de mayúsculas (estilos de código), 783-784****usuarios, 11****V****valores, 43**asignar, 236  
a variables, 51-52, 149-150  
booleanos, 79  
caracteres, 56  
centinela, 312  
constantes enumeradas, 61  
decrementar, 75  
expresiones, 69  
incrementar, 75  
intercambiar, 270  
interinos, imprimir, 767  
listas de parámetros, 100  
main(), función, 30  
múltiples  
regresar con apuntadores, 272-274  
regresar con referencias, 274-275parámetro (predeterminado), 117-118  
pasar, 113eficiencia, 275  
listado, 512-514  
por referencia, 266-268, 276-278  
por valor, 267-268

predeterminados (listado), 296-297

listado, 296-297  
métodos, 296-298, 329  
uso de funciones sobre-

cargadas, 298

probar (Linux), 31

regresar, 114

con apuntadores (listado), 272-273  
con referencias (listado), 274-275  
mediante operador de posfijo, 315  
regresos fraccionarios, 73  
sí/no (1/0), 951**variables**

asignar, 236, 329

declarar, 228, 231-232

apuntadores, 228, 231

atributos, 228

constantes, 228, 231

constantes, 231

constantes, 231-234

constantes, 231-234

constantes, 231-234

declarar, 228, 231-234

descartados atributos, 231-234

desreferenciar, 229, 230, 234

direcciones de memoria, 232-233

funciones, 463-477

initializar, 228, 234

manipulación de datos, 231-232

métodos, 477-482

nombrar, 229

nulos, 228

perdidos, 228

pisotear, 249

propiedad, 290

reasignar, 239

RTTI, 416

this, 246-247, 313-314

ventajas, 234

asignar, clases definidas por el usuario, 324-325

bloques, 110-111

char, 45, 56

caracteres de escape, 58

codificación de caracteres, 57

tamaños, 56

ciclos for, alcance, 202

ent., 608

condición (directivas de sincronización), 868-869

- crear, 60  
 datos miembro, 147  
 declarar, 285  
 definir, 44, 48-49  
     de tipo COLOR, 61  
     múltiples, 51  
 demostrar el uso de, 52  
 enteros, 45  
     long, 54  
     short, 54  
     signed, 46, 55-56  
     unsigned, 46, 54-55  
 entorno, 839-840  
 establecer por medio del shell, 840  
 externas, limitar alcance, 604  
 globales, 108-109, 130  
     limitaciones, 109-110  
     ocultar, 108  
 inicializar, 51, 109  
 línea de comandos, 839  
 locales, 106-108, 839  
     alcance, 110-111  
     espacios de nombres, 610  
     inicializar, 111  
     parámetros, 106-107  
     persistencia, 235  
 miembro  
     declarar, 154  
     inicializar, 301  
     uso de memoria, 54  
 nombres, 44, 48  
     claridad, 49  
     notación de camelio, 50  
     notación húngara, 50  
     palabras reservadas, 50  
     precauciones, 48  
     susceptibilidad al uso de mayúsculas y minúsculas, 49-50  
 ocultas, 246  
 prefijos, 147  
 punto flotante, 47  
 RAM, 44  
 shells, 846  
 sustitución, 844  
 tamaños, 44-48  
 tipos  
     crear, 145-146  
     definición, 53  
     lista de, 47  
     valores, 48, 51-52, 149-150, 159  
         actuales, imprimir, 767-768  
         variables de conteo, 201  
**vectores, crear, 694-697**  
**vectoriales, contenedores, 692-694, 698-699**  
**Ventana, espacio de nombres, 607-608**  
**ventanas, 896-898**  
**ver**  
     información gráfica, 898  
     IPC, estado de objetos, 881  
     trabajos actualmente en ejecución, 841  
**verdad, 79, 93, 951**  
**verdadero/falso, operaciones, 93-94**  
**vi, editor, 809**  
     ayuda en línea, 811-812  
     etiquetas (tags), 815  
     iniciar, 809-810  
     modo de comandos, 810  
     modo de inserción, 810  
     modo ex, 811  
     navegar, 811  
**vim, editor, 809**  
**violaciones de la interfaz, 165-166**  
**virtuales, constructores de copia, 360-363**  
**visibilidad, 602**  
**visión, 624-625**  
**vistas (diseño, programación orientada a objetos), 644**  
**Visual Basic, 10**  
**visualización, 639**  
**VM (máquina virtual), 10**  
**void main(), función, 116**  
**void, valores (funciones), 37, 114**  
**VolumenCubo(), función, 118**
- W**
- wait(), llamada al sistema, 856**  
**watch, instrucciones, 788**  
**while, ciclos, 183-186**  
     break, instrucción, 186-189  
     ciclos eternos, 189-190  
     ciclos for, comparación, 212  
     condiciones de inicio, 194  
     continue, instrucción, 186-189  
     do...while, 192-193  
         comparación, 212  
         sintaxis, 193  
     ejecutar, 191-192  
     expresiones complejas, 185-186  
     limitaciones, 191  
     listados, 184-186, 189-190  
     regresar al inicio de, 186  
     salir, 186  
     sintaxis, 184-185  
**widgets, 903-904**  
     crear (GNOME), 907  
     cuadro de lista, 931  
**width(), método, 577-578**  
**Window class (kit de herramientas de wxWindows)**  
     agregar menús a, 920-921  
     KDE, 926-927  
**write(), método, 575-577**  
**wxGTK**  
     biblioteca, 903, 909  
     manejo de eventos, 918-919  
**wxStudio, 922**  
**wxWindows, kit de herramientas, 909**  
     agregar botones, 913-917  
     crear aplicaciones, 910-912

funcionalidad, 909-910  
procesamiento de eventos,  
919-920  
recursos en línea, 923  
señales, 919-920  
Window, clase  
    agregar botones a, 913  
    agregar menús, 920-921  
wxStudio, 922

## X

X, clientes, 898  
X, Organización, 898  
X, Protocolo, 898  
X Windows, Sistema, 809, 898  
Xerox Palo Alto, Laboratorio  
    de Investigación, 897  
XINU, 8  
Xlib, 898  
xLímite, excepción, 738  
xxgdb, 825



ABR

---

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

EMPRESA CERTIFICADA POR EL  
INSTITUTO MEXICANO DE NORMALIZACIÓN  
Y CERTIFICACION A.C. BAJO LAS NORMAS  
ISO-9002:1994/NMX-CC-004:1995  
CON EL NO. DE REGISTRO RSC-048  
E ISO-14001:1996/NMX-SAA-001:1998 IMNC/  
CON EL NO. DE REGISTRO RSAA-003

---



# **Lea esto antes de abrir el software**

Al abrir este paquete, manifiesta estar de acuerdo con lo siguiente:

No puede copiar ni redistribuir este CD-ROM en su totalidad. La copia y redistribución de los programas individuales de software que vienen en este CD-ROM están reguladas por los términos establecidos por los respectivos poseedores del derecho de autor.

El instalador y el código del(de los) autor(es) están protegidos por el editor y el(los) autor(es) mediante los derechos de autor. Los programas individuales y demás elementos que vienen en el CD-ROM son propiedad registrada de sus respectivos autores o poseedores del derecho de autor. Algunos de los programas incluidos con este producto pueden estar regulados por la Licencia pública general de GNU, la cual permite la redistribución; para obtener más información, lea la licencia para cada producto.

En el CD-ROM se incluyen otros programas con permiso especial otorgado por sus autores.

Este software se proporciona "como está" sin garantía de ningún tipo, ni expresa ni implícita, incluyendo, pero no limitado a, las garantías implícitas de comercialización y adecuación a un propósito en particular. Ni el editor ni sus vendedores o distribuidores asumen la responsabilidad de cualquier daño supuesto o real que se produzca debido al uso de este programa. (Algunos estados no permiten la exclusión de garantías implícitas, por lo que esta exclusión podría no aplicarse en su caso.)

004  
LIB

# Instalación del CD-ROM

## Instrucciones de instalación para Windows 95/98/NT/2000

1. Inserte el disco compacto en la unidad de CD-ROM.
2. En el escritorio de Windows 95, haga doble clic en el ícono Mi PC (My Computer).
3. Haga doble clic en el ícono que representa la unidad de CD-ROM en su computadora.
4. Abra el archivo README.txt para obtener una descripción de los productos proporcionados por terceros.

## Instrucciones de instalación para Linux y UNIX

Estas instrucciones de instalación asumen que usted está familiarizado con los comandos de UNIX y con la configuración básica de su equipo. Ya que existen varias versiones de UNIX, sólo se utilizan comandos genéricos. Si llega a tener problemas con estos comandos, por favor consulte la página man apropiada, o póngase en contacto con el administrador de sistemas.

Inserte el disco compacto en la unidad de CD-ROM.

Si tiene un administrador de volúmenes, el montaje del CD-ROM será automático. Si no tiene un administrador de volúmenes, puede montar el CD-ROM escribiendo lo siguiente:

**Mount -tiso9660 /dev/cdrom /mnt/cdrom**

### Nota

/mnt/cdrom es sólo un punto de montaje, pero debe existir cuando se emita el comando mount. También puede utilizar cualquier directorio vacío como punto de montaje, en caso de que no quiera utilizar /mnt/cdrom.

Abra el archivo readme.txt para obtener una descripción de los productos proporcionados por terceros.

## Precedencia y asociatividad de operadores

| Nivel | Descripción                                                                                                                                                                                                           | Operadores                        | Orden de evaluación    |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------|------------------------|
| 1     | Resolución de ámbito (binario, unario)                                                                                                                                                                                | ::                                | de izquierda a derecha |
| 2     | Llamadas a funciones, paréntesis, subíndice, selección de miembros, incremento y decremento de posijo                                                                                                                 | () [] . -> ++ --                  | de izquierda a derecha |
| 3     | sizeof, conversión implícita en C++, incremento y decremento de prefijo, más y menos unarios, negación, complemento, conversión implícita en C, sizeof(), de dirección, new de desreferencia, new[], delete, delete[] | + - ! ~ ( cast )                  | de derecha a izquierda |
| 4     | Selección de miembros para apuntador                                                                                                                                                                                  | & *                               |                        |
| 5     | Multiplicar, dividir, residuo                                                                                                                                                                                         | * / %                             | de izquierda a derecha |
| 6     | Suma, resta                                                                                                                                                                                                           | + -                               | de izquierda a derecha |
| 7     | Desplazamiento a nivel de bits                                                                                                                                                                                        | << >>                             | de izquierda a derecha |
| 8     | Desigualdad relacional                                                                                                                                                                                                | < <= > >=                         | de izquierda a derecha |
| 9     | Igualdad, desigualdad                                                                                                                                                                                                 | == !=                             | de izquierda a derecha |
| 10    | AND a nivel de bits                                                                                                                                                                                                   | &                                 | de izquierda a derecha |
| 11    | OR exclusivo a nivel de bits                                                                                                                                                                                          | ^                                 | de izquierda a derecha |
| 12    | OR a nivel de bits                                                                                                                                                                                                    |                                   | de izquierda a derecha |
| 13    | Lógico AND                                                                                                                                                                                                            | &&                                | de izquierda a derecha |
| 14    | Lógico OR                                                                                                                                                                                                             |                                   | de izquierda a derecha |
| 15    | Condicional                                                                                                                                                                                                           | ? :                               | de derecha a izquierda |
| 16    | Operadores de asignación                                                                                                                                                                                              | = *= /= %= += -= <<= >>= &=  = ^= | de derecha a izquierda |
| 17    | Coma                                                                                                                                                                                                                  | ,                                 |                        |

Los operadores que están en la parte superior de la tabla tienen mayor precedencia que los operadores de la parte inferior. En expresiones que empiecen con argumentos en el conjunto de paréntesis más interno (en caso de haber), los programas evalúan los operadores de mayor precedencia antes de evaluar los operadores de menor precedencia.

A falta de paréntesis de aclaración, los operadores del mismo nivel se evalúan de acuerdo con su orden de evaluación, ya sea de izquierda a derecha o de derecha a izquierda.

### Operadores que pueden sobrecargarse

|     |    |     |    |     |     |        |    |    |     |     |
|-----|----|-----|----|-----|-----|--------|----|----|-----|-----|
| *   | /  | +   | -  | %   | ^   | &      |    | -  |     | ,   |
| =   | <  | >   | <= | >=  | ++  | -      | << | >> | ==  | =   |
| &&  |    | *=  | /= | %=  | ^=  | &=     | =  | += | - = | <<= |
| >>= | -> | ->* | [] | ( ) | new | delete |    |    |     |     |

Los operadores +, -, \* y & pueden sobrecargarse para expresiones binarias y unarias. Los operadores ., .\*, ::, ?: y sizeof no pueden sobrecargarse. Además, =, (), {} y -> deben implementarse como funciones miembro no estáticas.

### Plantillas de clase contenedora estándar

| Plantilla de clase | Descripción                                                                                                                                 |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| vector             | Secuencia lineal, similar a un arreglo de C++.                                                                                              |
| list               | Lista doblemente enlazada.                                                                                                                  |
| deque              | Cola con dos extremos.                                                                                                                      |
| set                | Arreglo asociativo de claves únicas. Un tipo especial de un conjunto capaz de guardar valores binarios, se conoce como un conjunto de bits. |

continúa

continuación

| Plantilla de la clase       | Descripción                                                             |
|-----------------------------|-------------------------------------------------------------------------|
| <code>multiset</code>       | Arreglo asociativo de claves posiblemente duplicadas.                   |
| <code>map</code>            | Arreglo asociativo de claves y valores únicos.                          |
| <code>multimap</code>       | Arreglo asociativo de claves y valores posiblemente duplicados.         |
| <code>stack</code>          | Estructura de datos de tipo LIFO (Último en Entrar, Primero en Salir).  |
| <code>queue</code>          | Estructura de datos de tipo FIFO (Primero en Entrar, Primero en Salir). |
| <code>priority_queue</code> | Cola o vector ordenado por evento crítico.                              |

### Tipos de datos de C++

| Tipo                            | Tamaño   | Rango                                                                       |
|---------------------------------|----------|-----------------------------------------------------------------------------|
| <code>unsigned short int</code> | 2 bytes  | 0 a 65,535                                                                  |
| <code>short int</code>          | 2 bytes  | -32,768 a 32,767                                                            |
| <code>unsigned long int</code>  | 4 bytes  | 0 a 4,294,967,295                                                           |
| <code>long int</code>           | 4 bytes  | -2,147,483,648 a 2,147,483,647                                              |
| <code>int</code>                | 4 bytes  | -2,147,483,648 a 2,147,483,647                                              |
| <code>unsigned int</code>       | 4 bytes  | 0 a 4,294,967,295                                                           |
| <code>char</code>               | 1 byte   | 256 valores de tipo carácter, con signo de forma predeterminada, -128 a 127 |
| <code>wchar_t</code>            | 4 bytes  | 4,294,967,296 valores tipo de carácter                                      |
| <code>bool</code>               | 1 byte   | Verdadero (true) o falso (false)                                            |
| <code>float</code>              | 4 bytes  | 1.2e-37 a 3.4e38                                                            |
| <code>double</code>             | 8 bytes  | 2.2e-307 a 1.8e308                                                          |
| <code>long double</code>        | 10 bytes | 3.4e-4931 a 1.1e+4932                                                       |

Tenga en cuenta que estos tamaños son típicos y pueden variar según la implementación.

### Las constantes `open_mode` de la clase `ios`

| Constante               | Estándar | Efecto                                                                                                               |
|-------------------------|----------|----------------------------------------------------------------------------------------------------------------------|
| <code>app</code>        |          | La siguiente operación de escritura agrega nueva información al final del archivo.                                   |
| <code>ate</code>        | *        | Busca hasta el final del archivo al abrirse. La palabra "ate" significa "at end" (al final).                         |
| <code>binary</code>     | *        | Abre el archivo en modo binario (no texto).                                                                          |
| <code>in</code>         | *        | Abre el archivo para entrada (lectura).                                                                              |
| <code>norecreate</code> |          | Si el archivo no existe, no se crea un nuevo archivo.                                                                |
| <code>noreplace</code>  |          | Si el archivo ya existe, no se sobreescribe.                                                                         |
| <code>out</code>        | *        | Abre el archivo para salida (escritura).                                                                             |
| <code>trunc</code>      |          | Abre y trunca un archivo existente. La nueva información que se escribe en el archivo reemplaza su contenido actual. |

### Palabras reservadas de C y C++

|                       |                               |                          |                         |
|-----------------------|-------------------------------|--------------------------|-------------------------|
| <code>auto</code>     | <code>bool</code>             | <code>break</code>       | <code>case</code>       |
| <code>char</code>     | <code>class</code>            | <code>const</code>       | <code>const_cast</code> |
| <code>default</code>  | <code>delete</code>           | <code>do</code>          | <code>double</code>     |
| <code>else</code>     | <code>enum</code>             | <code>explicit</code>    | <code>extern</code>     |
| <code>float</code>    | <code>for</code>              | <code>friend</code>      | <code>goto</code>       |
| <code>inline</code>   | <code>int</code>              | <code>long</code>        | <code>mutable</code>    |
| <code>new</code>      | <code>operator</code>         | <code>private</code>     | <code>protected</code>  |
| <code>register</code> | <code>reinterpret_cast</code> | <code>return</code>      | <code>short</code>      |
| <code>sizeof</code>   | <code>static</code>           | <code>static_cast</code> | <code>struct</code>     |
| <code>template</code> | <code>this</code>             | <code>throw</code>       | <code>true</code>       |
| <code>typedef</code>  | <code>typeof</code>           | <code>typeid</code>      | <code>typename</code>   |
| <code>unsigned</code> | <code>using</code>            | <code>virtual</code>     | <code>void</code>       |
| <code>while</code>    |                               |                          |                         |

evé sus conocimientos  
siguiente nivel

sólo 21 días tendrá los conocimientos necesarios  
trabajar en forma eficiente con C++ para Linux. Con  
esta de este tutorial dominará los aspectos básicos para  
es adentrarse en características y conceptos más avanzados.

mprenda los fundamentos de la programación en C++ para Linux  
mine todas las características nuevas y avanzadas que ofrece C++ para  
ux

ienda a utilizar en forma efectiva las herramientas y características más  
entes de C++ para Linux, mediante ejemplos prácticos y reales  
oveche los consejos de una de las principales autoridades de la progra-  
ción en C++ para Linux en el ambiente empresarial  
ro está diseñado para adaptarse a su propio plan de aprendizaje. Puede  
as lecciones paso a paso, capítulo por capítulo, o elegir sólo las leccio-  
le interesen.

#### D-ROM incluye:

odo el código fuente que se usa en el libro

utilerías de wxWindows creadas por otros fabricantes

mandrake 7.0 para Linux, que incluye un entorno completo  
de desarrollo con el compilador C/C++ de GNU, el depurador  
GDB de GNU, la utilería make de GNU, GTK+, KDE, GNOME  
mucho más



Horvath es autor de varios libros sobre C++, entre ellos *C++ para Prin-*

Es presidente de Liberty Associates, Inc., empresa que proporciona  
ción a domicilio sobre desarrollo de software orientado a objetos, con-  
programación por contrato.

Horvath, CCP, es consultor y profesor adjunto de medio tiempo en  
ciudades locales, donde imparte clases sobre la programación en C, UNIX  
y para bases de datos. Es autor de numerosos artículos para revistas y  
sobre UNIX y Linux, entre ellos *UNIX for the Mainframe, Red Hat  
Unleashed* y *UNIX Unleashed, Second Edition*. David participa con re-  
como orador en conferencias internacionales.

: Programación/Sistemas operativos

++ para Linux

#### Nivel de usuario:

Principiante Intermedio Experto

Visítenos en:  
[www.pearsonedlatino.com.mx](http://www.pearsonedlatino.com.mx)

Aprendiendo®

# C++ para Linux® en 21 Días

- Programe específicamente para Linux con GNU C++
- Descubra cómo escribir, compilar y enlazar su primer programa completamente funcional en C++
- Aprenda la programación por procedimientos y orientada a objetos desde sus bases
- Conozca el C++ estándar de ANSI/ISO y la Biblioteca Estándar de Plantillas
- Vaya más allá de los fundamentos de la programación en C++ para crear aplicaciones robustas y completas
- Incluye una semana adicional con temas como
  - El entorno de programación de Linux
  - Programación de shells
  - Programación de sistemas
  - Comunicación entre procesos
  - Programación de GUIs

ISBN 970-26-0012-X

90000



9 789702 600121